Linux の boot プロセスと APM の起動

指導教員 清水 尚彦 助教授

0AET2210 能登屋　修一

東海大学 工学部 通信工学科

目 次

図 目 次

1　はじめに

Linux のブートプロセスを解析することによって、OS に必要なプログラム知識、および OS が必要な機能、システムが理解できる。そして、バグ等の原因究明するためにも必要である。

2 Bootsect

2.1 起動

ＰＣの電源を入れることによってメモリの 0xffffff0 番地へジャンプする。ここには、BIOS(Basic Input Output System) の ROM が割りふられている。BIOS が起動すると電源投入時状態テスト（POST）を実行して、HDD 等から OS 起動用のコードを、0x07C00(図 1:35 行目) から 512 バイト分メモリに書き込む。（521 バイトはフロッピーディスクの 1 セクタ分）そして、0x7C00 からプログラムを実行する。

2.2 90000H への転送

Linux の boot は bootsect.S が読み込まれることによって始まる。bootsect.S は BIOS の起動ルーチンにより読み込まれ、07C00H から 07DFFH の 512 バイトの間に読み込まれている。始めに、bootsect.S は自分自身を 90000H(図 1:36 行目 9000H は (図 2:5 行目) で定義されている。) へ移動させる

```
34 SETUPSECTS      = 4                   /* default nr of setup-sectors */
35 BOOTSEG         = 0x07C0              /* original address of boot-sector */
36 INITSEG         = DEF_INITSEG         /* we move boot here - out of the way */
37 SETUPSEG        = DEF_SETUPSEG        /* setup starts here */
38 SYSSEG          = DEF_SYSSEG          /* system loaded at 0x10000 (65536) */
39 SYSSIZE         = DEF_SYSSIZE         /* system size: # of 16-byte clicks */
40                                       /* to be loaded */
41 ROOT_DEV        = 0                   /* ROOT_DEV is now written by "build" */
42 SWAP_DEV        = 0                   /* SWAP_DEV is now written by "build" */
```

図 1: Linux/arch/i386/boot/bootsect.S(34-42)

```
4 /* Don't touch these, unless you really know what you're doing. */
5 #define DEF_INITSEG     0x9000
6 #define DEF_SYSSEG      0x1000
7 #define DEF_SETUPSEG    0x9020
8 #define DEF_SYSSIZE     0x7F00
9
10 /* Internal svga startup constants */
11 #define NORMAL_VGA      0xffff          /* 80x25 mode */
12 #define EXTENDED_VGA    0xfffe          /* 80x50 mode */
13 #define ASK_VGA         0xfffd   /* ask for it at bootup */
```

図 2: Linux/include/asm-i386/boot.h

```
62 # First things first. Move ourself from 0x7C00 ->
63                                                  0x90000 and jump there.
64         movw    $BOOTSEG, %ax
65         movw    %ax, %ds                # %ds = BOOTSEG
66         movw    $INITSEG, %ax
67         movw    %ax, %es                # %ax = %es = INITSEG
68         movw    $256, %cx
69         subw    %si, %si
70         subw    %di, %di
71         cld
72         rep
73         movsw
74         ljmp    $INITSEG, $go
```

図 3: Linux/arch/i386/boot/bootsect.S(62-74)

分かりやすくするために、64行目から74行目のソースを解析してみる。

```
64 ax レジスタに 0x7C00 を代入する
65 dx レジスタに 0x7C00 を代入する
66 ax レジスタに 0x90000 を代入する
67 ex レジスタに 0x90000 を代入する
68 cx レジスタに 256 を代入する
69 si レジスタを 0 クリア
70 di レジスタを 0 クリア
71 EFLAGS レジスタの DF フラグをクリア
72 cx レジスタの値だけ次に実行される命令を繰り返す
73 ds:si レジスタの値を es:di レジスタに転送し,DF フラグの値を参照して
   si :di に転送するバイト数を加算もしくは減算する

74 セグメント 0x90000 オフセット 0x0 にジャンプ
```

図 4: bootsect.S 解析

bootsect.S を移動させるためのコードは図3である。ここで注意したいのが、71行目で DF フラグがクリアされていて、73行目の movsw は転送するバイト数を加算する。mov”w”や movs”w”の w は2バイトを意味している。ゆえに、si di は2づつ加算されていく。その結果 es レジスタの値も2づつ増える。つまり、256 × 2で512バイト分のデータを 0x9000 から 0x91FF まで転送できる。転送が終わると INITSEG の go までジャンプする。

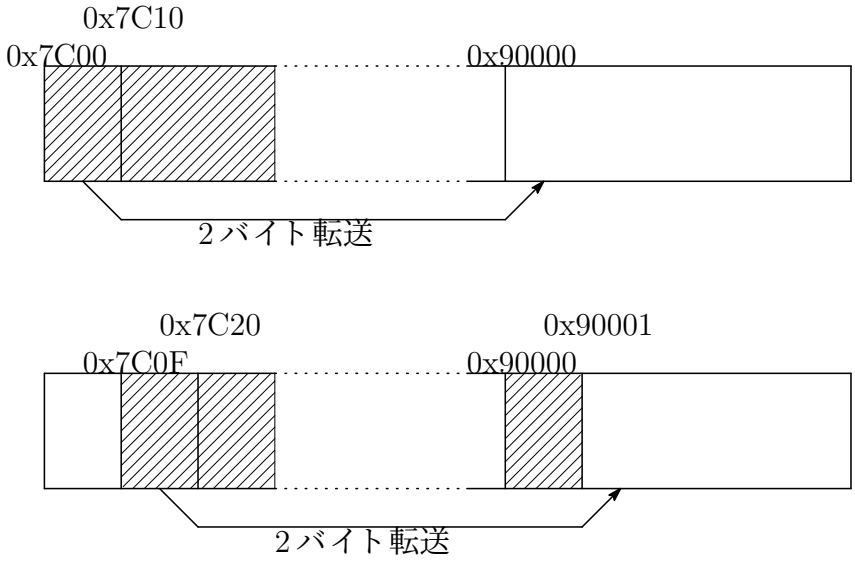

図 5: 転送方式

```
82 go:movw $0x4000-12, %di    # 0x4000 is an arbitrary value >=
83                            # length of bootsect + length of
84                            # setup + room for stack;
85                            # 12 is disk parm size.
86    movw %ax, %ds           # %ax and %es already contain INITSEG
87    movw %ax, %ss
88    movw %di, %sp           # put stack at INITSEG:0x4000-12.
```

図 6: Linux/arch/i386/boot/bootsect.S(82-88)

(図 6:82 行目) の 0x4000-12 という値は

bootsect、setup ルーチンのサイズ、スタック領域に十分なザイズを足し合わせたものである。-12 は disk parm のサイズである。

2.3 Loadingの表示

フロッピーやHDDからデータを読み込む前に、ディスプレーに「Loding」の文字を表示する。

```
141 got_sectors:
142         movb    $0x03, %ah      # read cursor pos
143         xorb    %bh, %bh
144         int     $0x10
145         movw    $9, %cx
146         movb    $0x07, %bl      # page 0, attribute 7 (normal)
147                                 # %bh is set above; int10 doesn't
148                                 # modify it
149         movw    $msg1, %bp
150         movw    $0x1301, %ax    # write string, move cursor
151         int     $0x10           # tell the user we're loading..

 *

411 msg1:           .byte 13, 10
412                 .ascii "Loading"
```

図 7: Linux/arch/i386/boot/bootsect.S(141-151)

図 7:144 行目で特殊な BIOS コールを使う。Int 0x10, AH=0x03 で呼びだされるのは VIDEO - GET CURSOR POSITION AND SIZE （表 1）である。その名前のとおりビデオのカーソルの位置とサイズを調べるために使われる、BIOS コールである。次に INT 10 - VIDEO - WRITE STRING（表 2）で呼ばれる BIOS コールは、ディスプレーに文字を表示するためのものである。

図 7:149 行目で文字列が格納されるアドレスが設定されている。msg1 は 411 行目で定義されていて、411 行目の「13,10」は改行と復帰文字の意味である。

IN	
AH	03h
BH	page number 0-3 in modes 2&3 0-7 in modes 0&1 0 in graphics modes
RETURN	
AX	0000h (Phoenix BIOS)
CH	start scan line
CL	end scan line
DH	row (00h is top)
DL	cloumn (00h is left)

表 1: VIDEO - GET CURSOR POSITION AND SIZE

IN	
AH	13h
AL	write mode bit 0 : update cursor after writing bit 1: string contains alternating characters and attributes string contains alternating characters and attributes bits 2-7 : reserved (0)
CX	number of characters in string
DH,DL	row,column at which to start writing
ES:BP	string to write
RETURN	nothing

表 2: INT 10 - VIDEO - WRITE STRING (AT and later,EGA)

2.4 スタートアップルーチンの読み込み

この時点で bootsect.S は 0x90000 から 0x901FF にある。そのために、このエリアのメモリは使用することができない。そこで、セットアップルーチンを 0x09200 以降にロートするように設定する。

```
157  movw    $0x0001, %ax     # set sread (sector-to-read) to 1 as
158  movw    $sread, %si      # the boot sector has already been read
159  movw    %ax, (%si)
160
161  xorw    %ax, %ax         # reset FDC
162  xorb    %dl, %dl
163  int     $0x13
164  movw    $0x0200, %bx     # address = 512, in INITSEG
165 next_step:
166  movb    setup_sects, %al
167  movw    sectors, %cx
168  subw    (%si), %cx       # (%si) = sread
169  cmpb    %cl, %al
170  jbe     no_cyl_crossing
171  movw    sectors, %ax
172  subw    (%si), %ax       # (%si) = sread
173 no_cyl_crossing:
174  call    read_track
175  pushw   %ax              # save it
176  call    set_next         # set %bx properly; it uses %ax,%cx,%dx
177  popw    %ax              # restore
178  subb    %al, setup_sects # rest - for next step
179  jnz     next_step
```

図 8: Linux/arch/i386/boot/bootsect.S(157-179)

図 8:157 行目で一番初めのセクタに入っていた、bootsect.S は読み込まれたので次のセクタに変更する。164 の 0x0200 は、セットアップルーチンを読み込むためのオフセット値であり、セグメントの値は ex レジスタの 0x90000 である。178 行目の setup_sects が 0 になるまで、セクタを次々に読み込む。

2.5 64KB 転送

セットアップルーチンを読み込んだ後、64KBづつデータを特殊な BIOS コールを使って 1 MB 以上のメモリに転送する。

```
840 bootsect_helper:
841  cmpw    $0, %cs:bootsect_es
842  jnz     bootsect_second
843
844  movb    $0x20, %cs:type_of_loader
845  movw    %es, %ax
846  shrw    $4, %ax
847  movb    %ah, %cs:bootsect_src_base+2
848  movw    %es, %ax
849  movw    %ax, %cs:bootsect_es
850  subw    $SYSSEG, %ax
851  lret                            # nothing else to do for now
852
853 bootsect_second:
854  pushw   %cx
855  pushw   %si
856  pushw   %bx
857  testw   %bx, %bx                # 64K full?
858  jne     bootsect_ex
859
860  movw    $0x8000, %cx            # full 64K, INT15 moves words
861  pushw   %cs
862  popw    %es
863  movw    $bootsect_gdt, %si
864  movw    $0x8700, %ax
865  int     $0x15
866  jc      bootsect_panic            # this, if INT15 fails
867
868  movw    %cs:bootsect_es, %es     # we reset %es to always point
869  incb    %cs:bootsect_dst_base+2  # to 0x10000
```

図 9: Linux/arch/i386/boot/setup.S(840-869)

ここで一旦 bootsect.S から setup.S に移動する。bootsect_helper が始めて呼び出されたとき、bootsect_second は 0 なので図 9:844〜851 行目までが実行される。

844 行目で 0x20 を書き込んでいるのは、ブートローダの種類とバージョンを設定するためである。このアドレスの意味と内容は Linux の規格で決められている。そのため後でセットアップ・ルーチンがこの値を参照できるように、設定している。この数字の上位４ビットがブートローダを識別するＩＤで下位 4 ビットがバージョンを表す。

No	ブートローダタイプ
0	LILO
1	Loadlin
2	bootsect-loader
3	SYSLINUX
4	Etherboot

表 3: ブートローダの種類

830 行目の bootsect_es は、es レジスタの内容を保存している。ここでの値は 0x1000 であり、SYSSEG も同じ値であるためこのルーチンから抜ける。2 回目以降は必ず bootsect_second にジャンプする。bx レジスタの値が 0 になるまで繰り返す。bx が 0 になったとき物理アドレス 0x1000 に、0x0000 から 0xFFFF までのデータが読み込まれたとこになり、841 行目〜846 行目までが実行される。ここで 0x0000〜0xFFFF までのデータが１ＭＢ以降の領域に転送される。

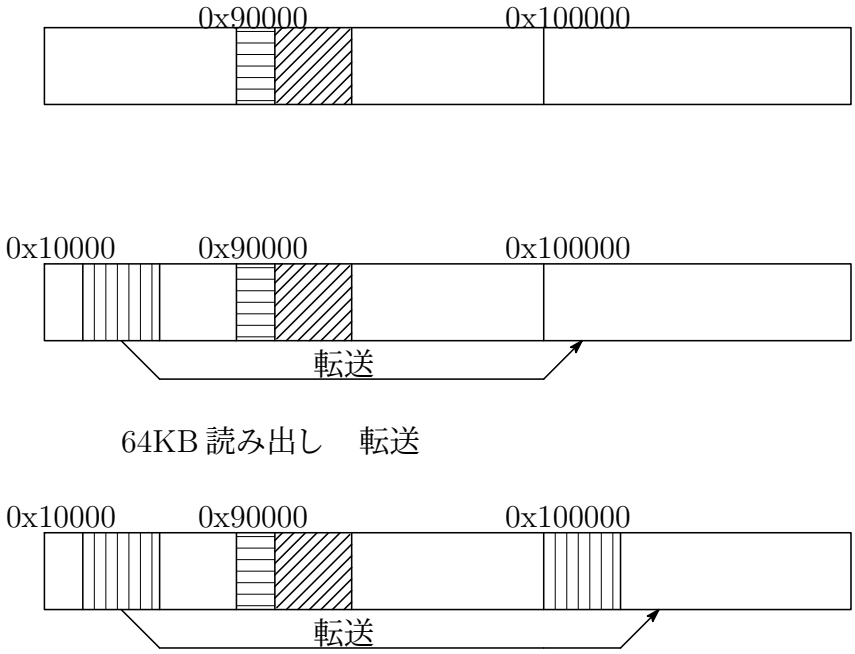

図 10: 64KB 転送方式

リアルモードのプログラムは 1MB 以降のメモリにアクセスできないため、図 9:866 行目で INT 15 - SYSTEM - COPY EXTENDED MEMORY 表 4 という特別な BIOS コールを使って 1MB 以降のメモリにアクセスする。

IN	
AH	87h
CX	number of words to copy
ES:SI	global descriptor table
RETURN	
CF	clear if successful
AH	status

表 4: INT 15 - SYSTEM - COPY EXTENDED MEMORYINT

2.6 ルートパーティションの取得

カーネルが起動時にどのディスクのパーティションをルートにするのか、モニターのモードは何か、などを記録する場所である。

```
195  movw    root_dev, %ax
196  orw     %ax, %ax
197  jne     root_defined
198
199  movw    sectors, %bx
200  movw    $0x0208, %ax      # /dev/ps0 - 1.2Mb
201  cmpw    $15, %bx
202  je      root_defined
203
204  movb    $0x1c, %al        # /dev/PS0 - 1.44Mb
205  cmpw    $18, %bx
206  je      root_defined
207
208  movb    $0x20, %al       # /dev/fd0H2880 - 2.88Mb
209  cmpw    $36, %bx
210  je      root_defined
211
212  movb    $0, %al           # /dev/fd0 - autodetect
213 root_defined:
214         movw    %ax, root_dev

  *

414 # XXX: This is a fairly snug fit.
415
416 .org 497
417 setup_sects:    .byte SETUPSECTS
418 root_flags:     .word ROOT_RDONLY
419 syssize:        .word SYSSIZE
420 swap_dev:       .word SWAP_DEV
421 ram_size:       .word RAMDISK
422 vid_mode:       .word SVGA_MODE
423 root_dev:       .word ROOT_DEV
424 boot_flag:      .word 0xAA55
```

図 11: Linux/arch/i386/boot/bootsect.S(195-214)

通常は、bootsect の最後の 15 バイトに（図 11:423 行目）ROOT_DEV が書き込まれる。197 行目から 214 行目にジャンプして toot_DEV を書き込む。

2.7 セットアップルーチンの起動

```
37   SETUPSEG    = DEF_SETUPSEG   /*setup starts here */
*
219   ljmp   $SETUPSEG,  $0
```

図 12: Linux/arch/i386/boot/bootsect.S(37,219)

図 2 より DEF_SETUPSEG は 0x9020 にロード されている。jamp 命令により、セグメント 0x9020、オフセット 0x0 へジャンプする。これで bootsect.S の動作が終了し、次の setup.S が起動する。

3 Setup

セットアップルーチンではBIOSからバードウエア情報を取得する。そしてハードウエアの初期化などをおこなって、カーネル起動に必要な情報や環境をつくる。
セットアップルーチンのなかで、APMの起動等もおこなわれる。

3.1 起動環境チェック

```
168 start_of_setup:
169     # Bootlin depends on this being done early
170         movw          $0x01500, %ax
171         movb          $0x81, %dl
172         int           $0x13
```

図 13: Linux/arch/i386/boot/setup.S(169-172)

始めにHDDのディスクタイプをBIOSコールINT 13 - DISK - GET DISK TYPE図13:172行目を使って取得する。(コールするだけで、ここで得た値は使用しない？)

IN	
AH	15h
DL	drive number (bit 7 set for hard disk
RETURN	
CF	clear if successful
AH	type code
	01h:floppy without change-line support
	02h:floppy (or other removable drive) with change-line support
	03h: hard disk
CX:DX	number of 512-byte sectors

表 5: INT 13 - DISK - GET DISK TYPE

```
181 # Set %ds = %cs, we know that SETUPSEG = %cs at this point
182         movw    %cs, %ax                # aka SETUPSEG
183         movw    %ax, %ds
184 # Check signature at end of setup
185         cmpw    $SIG1, setup_sig1
186         jne     bad_sig
187
188         cmpw    $SIG2, setup_sig2
189         jne     bad_sig
190
191         jmp     good_sig1
 *
1036 # Setup signature -- must be last
1037 setup_sig1:     .word   SIG1
1038 setup_sig2:     .word   SIG2
```

図 14: Linux/arch/i386/boot/setup.S(181-191)

次にセットアップルーチンは、自分自身が正しく読み出されたか調べる。setup_sig はセットアップルーチンの一番最後にくるマーカである。

3.2 BIOSチェック

3.2.1 メモリチェック

効率よくメモリを使うためには、ＰＣがどのぐらいのメモリを搭載しているか調べる必要がある。BIOSコールの中にメモリマップやサイズを教えてくれるものがあるその機能を使用して、どれだけ物理メモリがあるか調べる。

- INT 15 AX=E820H - newer BIOSes - GET SYSTEM MEMORY MAP
  4GBを超えるメモリサイズを調べることができる。32bitのCPUだと最大メモリアクセスが4GB(2の32乗)までしかアクセスできない。そこで　PentiumはPAE（Physical Address Extension）を使用して64GB(2の36乗）までメモリを扱うことができる。

- INT 15 AX=E801H- Phoenix BIOS v4.0 - GET MEMORY SIZE FOR 64M CONFIGURATIONS
  4GBまでのメモリを調べることができる。

- INT 15 AX=88H- SYSTEM - GET EXTENDED MEMORY SIZE
  64MBまでのメモリを調べることができる。

セットアップルーチンはBIOSの種類にかかわらず全てのメモリマップを取得する。カーネル起動時にそれぞれのデータを参照してシステムが利用できるメモリを計算する。

```
309 meme820:
310     xorl    %ebx, %ebx              # continuation counter
311     movw    $E820MAP, %di           # point into the whitelist
312                                     # so we can have the bios
313                                     # directly write into it.
314
315 jmpe820:
316     movl    $0x0000e820, %eax       # e820, upper word zeroed
317     movl    $SMAP, %edx             # ascii 'SMAP'
318     movl    $20, %ecx               # size of the e820rec
319     pushw   %ds                     # data record.
320     popw    %es
321     int     $0x15                   # make the call
322     jc      bail820                 # fall to e801 if it fails
 *
386 #endif
387         movb    $0x88, %ah
388         int     $0x15
389         movw    %ax, (2)
```

図 15: Linux/arch/i386/boot/setup.S(309-389)

IN	
AX	E820h
EAX	0000E820h
EDX	534D4150h ('SMAP')
EBX	continuation value or 00000000h to start at beginning of map
ECX	size of buffer for result, in bytes (should be $\geqq$ 20 bytes)
ES:DI	buffer for result
RETURN	
CF	set on error
AH	error code (86h)

表 6: INT 15 - newer BIOSes - GET SYSTEM MEMORY MAP

IN	
AX	E801h
RETURN	CF clear if successful
AX BX CX DX	extended memory between 1M and 16M extended memory above 16M, in 64K blocks configured memory 1M to 16M, in K configured memory above 16M, in 64K blocks

表 7: INT 15 - Phoenix BIOS v4.0 - GET MEMORY SIZE FOR

IN	
AH	88h
RETURN	CF clear if successful
AX AH	number of contiguous KB starting at absolute address 100000h status (CF set on error) 80h invalid command (PC,PCjr) 86h unsupported function (XT,PS30)

表 8: INT 15 - SYSTEM - GET EXTENDED MEMORY SIZE (286+)

3.2.2 キーボード リピートレート

BIOS コールで INT 16 - KEYBOARD - SET TYPEMATIC RATE AND DELAY 呼び出し、キーボードのリピートレートを最大にする。

```
391 # Set the keyboard repeat rate to the max
392         movw    $0x0305, %ax
393         xorw    %bx, %bx
394         int     $0x16
```

図 16: Linux/arch/i386/boot/setup.S(391-394)

IN	
AH	03h
AL	subfunction
	00h set default delay and rate (PCjr and some PS/2)
	01h increase delay before repeat (PCjr)
	02h decrease repeat rate by factor of 2 (PCjr)
	03h increase delay and decrease repeat rate (PCjr)
	04h turn off typematic repeat (PCjr and some PS/2)
	05h set repeat rate and delay (AT,PS)
	BH = delay value (00h = 250ms to 03h = 1000ms)
	BL = repeat rate (00h=30/sec to 0Ch=10/sec [def] to 1Fh=2/sec)
	06h get current typematic rate and delay (newer PS/2s)
RETURN	
BL	repeat rate (above)
BH	BH = delay (above)
	AH destroyed by many BIOSes

表 9: INT 16 - KEYBOARD - SET TYPEMATIC RATE AND DELAY

3.2.3 VGA

サブルーチンを呼び出し、カーソル位置やページ番号を取得する。

```
398         call    video               # NOTE: we need %ds pointing
399                                     # to bootsector
```

図 17: Linux/arch/i386/boot/setup.S(390)

3.2.4 HDD

HDD のパラメータテーブルをデータ領域に書きこむ。

```
401 # Get hd0 data...
402         xorw    %ax, %ax
403         movw    %ax, %ds
404         ldsw    (4 * 0x41), %si
405         movw    %cs, %ax                # aka SETUPSEG
406         subw    $DELTA_INITSEG, %ax     # aka INITSEG
 *
438         movw    $0x10, %cx
439         xorw    %ax, %ax
440         cld
441         rep
442         stosb
443 is_disk1:
```

図 18: Linux/arch/i386/boot/setup.S(401-443)

3.2.5 マウスの確認

PS/2 マウスが接続されているか、確認する。

```
474 # Check for PS/2 pointing device
475  movw    %cs, %ax                # aka SETUPSEG
476  subw    $DELTA_INITSEG, %ax     # aka INITSEG
477  movw    %ax, %ds
478  movw    $0, (0x1ff)             # default is no pointing device
479  int     $0x11                   # int 0x11: equipment list
480  testb   $0x04, %al              # check if mouse installed
481  jz      no_psmouse
482
483  movw    $0xAA, (0x1ff)          # device present
484 no_psmouse:
```

図 19: Linux/arch/i386/boot/setup.c

BIOS コール、INT11 を使ってマウスの確認をする。接続されている場合にはアドレス 0x901ff に 0xaa を書き込む。

AXbit	RETURN
F	number of printer ports
E	number of printer ports
D	unused, internal modem (PS/2)
C	game adapter installed
B	number of serial ports
A	number of serial ports
9	number of serial ports
8	0 if DMA installed
7	# of diskette drives, less 1
6	# of diskette drives, less 1
5	initial video mode
4	initial video mode
3	not used on PS/2
2	pointing device installed (PS/2)
1	math coprocessor
0	IPL diskette installed

表 10: INT 11 - BIOS Equipment Determination

3.2.6 APM

APMBIOS が有効になっているとき、APM の起動ルーチンが起動する。BIOS コールを使い、BIOS が APM を使用することができるか調べる。APM が使用できるなら BIOS のメモリ上に書き込む。

INT 15 AX=0x5300 図 20:490 行目 は APMが使用できるか調べる BIOS コールであり、APM が使用できる時には、cx レジスタの 2bit 目を調べて 32bitAPM がサポートされているか調べる。32bit APM がサポートされていない場合 APM は起動しない。

```
488                                     # %ds points to the bootsector
489   movw    $0, 0x40                  # version = 0 means no APM BIOS
490   movw    $0x05300, %ax             # APM BIOS installation check
491   xorw    %bx, %bx
492   int     $0x15
493   jc      done_apm_bios             # Nope, no APM BIOS
494
495   cmpw    $0x0504d, %bx             # Check for "PM" signature
496   jne     done_apm_bios             # No signature, no APM BIOS
497
498   andw    $0x02, %cx                # Is 32 bit supported?
499   je      done_apm_bios             # No 32-bit, no (good) APM BIOS
```

図 20: Linux/arch/i386/boot/setup.S(488-499)

```
501   movw    $0x05304, %ax             # Disconnect first just in case
502   xorw    %bx, %bx
503   int     $0x15                     # ignore return code
504   movw    $0x05303, %ax             # 32 bit connect
505   xorl    %ebx, %ebx
506   xorw    %cx, %cx                  # paranoia :-)
507   xorw    %dx, %dx                  #   ...
508   xorl    %esi, %esi                #   ...
509   xorw    %di, %di                  #   ...
510   int     $0x15
511   jc      no_32_apm_bios            # Ack, error.
```

図 21: Linux/arch/i386/boot/setup.S(488-499)

INT 15 AX=0x5304(図 21:501 行目) は DISCONNECT INTERFACE である。APM のインターフェイスにコネクトできるか調べる。INT 15 0x05303 は、CONNECT 32-BIT PROTMODE INTERFACE(図 21:504 行目) で、32bit モードの 32,16bit 用セグメントアドレスを取得する BIOS コールである。値を受け取る前にレジスタを xor で 0 クリアしている。

図 22:513 行目からは、直接 INT 15 0x05303 で得た、セグメントやオフセットの値を BIOS に直接書き込んでいる。

図 22:521 行目で、もう一度 APM の起動のチェックをおこなう。エラーがあると apm_disconnect へジャンプする。2 回チェックして問題が無い場

```
513         movw    %ax,  (66)          # BIOS code segment
514         movl    %ebx, (68)          # BIOS entry point offset
515         movw    %cx,  (72)          # BIOS 16 bit code segment
516         movw    %dx,  (74)          # BIOS data segment
517         movl    %esi, (78)          # BIOS code segment lengths
518         movw    %di,  (82)          # BIOS data segment length
```

図 22: Linux/arch/i386/boot/setup.S(513-518)

```
521   movw    $0x05300, %ax         # APM BIOS installation check
522   xorw    %bx, %bx
523   xorw    %cx, %cx              # paranoia
524   int     $0x15
525   jc      apm_disconnect        # error -> shouldn't happen
526
527   cmpw    $0x0504d, %bx         # check for "PM" signature
528   jne     apm_disconnect        # no sig -> shouldn't happen
529
530   movw    %ax, (64)              # record the APM BIOS version
531   movw    %cx, (76)             # and flags
532   jmp     done_apm_bios
```

図 23: Linux/arch/i386/boot/setup.S(521-532)

```
534 apm_disconnect:                     # Tidy up
535   movw    $0x05304, %ax             # Disconnect
536   xorw    %bx, %bx
537   int     $0x15                     # ignore return code
538
539   jmp     done_apm_bios
540
541 no_32_apm_bios:
542         andw    $0xfffd, (76)       # remove 32 bit support bit
543 done_apm_bios:
```

図 24: Linux/arch/i386/boot/setup.S(534-543)

IN	
AX	5300h
BX	device ID of system BIOS (0000h)
RETURN	CF clear if successful
AH	major version (BCD)
AL	minor version (BCD)
BX	504Dh (”PM”)
CX	flags

表 11: INSTALLATION CHECK

合 INT 15 0x05303 で値が帰ってくる APM のバージョンとフラグを書き込む。

2 回目のチェックの時エラーがでた場合 APM へコネクトできるかもう一度調べる。　コネクトできない時には 32bit APM がサポートされていないとして、フラグエリアの 32bitAPM サポートのフラグをクリアする。

APM を 2 回チェックするのは、APM は電源周りを制御しているため、bios のバグ等で致命的な問題を避けるためだと思われる。ACPI でも同じように、ACPI ブラックリストを実行することにより、バグを持っている BIOS を調べている。

IN	
AX BX	5303h device ID of system BIOS (0000h)
RETURN	CF clear if successful
AX EBX CX DX	real-mode segment base address of protected-mode 32-bit code segment offset of entry point real-mode segment base address of protected-mode 16-bit code segment real-mode segment base address of protected- mode 16-bit data segment

表 12: CONNECT 32-BIT PROTMODE INTERFACE

IN	
AX BX	5304h device ID of system BIOS (0000h)
RETURN	CF clear if successful
AH	error code (CF set on error)

表 13: DISCONNECT INTERFACE

3.3 プロテクトモード

CPU をリアルモードからプロテクトモードにするために、CR0 レジスタの 1bit 目を 1 にする。

```
793   movw    $1, %ax                 # protected mode (PE) bit
794   lmsw    %ax                     # This is it!
795   jmp     flush_instr
```

図 25: Linux/arch/i386/boot/setup.S(793-795)

3.4 圧縮カーネルの展開

圧縮されたカーネルを展開するルーチンは decompress_kernel である。decompress_kernel は misc.c に書かれている。

```
375         makecrc();
376         puts("Uncompressing Linux... ");
377         gunzip();
378         puts("Ok, booting the kernel.\n");
379         if (high_loaded) close_output_buffer_if_we_run_high(mv);
380         return high_loaded;
```

図 26: Linux/arch/i386/boot/compressed/misc.c(375-380)

このルーチンが実行されることによって、画面に
Uncompressing Linux... Ok, booting the kernel
と表示され、この後セットアップルーチンは head.S へジャンプする。

4 APMの設定

4.1 APMとは

PCの電源管理をする規格である。APMを使うことによって、バッテリーを5段階の状態に分けて監視し、アプリケーションソフトが自動的に電源を切ったりすることができる。BIOSの機能として実装される。APMに変わる機能として、OSが直接電源管理できるAPPIが制定されている。ACPIにはAPMに互換性がある。

4.2 読み出し

setup.Sで得たパラメータをsetup.cで読み出す。そしてそのパラメータを使用し、APMの機能を有効にする。

```
179 #define PARAM   ((unsigned char *)empty_zero_page)
180 #define SCREEN_INFO (*(struct screen_info *) (PARAM+0))
181 #define EXT_MEM_K (*(unsigned short *) (PARAM+2))
182 #define ALT_MEM_K (*(unsigned long *) (PARAM+0x1e0))
183 #define E820_MAP_NR (*(char*) (PARAM+E820NR))
184 #define E820_MAP    ((struct e820entry *) (PARAM+E820MAP))
185 #define APM_BIOS_INFO (*(struct apm_bios_info *) (PARAM+0x40))
```

図 27: Linux/arch/i386/boot/setup.c(179-185)

APMのブートパラメータはzero_page上のPRAM+0x40にロードされている。zero_pageとはその名前の通り0で埋め尽くされたページテーブルである。起動時には一時的に、APMなどのブートパラメータを保存している。

4.3 構造体へ編入

```
1078 void __init setup_arch(char **cmdline_p)
1079 {
1080         unsigned long max_low_pfn;
1081
1082 #ifdef CONFIG_VISWS
1083         visws_get_board_type_and_rev();
1084 #endif
1085
1086 #ifndef CONFIG_HIGHIO
1087         blk_nohighio = 1;
1088 #endif
1089
1090         ROOT_DEV = to_kdev_t(ORIG_ROOT_DEV);
1091         drive_info = DRIVE_INFO;
1092         screen_info = SCREEN_INFO;
1093         apm_info.bios = APM_BIOS_INFO;
```

図 28: Linux/arch/i386/boot/setup.c(1078-1093)

setup_archが起動すると、図 28:1093 行目で構造体に入れられる。この構造体の定義は apm_bios.h にある。

BIOS コール INT 15 0x05300、0x5003、0x5004 で呼び出された値が構造体に入る。

```
29 struct apm_bios_info {
30         unsigned short  version;
31         unsigned short  cseg;
32         unsigned long   offset;
33         unsigned short  cseg_16;
34         unsigned short  dseg;
35         unsigned short  flags;
36         unsigned short  cseg_len;
37         unsigned short  cseg_16_len;
38         unsigned short  dseg_len;
39 };

51 struct apm_info {
52         struct apm_bios_info    bios;
53         unsigned short          connection_version;
54         int                     get_power_status_broken;
55         int                     get_power_status_swabinminutes;
56         int                     allow_ints;
57         int                     realmode_power_off;
58         int                     disabled;
59 };
```

図 29: Linux/include/linux/apm_bios.h

4.4 起動

```
1849 static int __init apm_init(void)
1850 {
1851         struct proc_dir_entry *apm_proc;
1852
1853         if (apm_info.bios.version == 0) {
1854                 printk(KERN_INFO "apm: BIOS not found.\n");
1855                 return -ENODEV;
1856         }
                                :
                                :

1922         set_base(gdt[APM_40 >> 3],
1923                  __va((unsigned long)0x40 << 4));
1924         _set_limit((char *)&gdt[APM_40 >> 3], 4095 - (0x40 << 4));
1925
1926         apm_bios_entry.offset = apm_info.bios.offset;
1927         apm_bios_entry.segment = APM_CS;
1928         set_base(gdt[APM_CS >> 3],
1929                  __va((unsigned long)apm_info.bios.cseg << 4));
1930         set_base(gdt[APM_CS_16 >> 3],
1931                  __va((unsigned long)apm_info.bios.cseg_16 << 4));
1932         set_base(gdt[APM_DS >> 3],
1933                  __va((unsigned long)apm_info.bios.dseg << 4));
                                :
                                :
1958 kernel_thread(apm, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);
```

図 30: Linux/arch/i386/kernel/apm.c

apm_init() で APMBIOS に必要な値を与えてプロセスをカーネルスレッドとして動かす。スレッドとは、長時間かかる処理を実行してもプライズマリスレッド等に影響を与えない。また、同時に複数のスレッドと同期をとりながら処理することができる。

5　まとめ

Linux の起動ルーチンは理解できたが、super page が有効になったとき ACPI 等でオフセット値から、直接メモリが参照できないバグの原因は完全には分かっていない。

6　参考文献

Linux のブートプロセスをみる。　白崎　博生
http://lxr.linux.no/blurb.html
http://lkh.linux.or.jp/24kernel/source/html4/boot.html
http://www.delorie.com/djgpp/doc/rbinter/ix/