オペレーティングシステムを学ぶなら、LINUX 0.01を選ぼう

現実のオペレーティングシステムの動き、その実体となったカーネルを理解する事は見かけほど単純ではありません。というのは
例えソースコードを手元に持ってきても、何故なら必要とされる知識が広範囲にわたるため手取り足取りにはいかず、必ず自分で
学ばないといけない領域があるからです。問題はここが初学者を挫折させてしまう大きな壁となっている事でしょう。
有名な幾つかの文献を読んでみましたが、これまでに出版されてきたカーネルの解説書には欠点があると感じました。まず最初に
引用されたソースコードの実体は膨大で、初学者には何処から手をつけてよいか解らなくなってしまう点にあります。といえばさ
何十年の時間と世界中の人々の努力から、新しい機能の追加や削除の積み重ねられた集大成だと実感出来るでしょう。問題なのは
比較的ページ数の多い文献を選んでさえ、引用されるのは何百万行の内何万行という局所的に絞られた点にあります。対象となる
カーネルは最近のバージョンになるため、十年二十年先には何千万行の内何万行程しか引用されていないでしょうね。につづいて
プログラミング手法やプログラム言語で、アセンブリ言語・ハードウェア・Ｃ言語に関する内容は含まれないのです。という事は
本を読むにもソースコードを読む前にも、初学者は最初に上記のプログラミングに長い時間を得なければなりません。無謀な事に
アセンブリ言語とハードウェアは一対で、アセンブリ言語＝ＣＰＵを学ぶという意味合いがあることも知って下さい。そして時に
ハードウェアのデータブックの入手から、プログラミングマニュアルに至るまで多くの情報源が必要になるでしょう。といっても
現在は高級言語を扱うことが主流なため、低級言語のプログラミングに関する和書は年々出版されなくなっています。本も少なく
ウェブ上の情報も断片的であることから、実際に学んでいくには相当の時間と根気が必要であろうことがわかります。したがって
調べる事にばかり気をとられてしまって、これから学びたい方にとって必ずしもいい環境とは言えないのが現状です。
世界中の先人の知恵を和書と組み合わせ、それを如何に吸収して消化するのかにより理解の質には大きな差が出ます。先に挙げた
文献の中で引用された最近のカーネルも、全ては過去のカーネルの連続性であって元を辿ると数万行に収まるのです。最たるのは
x86CPUの下位互換性は維持されており、基本的な原理原則を理解してから現行のカーネルを読み解く事が可能です。少なくとも
0.01-0.02-0.03と次々に読み解いたら、カーネルに対する日本語のドキュメントは凄まじい勢いで増えるでしょう。当たり前に
何百万行に及んだソースコードからでは、根本を理解する為に必ずしも必要ではないノイズが存在する訳ですからね。だからこそ
初学者に適した学び方を実現するのにも、基本的な原理原則の反映された初期のカーネルから学ぶ必要があるのです。といっても
単純な行数だけの問題ではなく実際には、作りながら学ぶＯＳカーネルや３０日で作るＯＳ自作入門は適してません。というのは
数千行というコンパクト化されていても、ＯＳを学ぶのではなく作ることに主眼が置かれたために発展性に欠けます。内容に沿い
読み進める事で一応の動く物は出来ても、メモリ管理やファイルシステムといった所は自前で作る必要があるのです。そうはいえ
ハードウェアに関する記述も簡略化され、独自のＯＳを作るのならば他のＯＳを参考にしなければならないでしょう。そもそもに
ＯＳの役割とはハードウェアの抽象化で、ここを掘り下げなければならない事は自明で絶望的に骨が折れるはずです。だからこそ
これらの本を基にした大多数のブログは、作り終えるまでのプロセスだけに留まり新しい機能の追加が少ないのです。見渡したら
書店に溢れている大抵の解説書を一言で、懐は温かいが敷居は高いか敷居は低いが懐は使い捨て懐炉に大別出来ます。
このような背景もあり大多数の人々なら、オペレーティングシステム設計と実装というＯＳ教科書を勧めるでしょう。というのは
1987年(邦訳は1989)で第三版にも続き、UNIXﾗｲｸなMINIXを教材よりソースコード一式が添付されているからです。これが基で
ＯＳの基本的な理論を実装面と合わせて、現実のオペレーティングシステムがどのように動いているのかを学べます。最たるのは
Linux0.01の生みの親Linus Torvaldsも、MINIXへの技術的アプローチをきっかけにＯ Ｓを書き始めたという点です。こういった
理由により多くの方々はこの本の一読を、オペレーティングシステム初学者の最も無難な学び方と捉えているのです。けれど僕は
とりわけ日本において先人の少ない事や、日本語の資料も少ない事から他の選択肢がなかっただけだと捉えています。だからこそ
新しい学び方を考えて資料の充実してる、Linux0.01から学ぶことこそ実践により最も無難であると理解したのです。

ソースコードを読み進めるまえに

ＯＳが制御しているハードウェアの動き、8042/8251/8253/8255/8259 などを読み解くにはどうすればよいのか。それぞれは
ＧＵＩプログラミングが主流となる今も、身の回りにある大抵のパソコンに搭載されたＯＳに共通した基本技術です。そもそもの
役割はハードウェアの抽象化にあるので、ＯＳであるLinux0.01にも例外はなく上記の制御コードが含まれています。
Linux0.01を初学者が読み解くためには、Intel80x86プロセッサを制御するためのアセンブリ言語の知識が必要です。また同時に
Ｃ言語によるプログラミングの経験含め、熟練とまではいいませんが少なくとも使用出来る必要のある事は確かです。もし仮にも
未経験ならＫ＆Ｒと呼ばれ古典といえる、The C Programming Language(邦題ﾌﾟﾛｸﾞﾗﾐﾝｸﾞ言語C)一読を勧めます。この次には
ハードウェアの一般的な構成を知るため、ﾊﾟｿｺﾝのｺﾓﾝｾﾝｽ徹底図解 ﾊｰﾄﾞｳｪｱ&ｿﾌﾄｳｪｱの動作のしくみ一読を勧めます。その中心の
プロセッサとＯＳとの関わりを知るため、WindowsOS内部ｱｰｷﾃｸﾁｬのすべて,CPUの謎,CPUのきもち一読を勧めます。少なからず
アセンブリ言語でプロセッサを扱うため、はじめて読む{8086→80486}と進み x86アセンブラ入門一読を勧めます。併用として
例題プログラムを読み解くための仕様書、IA-32 ｲﾝﾃﾙ(R) ｱｰｷﾃｸﾁｬ・ｿﾌﾄｳｪｱ・ﾃﾞﾍﾞﾛｯﾊﾟｰｽﾞ・ﾏﾆｭｱﾙ：上と中と下巻。その他にも
以下に挙げる本があれば効率よく学べる、INTEL 80386 Programmer's Reference Manual (邦訳此処)を読むこと。LINUX0.01
であればインライン構文も出てくるので、Using Intel Assembly With gcc この資料とウェブ上の情報が役立ちます。そして更に
幾つかのﾊｰﾄﾞｳｪｱのﾌﾟﾛｸﾞﾗﾐﾝｸﾞﾏﾆｭｱﾙは、MS-DOS時代に豊富で8042/8251/8253/...という低級な内容に溢れてる。ただ多くの
和書は既に絶版となってしまったために、Amazonﾏｰｹｯﾄﾌﾟﾚｲｽで駄目なら国立.大学.公共の図書館から借りて下さい。最後として
ウェブ上で公開された講義資料も役立ち、09h15.com文書の検索エンジンや大学のOCWを利用するとよいでしょう。
そしてLinus Torvalds自身が参考とした、オペレーティングシステム設計と実装は学んでいく時の一番の参考書です。というのは
0.01はMINIX環境下でクロス開発されて、ソースコード木からファイルシステムに至るまでに多くは似通っています。このことは
オペレーティングシステムの理論面含め、ソースコード一式の構造体や変数のコメントを当て嵌めて参考に出来ます。もう一つは
古くとも様々な箇所で痕跡の残っている、The Design of the UNIX Operating System(邦題UNIXカーネルの設計)。この他では
初期のカーネルを解析した二つの文献で、The Linux of the 0.01 Commentaryという 0.01を解析した文書がある。題名の通り
ソースコードの書き加えられた注釈含め、80386プロセッサの構造を本家と違い最低限必要な分だけに纏めています。全体を眺め
百ページと主要な文献に比べ少ないけど、一個のソースファイルに対する概要と注釈があるので読みやすいでしょう。もう一つは
Linux0.11とバージョンこそ違うものの、Linux 内核完全注釈という中国の文献で徹底的に解析されている文書です。 題名の通り
ソースコードの注釈は膨大尚且つ詳細で、ﾏｲｸﾛﾌﾟﾛｾｯｻの構造や制御からアセンブリ まで徹底的に纏められています。 全体を眺め
九百ページと辞書にも等しくなりますが、初版のものであれば五百ページ程に収まるので追って読み進めていけます。凄いことは
著者の公開しているサイトoldlinux.org、歴史的な資料から古代のﾃﾞｨｽﾄﾘﾋﾞｭｰｼｮﾝやﾊﾞｲﾅﾘ等を収集している事です。 最後として
Linux の歴史的な生い立ちを知るために、ソースコードの反逆-Linux開発の軌跡とｵｰﾌﾟﾝｿｰｽ革命 が参考となります。その他にも
Linus Torvaldsの自伝として取り上げた、それがぼくには楽しかったからでは彼の立場から考え方がよくわかります。
さて多くの参考書を最初に挙げた理由は、読み解いていく時間の短縮に役立ちたいという第一の趣旨があるからです。こういう時
ある人々はソースコードがあれば充分だ、等と言いますが繊細な細部についての概要を使うには不十分だといえます。十分ならば
言い換えてカーネルを学ぶ初歩の学生は、8086を知らずにIA-32ｱｰｷﾃｸﾁｬへ通じる全ての意図を引き出すことができ、その他での
補助テキストは必要ないことになります、多分世の中にはそのくらい簡単に出来るよと主張する天才もいるでしょう。といっても
圧倒的な多数は一般に様々な情報源から、より多くの洞察を得る必要があって天才にも即座に理解できない領域です。よって時に
詳細に理解していくのに別の文脈として、古いシステムから受け継いだ遺産からも論じる必要があると感じています。カーネルが
過去の連続性であると最初に言ったのは、先人との対話によってのみ得られる理解があることを伝えたかったのです。

ビルドしてエミュレートさせよう

Adbel Benamroucheという方によって、Linux 0.01はGCC-4.X,BASH-3.2上に移植した事KernelTrapで発表され、そのさいに

ソースコードやドキュメントについては、その方の個人サイトで公開され www.kernel.org でも共有されています。ビルドしてエミュレータよりデバッグもできるうえ、現行のカーネルに至るまで移植もかねて ﾊｰﾄﾞに詳しくなるいい機会です。

- Index Of
  - patch and source : linux-0.01-rm-3.x/
  - floppy and qemu hdd : Image/
  - linux 0.01 document : doc/

ビルド環境はBackTrack 4R2なのですが、最低限必要なパッケージとして bin86, gcc-4.1, qemuをインストール。もし違ったディストリビューションで行う場合には、userland_src/に含まれるパッケージをインストールする必要があります。

- ﾌﾞｰﾄｺｰﾄﾞ用の8086ｱｾﾝﾌﾞﾗ及ﾘﾝｶ
  ```
  aptitude install bin86
  ```
- rm-3.5用のgcc-4.1
  ```
  aptitude install gcc-4.1
  ln -sf /usr/bin/gcc-4.1 /usr/bin/gcc
  ```
- ｴﾐｭﾚｰﾄ用のqemu
  ```
  aptitude install qemu
  ```

BackTrack 4R2標準のgcc-4.3を使うと、ｺﾝﾊﾟｲﾙ時にｲﾝﾗｲﾝｱｾﾝﾌﾞﾘのｴﾗｰが発生するのでv4.1にﾀﾞｳﾝｸﾞﾚｰﾄﾞします。幸いな事にPlan9日記の oracchaという方の手より、Linux 0.01をUbuntu 10.10の QEMUで動かすでﾊﾟｯﾁが公開されました。

- gcc-4.1は導入せずﾊﾟｯﾁを当てて使う
  ```
  patch -p0 < linux-0.01-rm-3.5-rt20110112a.patch
  ```

準備は整ったのでエミュレータの設定は、リリースノート及びLinus Torvalds自身のマシンの仕様を基に構成します。詳細として前回の記事で紹介した書籍を見てみれば、0.01はCPU:386-DX33,RAM:8MB,HD:40MB で387ｺﾌﾟﾛｾｯｻ無しでした。

```
cd linux-0.01-rm-3.5
make
qemu -m 8 -hdb hd_oldlinux.img -fda Image -boot a
```

EMUのエミュレーションドキュメント、GUIフロントエンドのAQemuを使う事でより簡単にエミュレートできます。機会あればより多くのＯＳに触れておくべきであり、爆速エミュレーター QEMUで行こう!から幾つかの代表的なＯＳを試せます。このほかにＯＳの紹介にて群を抜いているサイトで、オープンギャラリー: OS博物館 を参考にエミュレートする事もお勧めです。そうはいえ例えばウィンドウズ旧版を入手するには、MSDNサブスクリプションに登録するか地道に探していく他ないでしょう。 個人的には現在のＯＳの脆弱性を突いてのハックや、それを実現する攻撃ソフトやマルウェアは歴史を辿って学ぶべきだと思う。

ブートローダを読み明かすまえに

ブートストラップローダを読み解くには、主記憶にﾌﾞｰﾄｾｸﾀが読み込まれた直後のﾒﾓﾘﾏｯﾌを意識しなければならない。高級言語によるプログラミングで意識せず済むのは、コンパイラやOSがこういった低レベルな事象より隠蔽してくれるからです。という事でOSを読み込むための ブートローダでは、予めBIOSやIOアドレス空間を知らないと処理を意図を掴む事ができません。
コンピュータの電源を起動させた直後は、昔のOSのブートできる互換性あるモードより煩雑な過程で切り替わります。この部分は参考としたウェブサイトや文献をあげて、沿って説明していく事で本来の役割ではない 雑多な所を省こうと思います。という事でBIOSのPOSTを経て制御が移されるとき、How Computer Boot Up,Motherboard Chipsets and the MemoryMap。ただこれは英語の記事なので日本語サイトでいうと、ﾌﾞｰﾄｽﾄﾗｯﾌﾟ - ｺﾝﾋﾟｭｰﾀの起動手順やLinux2.4 ｶｰﾈﾙ開始まで等もあります。まず最初にブートローダの動き方を把握するために、Minimal Intel Architecture Boot Loader より読み進めるといいでしょう。その次には実際にソースコードを読み解く段階から、LinuxのﾌﾞｰﾄﾌﾟﾛｾｽをみるとLinux JF ｶｰﾈﾙ関連を何度も読み返すはずです。より細かなハードよりの処理はWiki,Forumなどで、大御所よりosdev{JP:w,f}-{EN:w,f}やoldlinux{ZH:f}を活用しましょう。歴史を追い現在へ移り変わるブートローダにおいて、なぜx86ではMBRが"0x7C00"にロードされるのか？を読むといいでしょう。
0.01 Boot Strap Loader ファイルとは、boot/boot.sであり前回のbin86ﾊﾟｯｹｰｼのas86&ld86により生成されます。この両方は8086 Assembler and Linker であって、昔のOSに歴史を引きずる 8086互換動作ﾓｰﾄﾞ用の機械語を扱ってくれます。全体としてﾌﾞｰﾄｽﾄﾗｯﾌﾟﾛｰﾀﾞがﾌｰﾄｾｸﾀにになるまで、make->boot.s->as86(boot.o)->ld86(boot)->a.outﾍｯﾀﾞ除外を経ます。まず最初に七誌の開発日記のa.outﾀﾞﾝﾌﾟﾂｰﾙを使い、bootﾌｧｲﾙの内部構成をざっと見た後にソースコードの方を見てみましょう。

```
00000000: a_magic     01 03    magic number
                      a.out executable
00000002: a_flags     10       flags, see below
                      A_EXEC
00000003: a_cpu       04       cpu id
                      A_I8086: A_BLR=0, A_WLR=0
00000004: a_hdrlen    20       length of header
00000005: a_unused    00       reserved for future use
00000006: a_version   0000     version stamp (not used at present)
00000008: a_text      000001c4 size of text segement in bytes
0000000c: a_data      00000000 size of data segment in bytes
00000010: a_bss       00000000 size of bss segment in bytes
00000014: a_entry     00000000 entry point
00000018: a_total     000081c4 total memory allocated
0000001c: a_syms      00000000 size of symbol table
A_TEXTPOS: 00000020 <- a_hdrlen
A_DATAPOS: 000001e4 <- A_TEXTPOS + a_text
```

text,data,bss が共有されていることと、a_magic=0x01,x03(minix a.out header)で0x20(32byte)長と解ります。 このあとにtools/build.c がヘッダを取り除きして、boot signature {0xAA,0x55} 有効にしてブートセクタの出来上がりです。 さてここでリンカに興味を持たれた方が いるなら、Linkers & LoaderesやリンカとローダとC++実践開発テクニックや役立つはずです。

ブロック毎に抽象的に読み進める

ソースコードの読み書きを行う環境には、UNIX以前に遡るEmacsやviと違いKDEのkateというテキストエディタです。実はあまり

最初に上げた二大エディタは使用せずに、もっぱらEclipseやNetbeansというIDE(統合開発環境)を使う事が多いです。その理由はカスタマイズに手間がかかることに飽き、熱心な信奉者になるにはソースコードを基に理解する必要があるからです。両方とも英語圏で生まれたものですから、海外のハッカーの環境やその情報等はどうしても遅れをとってしまいます。それだけの収集に手間と時間をかけるのではなくて、ソースコードを読んだ上でカスタマイズする方に重きを置く性格なのです。このことは歴史のあるソフトウェア程に感じていて、僕はカーネルでいえばアセンブラやコンパイラにも興味をわかせています。本題に移り0.01Boot Strap Loaderを読んでいくと、レガシーインターフェイスが何処で制御されているかまず気になりました。で調べるとMotherboardSuperI/O に組み込まれて、チップセットのサウスブリッジにLPCLowCountPinBusで繋がっています。このようにハードウェア寄りな所は多い訳ですから、実際のプログラミングは機能毎に仕様や特性を調べる必要があるでしょう。といっても必要になる時はリファレンスがあるので、関数や構造体を対に主要な要点を記事に纏めていくアプローチをとります。なのでまずハードウェア寄りなコードである場合は、後々プログラミングする為に参考文献やサイトの紹介をしておきましょう。0.01Boot Strap Loaderboot/boot.sは、コンピュータの電源が起動すればBIOSにより自動でx7c00へ読み込まれる。いうもなくノイマン型アーキテクチャの特徴の内で、ＯＳが電源投入直後に主記憶に読み込まれていない矛盾の解決役割を担う。続きましてプログラム中のエントリポイントとなる、entry start: から全体を纏まりある処理毎に纏めてみると以下の様になる。

1. 自身を0x07c0(31KB)から0x9000(576KB)へと移動させる
2. "Loading system" という文字列を改行付きで出力する
3. ｼｽﾃﾑを0x1000(64KB)へFDから読み込ませる
4. ｼｽﾃﾑを0x0000(00KB)へ移動させる
5. 保護モードへ移行するため仮のGDTとIDTを登録させる
6. 1MB以上のﾒﾓﾘを使用のためA20 ｱﾄﾞﾚｽﾗｲﾝを有効にする
   基本輸入/輸出系統(BIOS)研究.The PS/2 Keyboard Interface.ｷｰﾎﾞｰﾄﾞｺﾝﾄﾛｰﾗ KBC.A20ﾗｲﾝ
7. 例外とﾊｰﾄﾞｳｪｱ 割り込みのﾍﾞｸﾀ重複のため再設定する
   割り込みﾌﾟﾛｸﾞﾗﾐﾝｸﾞ技法.8259AのIOﾎﾟｰﾄ番号が謎だった件
8. CPUを保護モードへ移行してGDT[0x08]へジャンプする
   GDT + 0x08 = System.map:[head-main-kernel-mm-fs-lib]

```
.text           0x0000000000000000     0xf3e5 /* [section name] [base address] [size] */
 *(.text .stub .text.* .gnu.linkonce.t.*)
 .text          0x0000000000000000     0x5078 boot/head.o /* boot/head.o = GDT+0x08   */
                0x0000000000004878                gdt
                0x0000000000000000                startup_32
                0x0000000000000000                pg_dir
                0x0000000000004078                idt
 *fill*         0x0000000000005078        0x8 90909090
```

システムのスタートアップコード

前回のブートストラップローダの処理は、NMIを除く全割込みをCPUと8259で無効にして空IDTと仮GDTを設定した。基本的にはIDTとGDTは保護モードの移行に必須で、即ちリアルモート時ブートセクタ512bに収まる最小限の構成になっている。という事でプロテクトモードに切り替わった段階に、システム・セットアップ・コードで本設定してから ページングを開始する。

1. ds = es = fs = gs = gdt[0x10](data segment)へと初期化する
2. esp = stack_start(ss:esp)へﾌﾟﾛﾃｸﾄﾓｰﾄﾞ用に切り替える
3. ｶｰﾈﾙ:GDT(8MB,CS,DS)とIDT(ignore_int)へ再設定する　(call - use stack)
4. esp = stack_start(ss:esp)を再度初期化してあげる　(interrupt - ignore_int)
5. CPU内部のｾｸﾞﾒﾝﾄﾃﾞｨｽｸﾘﾌﾟﾀｷｬｯｼｭを更新してあげる
6. A20が確実に有効であるかを確かめる
7. 80387確認で無ければエミュレート
8. after_page_tables() - push $main(argc, *argv[], *envp)
9. setup_paging() - pg_dir,pg0:4MB,pg1:4MBで有効化してret

このコードはGNU Asasembler(GAS)に、code32ﾃﾞｨﾚｸﾃｨﾌﾞ(ｱｾﾝﾌﾞﾗに対する疑似命令)で32bitの機械語で出力する。そのうえでカーネルが利用するGDTを本設定させて、IDTの全ｴﾝﾄﾘにignore_intUnknown interruptを標準ﾊﾝﾄﾞﾗとして初期化。この時点で前回のｾｸﾞﾒﾝﾄﾃﾞｨｽｸﾘﾌﾟﾀはCPUにｷｬｯｼｭ、ﾍﾞｰｽｱﾄﾞﾚｽとﾘﾐｯﾄを主記憶から読み取る際の余分なﾊｽｻｲｸﾙを回避している。という事は最新のカーネルが利用するGDTに修正後、ｾｸﾞﾒﾝﾄｾﾚｸﾀを再度読み込みわざと新しいｾｸﾞﾒﾝﾄﾃﾞｨｽｸﾘﾌﾟﾀをｷｬｯｼｭさせる。終えてから1MB協会のｱﾄﾞﾚｽﾊﾞｽ A20ﾃﾞｺｰﾄﾞを確認、80387のｴﾐｭﾚｰﾄ及びmain()の引数をｽﾀｯｸに積んでﾍﾟｰｼﾞﾝｸﾞ後RETしてる。

```
ページディレクトリ:ページテーブル = 4096b(4K) 1024 entries(4b)
  pg_dir[0] = pg0
  pg_dir[1] = pg1
  pg0 = 4MB : 0x07 = 00000111(r/w user,p)
  pg1 = 4MB : 0x07 = 00000111(r/w user,p)
  ↓
  CR3 = 0x0000(pg_dir)
  CR0 = PG bit on
  ↓
  ret = (eip = main)(esp = esp + count)
```

インテルの日本語技術資料をみることで、ここまで多くの処理の位置はだいたい掴むことが出来るだろうとかんじる。といっても本家は分厚くお世辞にも見やすくはない、なので各種のレジスタやメモリ保護にKBCまでHazy Moonをお勧めしよう。

include/bits.h ワイド型の定義

Wide Character Header(wchar.h)は、1995年にプログラム言語Ｃが国際化に対応するために追加されたものです。という訳でオリジナルには存在していないですから、ワイド型の範囲を定義しただけで関数はなかったので注釈は省いています。

include/asm 低級な補助関数群

カーネルが利用するための低級な処理は、インラインアセンブリを通した補助的な関数群で此所に定義されています。という事で

インラインアセンブリの処理を追うには、時間がかかるので関数間の全体像を把握してから読んでゆこうと思います。

include/asm/io.h

入出力ポートのアクセスを単純化する為、アセンブリの入出力命令は補助的な関数としてインラインで実装している。主な要点は関数名のサフィックス(b = byte)であり、(p=pause)有りは CPUとI/Oコントローラ間のギャップを埋める為にある。実装の方はわざと遅く動作するように作られており、入出力命令の直後に明示的な空ループが挿入されているだけで単純である。

include/asm/memory.h

メモリブロックの移動を行う関数として、カーネル・ユーザ空間毎に独立したセグメントセレクタを予想しています。

include/asm/segment.h

カーネル・ユーザ空間を繋げる架け橋で、ds=es=x10(kernel space):fs=x17(user space)でデータ交換しています。というのは特権レベルの異なる空間の行き交いには、Ｃ言語で実装できないからこそインラインアセンブリを用いたのでしょう。

include/asm/system.h

システムを直接的に制御する際の関数で、INIT_TASKのユーザモード移行やGDT[TSS/LDT]やIDTの操作を行います。というのはカーネルに対する一般的な理解があれば、分厚いインテルのマニュアルを読むにも何処を読めばいいか解るでしょう。

include/sys システムの参照用

システムの全体に為すヘッダファイルで、全体で関係のあるものや全体がなければ成り立たない関数などがあります。

include/sys/stat.h

Status Header (stat.h)という名の通り、開いているファイルの状態を取得する関数や構造体等が定義されています。

include/sys/times.h

Times Header(times.h)という名の通り、現在のプロセス時間を取得するための関数や構造体等が定義されています。

include/sys/types.h

Types Header(type.h) という名の通り、プラットフォームに依存する型の移植性や可読性などを向上させています。

include/sys/utsname.h

UNIX Timesharing System Nameの略、システムを識別する為の名前の定義でsystemcall:(uname)で参照されます。

include/sys/wait.h

Wait Header (wait.h)と呼ばれるもので、呼び出し元の子プロセスの状態変化を監視・取得する為に必要なものです。

include/linux カーネルの実装用

カーネルの根幹を為すヘッダファイルで、include/階層のユーザ層レベル(sys_call)と違ってカーネル層が扱います。予想するにシステムコールより先に実装されていて、ＯＳを自作するなら一番参考にしなければならないヘッダになるでしょう。

include/linux/config.h

Configure Header(config.h)にはまず、カーネルの始動環境 {HD_TYPE, BUFFER_END}を定数を定義しています。具体的にはハードウェアの構成を記すためにあって、バッファキャッシュを搭載メモリ容量に合わせて調整をしたりしています。重要なのはルートファイルシステムの存在している、デバイス番号 {0x301=/dev/hd1}でﾄﾞﾗｲﾌﾞ:ﾊﾟｰﾃｨｼｮﾝの指定しています。

include/linux/fs.h

File System Header(fs.h)はそのままに、MINIX File System と互換性があり必要な定数は一ヶ所に纏まっています。具体的にはsupoer_blockやd_inode を見てみると、パーティション は64MBまでファイル名は14文字という単純なものでした。問題なのはカーネルの文献に載せられている注釈は、マクロや構造体の注釈が実装の結びつきに弱い為 意図を掴みにくい事です。このようにカーネルが動くために必要不可欠なので、データ構造が設計されている領域として注釈を省略すべきではありません。という事で英語・日本・中国のサイトを巡りながら、ﾆｭｰﾖｰｸ州立大学ｽﾄｰﾆﾌﾞﾙｯｸ校:Very Simple Real File Systemを発見した。これ自体はi-node->i_nlinksの役割を知る為で次に、法政大学情報科学部:ﾌｧｲﾙｼｽﾃﾑとi-node が凄く明解で解りやすかったです。最期としてNR_TASK 307 に関する情報が全くなく、0x307-/dev/hd7のデバイス番号から素数として選んだのか悩んでました。これだけは今も解っていないので知る方がいたなら、ツイッターやメールから紹介された記事等でも教えて頂けると助かります。このようにヘッダファイルを先に調べるというのは、演繹的であるが故に適切な用語を組み合わせ適切な意図にする必要がある。こうやって注釈が設計全体を網羅されていくなかで、知りたい処理が実装上のどの関数や変数に着目すれば良か一目瞭然となる。という事でヘッダファイル全体に注釈を付ける場合、的を得た注釈でないと実装を読み解く上で面倒くさい読み方になるだろう。

include/linux/hdreg.h

Hard Disk Register Header(hdreg.h)は、ハードディスク(IDE)の為のドライバで各種のレジスタを定義しています。具体的には1994年:ANSIによりIDEを規格化した文書、Guide to ATA/ATAPI documentationよりATA1のPDFを見てください。ともなって#define HD_CMD 0x3f6の定義を読む時、ﾊﾟｿｺﾝのﾚｶﾞｼｨI/O活用大全とirqs-dma-ioports.txtが読みやすいでしょう。

iniclude/linux/head.h

ｼｽﾃﾑ・ｾｯﾄｱｯﾌﾟ・ｺｰﾄﾞ(head.s) 定義から、pg_dir, gdt, ldtの外部シンボルをＣ言語側より扱えるようにしています。具体的にはそれぞれに extern 修飾子を用いることで、外部で定義された (head.s - symbol)をリンカがリンクしてくれています。

include/linux/kernel.h

カーネル内部で頻繁に呼び出される関数、例えばprintk()はカーネルデバッグの一番お手軽なツールでお馴染みです。どれくらいお手軽かはデバッグのテクニックを見て、口でクソたれる前と後に"printk"とつけろ！わかったかウジ虫！のような。例えばそうLinus Torvaldsがどうデバッグしたのか、1993年09/11和訳されたWriting an OSを見ると当時の状況を垣間見れます。

include/linux/mm.h

Memory Header (mm.h) というのは、、ページングにおけるメモリ管理(head.s - pg_dir: pg0-pg1)を処理する。具体的にはBUFFER_ENDを基に物理メモリ容量を、6MB以下ならx100000-&endまで8MBなら0x200000-&endをマップする。このように１ページの取得:挿入:解放の関数があり、640Kb - 1MBのI/Oアドレスマップ空間はメモリとして使用されていません。

include/linux/sched.h

Schedler Header(sched.h)は名の通り、マルチタスクを可能にするプロセス・スケジューリングに必要なものです。具体的にはカーネルに静的に定義した INIT_TASK、インテルの日本語技術資料(TSS)とコンテキストスイッチ(switch_to)です。例えばそうt

ss_struct->sig_restorerは予想ですが、シグナルハンドラから復帰した時に呼び出される{SA_RESTORER}と思う。

include/linux/sys.h

System Calls Header(sys.h)というのは、システムコール(int 0x80, eax=n)からジャンプテーブル[n]を定義します。具体的にはユーザプログラムから呼び出された際に、ユーザ側:unistd.h-カーネル側:sys.hに名前解決を行う架け橋になります。というのはシステムコール一覧の注釈を付ける前に、extern int sys_n(): をモジュール別に再度分類する必要があるでしょう。

include/linux/tty.h

Teletypewriter Header(tty.h)といって、端末の入出力制御を行うためのキューと操作用のマクロを定義しています。という事で読み解くにはCQ出版:1997/以前の文献、ﾊﾟｿｺﾝ通信ﾌﾟﾛｸﾞﾗﾑのすべてや実用RS-232C通信ﾌﾟﾛｸﾞﾗﾑの作成法が必須。

include/ 共通かつ標準な処理用

ソースファイルを読み解いていく前には、過去の文献にある豊富な遺産(コメント)をヘッダファイルに書き足します。まず最初にオペレーティングシステム設計と実装を、巻末にある MINIX 3 Source Code Listの注釈を Linux0.01に移植します。というのは処理に対するインデックスを作ることで、多量の文献＋実際の処理を横断的に読み解いて網羅していくためなのです。

include/const.h

Constant Header(const.h)が含むのは、i-node[mode]に保持するファイル種別とアクセス権限に使われる定数です。具体的には[4:type-12:permission]と区画分して、4:type - 3:stickyを定義して残りはinclude/sys/stat.hで利用されています。先頭にあるBUFFER_END 0x200000 については、バッファキャッシュをBUFFER_END:x200000境界まで展開する終端値です。

include/ctype.h

Character Type Header(ctype.h)には、アスキコードに限り文字種別の判定と変換する関数群が定義されています。具体的にはアスキ文字のビットパターン処理なので、単純なマスクと演算だけでＣ標準ライブラリの中でよく使われるものです。より詳しく学ぶには文字コードというものについて、テキストファイルとは何かや文字コード研究を読まれるといいでしょう。

include/dirent.h

Directory Entry Header(dirent.h)には、ディレクトリ属性を持つファイルのデータブロックを処理する構造体です。具体的にはディレクトリは特別な種類のファイルで、ファイル名とi-nodeを一対(組み)にしたエントリが敷き詰められています。このようにDIR構造体:ディレクトリストリームから、ディレクトリファイルからレコードを一個毎に走査する事が出来るのです。

include/elf.h

Executable and Linking Format:elf.h、実行可能プログラムや共有ライブラリのバイナリファイル形式になります。具体的には移植版のみに存在するヘッダファイルで、オリジナル:a.out(assembler output header)MINIXと互換性があります。より詳しく学ぶにはリンカというものについてまず、Linkers & Loaders やﾘﾝｶ・ﾛｰﾀﾞ実践開発ﾃｸﾆｯｸを読まれるといいでしょう。

include/errno.h

関数は呼出し元にエラーを通知しますが、戻り値は何らかのエラーの発生を示すだけで原因を指す情報は持ちません。という事で原因はグローバル変数(error no)で共有、関数がエラーを意味する値を返した場合のみ参照する事が可能になります。何故ならばint errno;　はエラーが発生しなくても、値が初期化される事は無いしその為の関数が存在する訳でもないからです。

include/fcntl.h

File Control Header (fcntl.h)において、開いたファイルに対する操作種別の定義で主にopen.c,fcntl.cが使います。具体的にはロックに関する処理は未実装になるため、もしロックコマンドを指定された場合は全て-1が返される処理になります。難しいのはコマンドの複製や取得と設定という所で、多くの文献で用語や意味合いが違うので実装を見て納得をしてみましょう。

include/signal.h

外部機器からの通知は狭意味で割り込み、もしくは外部割り込み・ハードウェア割り込み等と混在して呼ばれてます。二つ目にはプログラムがＯＳの機能を呼び出す時は、トラップ又はソフトウェア割り込みの機構でこれがシグナルのモデルです。三つ目にはゼロ除算やメモリアクセスエラーの時は、プログラムが先に進めない場合に起こる通知は狭い意味で例外と呼びます。

include/stdarg.h

Standard Arugment Header(stdarg.h)、実引数の個数や型を呼び出しごとに変えられる引数を扱うことが出来ます。具体的には定義されたマクロを用いることによって、printf(const char *fmt, ...)固定引数;fmtと可変長引数:..を扱っています。少なくとも最低限一個の可変長引数を持つ必要性は、檜山正幸のキマイラ飼育記を見ると歴史の流れを感じ取ることができます。

include/stddef.h

Standard Define Header(stddef.h)には、処理系に依存する型とマクロが標準ライブラリ内で共通に使われています。そう例えばポインタ同士の減算結果やsizeof()には、32 or 16 bit OS により計算結果が異なるので移植性の観点でも必要です。

include/stdint.h

Standard Integer Header (stdint.h)は、移植性を保ったままＣＰＵの最も効率良い整数型(ワード)を指定出来ます。というのはソースコードを読み解いていくときには、全く意識することのない定義の羅列でコード行数も長いので割合いします。

include/string.h

String Header (string.h)は説明無しに、文字列を操作する為の関数群を定義しているものとしてお馴染のものです。といってもインラインアセンブリによる実装なので、IBM developerWorks:Linuxにおけるx86ｲﾝﾗｲﾝｱｾﾝﾌﾞﾗ を参考にしました。少なくともアセンブラとリンカを先に学んでみたく、現時点はGCCでｲﾝﾗｲﾝｱｾﾝﾌﾞﾘを使用する方法と留意点等forx86で充分です。

include/termios.h

Terminal I/O Interfaces(termios.h)は、端末入出力操作はUNIXの中で無駄に複雑でマニュアルの頁数も多く占める。歴史を辿り1970年代にてRitchieとK.Thompsonの、The UNIX Time-Sharing Systemという論文から現在へ発展を垣間見れる。その当時にPDP-11に20の通信回線 (ﾃﾞｰﾀｾｯﾄ)と、12の直帰回線(端末)と同期通信回線(ファイル転送)が接続されていました。

include/time.h

カーネルの提供している時刻サービスは、協定協定時[UTC]:1970/1/1/00:00:00から現在まで経過秒数となります。具体的には構造体のint sec(59~61)は閏秒を考慮し、CLOCKS_PER_SEC=一秒間の合計ティック:timer_interruptとなります。少なくともヘッダファイルをつらつら読むことより、2004年に話題となったUNIX TIME 2038年問題等を調べるといいでしょう。

include/unistd.h

UNIX Standard Header:unistd.hと言い、ユーザ層からカーネル層のサービスを呼び出す際のインターフェイスです。具体的には

一連の数値が実装順であるのかないのか、MINIXと違いUNIX Version 7のシステムコールの連番とほぼ一緒なのです。少なくともインデックスを割り振っただけですから、わざわざ注釈を付けることはしないでそのまま読んでいくことにしました。

include/utime.h

UNIX Timestamp Header(utime.h)は、ファイルの参照及び修正時刻を変更するための関数と構造体を利用します。具体的には注釈の通りMINIX File System にはなく、d_inode(disk)にないのでm_inode(memory)側に保存されるだけになります。

Commentary on Linux 0.01

オープンソースソフトウェアを読む事で、多種多様な文献・サイトを横断的に網羅していく過程が生み出されている。これこそがプログラミングによるハックが本質的に、脳内の知識に対する連想的インデキシングでソースコードだと感じている。だからこそ熟練のハッカー達が言い放っている一言、ドキュメントはソースコードだとは体系的で広範囲な知識に裏付けられる。
カーネルの設計に体系的理解を得て上で、データ構造に注釈を入れていけば実装側の関数が何をするものか大体解る。ある程度にカーネルの実装に系統的理解を得た上で、関数による名前と引数をデータ構造と対に読んでいく事で明瞭に把握する。この時点で設計＝ヘッダファイルに注釈を付け終え、実装＝ソースコードに注釈を付けても乱雑な複雑を生み出しただけだった。例えるなら四種類の文字によるゲシュタルト崩壊と、カーネルを読み解いていく過程で言わなくても解る所は読まなくてもよい。という事で最初は一個毎にソースファイルを解析し、呼び出される関数と引数だけを注釈に置き換えていくフォーマットを作る。詰まる所はモノリシックなカーネルを読み解く場合、大局から局所へ即ち設計:header)から実装:sourceと進む事が適している。この理由でカーネルにおける解説書に足りないのは、著者がどのような順番で読み解いたかの過程を示していく事と感じられる。少なくとも物事をどう解析して発見に辿り着いたか、著者の視点を辿っていく学び型こそが悪書を増やさない一つの方法となる。だからこそ僕自身がこうやって記事を書いている時、読み解いている途中でもカーネルを元に感じたことは記事にしているんだ。しかしまぁハウツー的な文書ばかり書くというのは、断片的な知識ばかり満たされて道具の製作者と張り合えていない気がする。そう例えば貴方は道具の使い方ばかり知っているが、それを使って登山したり旅行したりした事がないなら面白みがないという。実際の所は車輪の再発明や再実装は両方とも必要で、道具自体を再発明する事で知恵を再実装する事で歴史を紐解いているのだ。
オペレーティングシステム設計と実装は、ＯＳの実装を読み解く過程であまり良い注釈ではないことに気付かされた。多くの所に最低限で適切でない曖昧な用語がまじり、ソースコードリストを参照したところで多くの洞察は得られていなかった。この理由で実際に注釈を付ける際には主要な文献に、詳解LINUXｶｰﾈﾙ,詳解LINUXﾌﾟﾛｸﾞﾗﾐﾝｸﾞ,例解UNIXﾌﾟﾛｸﾞﾗﾐﾝｸﾞ教室を引用。とりわけてＣ言語の標準ライブラリと言える所では、日本工業規格:JIS X 3010(2003版)と新 ANSI C言語辞典で統一している。可能ならばソースコードに出てくる全用語に対して、コンピュータ英和和英辞典と英和コンピュータ用語辞典を適用したかった。これと別にシステムコール一覧といった纏まりには、25+ USEFUL LINUX AND UNIX CHEAT SHEETS等を参考にすると良い。以上に伴い曖昧なデータ構造を明瞭にしていく中で、カーネルの設計に関する文献の上に関数とデータ構造の結びつきがわかる。この過程でＣソースファイルはシンプルにしておき、詳細な文書はヘッダファイルと共に纏めておくことがベストだと感じます。そのためにオペレーティングシステムの設計として、岩波情報科学辞典と情報処理ハンドブックが纏める際必要になるでしょう。大切な事は読み解くという表現より紐解いていく中、多くの知恵や哲学を得たと同時にコミュニティでの対話の必要性を感じた。多くの所でカーネルは独りよがりには勉強出来ない、多くの文献を網羅して対話を重ね歴史を振り返る事を余儀なくされるんだ。という事で僕がどう読み解いたかの順番については、文献→マニュアル→ブログ→歴史的な所はコミュニティを経ているのです。少なくとも交流を深めておく事はとても大事だけど、素晴らしい学歴や情熱を持った方と最初から触れ合うのはあまり好まない。詰まる所は僕のように何かに取り組もうとする前に、趣味として学んでいく者にとっては周りというものが重圧になるでしょう。
視野の狭い男に見えるかもしれないけど、湧き上がる知恵と情熱を維持する為には触れ合わない事も必要になるんだ。このことは科学的な議論と似ているのかもしれない、ＳＦ小説:青い星まで飛んでいけ に良い文章があったので紹介しておこう。

> 科学的な議論においては、古い説と衝突することになる新しい説を持ち出す方に、説明責任がある。なぜならば古い説は、長い間多くの実験に耐えて生き残ってきたからこそ、今でも生き残っているのだ。古いものとぶつかりたければ、同じぐらい厳しい検証に耐えなければならない。そして、それは新しいものを持ち込んだ側の義務となる。義務というよりも権利ととらえるべきなのだ。科学のフィールドでは、どんな新人にも、新説を主張し検証する権利がある。十分な検証がなされたのなら、必ずそれは本流に受け入れられる。---検証を相手方に押し付けて新説を振りかざすのは、セールスマンが新製品のセールスを客にやらせるようなものだ。そんなことをやる客はいないし、そんなもったいないことはない。

LINUX 0.01から0.00のフェイクバージョンへ

ブログを初めて私が最初に主張したことはオペレーティングシステムを学ぶ初学者にとって理想的な選択の一つに LINUX 0.01があることで、巷で噂のOSの自作に関していえば本を読み切ってからではどのみちUNIX系のカーネルを参考にしなければ実装すらできない者達で溢れるだろうということだ。基本的な原理原則は既にそこにあるのに普段は再実装を謳いながらこのような時だけ再発明を強要するのはよろしくない。ウェブ上にある海外の資料と書籍として参考になる文献の数々を読む時に必要になるものとして一覧を挙げ、実際に解析していく中でヘッダファイルに先に注釈を付け大局的な視野を得た上でソースファイルに赴くという大局から局所に向かう段階のあることに気付けた。そしてLINUX 0.01 内核完全注釈という中国語の資料の作者である方の作ったフェイクバーションとして、LINUX 0.01に辿り着くまでの過程を実装したプロジェクトを見つけて最初に日本で紹介もしているしかし、私が解析したのは22個ある内の2個であり、0.01で言えばヘッダファイルは全て終えたがソースファイルに関しては零でシステムスタートアップコードとブートローダ以外は放っていた。後は時間があればどうとでも読めるだろうと考えていたということもあるけれど、とりあえずは読み切っておきたいということを最近になって思い、又、解析を始めることに決めました。さて理想的な選択が0.01だと主張した当時、その影響を受けてか何人かのプログラマは自主的にカーネル読みを始めたので今になって調べると読むのに苦労など全くないほど日本語の情報に満ち溢れていることに気付けます。そろそろ読んでおこうという感じです

当時の私はBackTrack 4 R1というディストリビューションの上で解析していました。けれど、デュアルブート環境にすることやデスクトップPC等で使い分けるより 一つの高スペックなノートPCの上で仮想エミュを通して操作する方が便利だと感じています ThinkPad T410i - Core i3:32bを使い続けていますがメモリ3GBでも軽快に動いてくれるので　VMWare WorkStation 8.04にBT4R1を導入してカーネルのビルドやエミュレートを行い、ソースファイルのやり取りには Dropboxを利用することにしました 尚、日本語化は面倒なので行わずに作業することにします。いつか同じ事をする方の参考になるようにここまでを纏めておきます

1. VMWare WorkStation 8.04 にBackTrack 4 R1をインストール:日本語化は無し
2. ｼｽﾃﾑﾄﾚｲ{Resize and Rotate, Keyboard Layout}は1280x768, Japaneseに設定
   KDEﾊﾟﾈﾙはデスクトップ上部に移動してﾀｽｸﾊﾞｰやﾒﾆｭｰを開きやすいようにする
3. そのままのリポジトリは扱えないため、新しいリポジトリのURLに入れ替える
   ```
   ## EOL upgrade /etc/apt/sources.list
   # Required
   deb http://old-releases.ubuntu.com/ubuntu/ intrepid main restricted universe multiverse
   deb http://old-releases.ubuntu.com/ubuntu/ intrepid-updates main restricted universe multiverse
   deb http://old-releases.ubuntu.com/ubuntu/ intrepid-security main restricted universe multiverse
   ```
4. 8086用アセンブラ[bin86]及びPCエミュレータ[qemu]をインストールしておく

```
aptitude install bin86 qemu
```

5. ホストOSとゲストOS間によるファイルやり取りにDropboxをインストールする
```
dropbox start -i
```
6. ソースコードの解析や注釈はKateというテキストエディタを使ってのみ行う
リモートデバッグが必要になった場合は、 Eclipseに移行するつもりでいる

これで一先ずパッケージのインストールやカーネルのビルドを行うことが出来ます。今の時代ならば統合開発環境によるリモートデバッグ等、便利な機能をモリモリ使って解析できますが90年代のハッカー達は今と比較すると、 とてつもない非力な環境でもカーネルを作り上げてきました。根本的に重要な力所は互換性を引き継ぐCPUの歴史から辿るように伺い知ることが出来るのですと感じているがここまま書いていくと非常の質の低い文章を書き垂らすことになると思うので文体や、カーネルへのアプローチも含めて再考することにした。オペレーティングシステムを学ぶという意味ではやはり２０年の歴史を学ぶぐらいの気概を持ちたい