nest# hyakumangoku

百萬石 2010 秋号

Microcomputer Making Association

部長挨拶

kyogoku42
kyogoku42@mma.club.uec.ac.jp

こんにちは.

百萬石を手にとって頂きましてありがとうございます.
今年は例年とかわって多くの新入生が入ってきましたので，今回の秋号は記事の本数も大幅に増えました.

MMA とは, ”Microcomputer Making Association” の略称からきています.
とても古くからあるサークルだそうで[*1]もともとはマイコンで遊ぼう，という趣旨のサークルだったそうですが,
今ではソフト面を多く扱っています.

MMA に興味があるかた，部員は年中募集しておりますので，気兼ねなく サークル会館 2 階 部室 へどうぞ.

[*1] 過去の部誌を見る限りで, 1978 年より前からあったそうですね.

目次

ssh と学内無線を駆使した VPN (wakisaka) **1**

1 **VPN** とは **1**

2 なぜ **VPN** が必要になったか **1**

3 実験 **1** とりあえず **VPN** を作ってみよう **1**

4 実験 **2** ブリッジしてみよう **1**

5 実験 **3** ブリッジ経由で **DHCP** を使ってみよう **2**

6 本運用に向けて **2**

7 最後に **3**

HTTPS 通信に対して透過的にプロキシを経由させる (ytoku) **4**

1 事の始まり **4**

2 **HTTP** における実現方法 **4**

3 通常の、プロキシサーバを介した **HTTPS** 通信の流れ **4**

4 具体的に **Squid** は **pf** からどうやって情報を得ているのか **5**

5 どのように通信を仲介するか **5**

6 今後の課題 **6**

アセンブリ入門講座 (hiro1357) **7**

1 そもそもコンピュータって何？ **7**

2 メモリ **7**

3 **CPU** **7**

4 プログラミング言語とは **7**

5 アセンブリ言語の特徴 **7**

6 アセンブリ言語の種類 **7**

7 **x86CPU** のレジスタ **8**

8 スタックとキュー **8**

9 アセンブリ言語の主な命令 **32bit** **9**

10 アセンブリ言語で書いてみよう！ **9**

11 API **10**

12 ループ処理 **10**

13 セグメントとオフセット **11**

14 サブルーチン **11**

15 数値の演算 **12**

16 BIOS コール **14**

17 さいごに **16**

OCaml 入門 (fujii) 17

1 OCaml とは **17**

2 とりあえず使ってみる - 演算、型推論 **17**

3 関数定義 - 部分適用 **18**

4 リスト **18**

5 パターンマッチ **19**

6 高階関数 - 匿名関数 **19**

7 おわりに **20**

Haskell 入門 (kyogoku42) 21

1 関数型の言語とはなんだろうか. **21**

1.1 手続き型の言語 21

1.2 関数型の言語 22

2 関数型である利点 **23**

3 遅延評価 **23**

4 注意 **23**

Let's note CF-W2 の HDD 交換 (wakisaka) 24

1 Let's note CF-W2 とは **24**

2 IO error **24**

3 交換の問題点 **24**

4 dry-run **24**

5 本番 **25**

6 これから挑戦する人に **26**

7 後日談 **26**

残念な目的の為の自作 PC (shimazaki) 27

1 **PC 構成の選定** **27**
1.1 CPU 27
1.2 マザーボード 27
1.3 メモリ 27
1.4 HDD 27
1.5 光学ドライブ 27
1.6 ケース 28
1.7 チューナー 28

2 各パーツの組み立て **28**

3 インストール後の設定 **28**

4 拡張、そして何所へ向かうのか **28**

5 まとめ **29**

PrivateIP しか振られていない自宅に外部からアクセス (sawada) 30

1 はじめに **30**

2 どんなサービス？ **30**

3 メリット・デメリット **30**

4 導入 **30**
4.1 sshd 設定 30
4.2 PacketiX VPN Client 導入 31
4.3 PacketiX VPN Client 設定 31
4.4 DNS 設定 32

5 最後に **32**

fluxbox のオレスタイル (hayakawa) 33

1 準備 **33**

2 書き方 **33**

3 設定項目 **33**

4 作成したオレスタイル **34**

5 最後に **35**

高度情報通信ネットワーク社会における人間存在の超自然的な在り方 (yamazaki) 36

1 序章 36

2 懸念の想起 36

3 感覚的ヒロイズム 37

4 結論 37

UNIX 系 OS を使う人々を分ける二つの戦争 (松宮　遼) 38

1 カーネル戦争 38
1.1 概要 38
1.2 BSD カーネル 38
1.3 Linux カーネル 38

2 エディタ戦争 38
2.1 概要 38
2.2 emacs 38
2.3 vim 38

3 最後に 38

iFrog (noy) 39

1 大きな驚きをあなたに 40

2 最高の素材 40

3 信じられない製法 40
3.1 潰す 40
3.2 頭を畳む 40
3.3 股を作る 40
3.4 足を作る 40
3.5 完成 40

4 iFrog をパーソナライズする 41
4.1 丸みをつける 41
4.2 頭を立てる 41

ssh と学内無線を駆使した VPN

wakisaka
wakisaka@mma.club.uec.ac.jp

概要
サークル棟と西 9 を結ぶ学内最長 (推定) の VPN を作った記録です。

1 VPN とは

VPN とは Virtual Private Network の略で、大雑把に説明するとネットワークの中にネットワークを通す技術です。この技術を使用することにより、複数のローカルネットワークを接続して一つのネットワークのように扱ったり、信頼できないネットワークで安全にデータを通すことができます。sawada 君の解説している筑波大 IPv6 実験もこの VPN 技術を利用することで面倒な手続き無しに、VPN クライアントを入れるだけで IPv6 を扱える環境を実現しています。

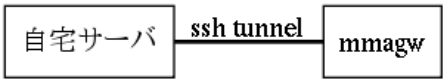
2 なぜ VPN が必要になったか

さて、学内には学内無線 LAN が各教室に届いており、サークル棟にあるサーバに接続するだけならば学内無線 LAN を通して ssh すれば接続することが出来ます。しかし、MMA の端末は先日全てネットワークを利用して起動するシンクライアント化が完了しており、このままでは利用することが出来ません。[*1]かといってサークル棟のサーバを動かすわけにもいかないので、「物理的に持っていけないなら論理的に持っていけばいいじゃない」という発想のもとサークル棟-西 9 間の VPN を構築することになりました。

3 実験 1 とりあえず VPN を作ってみよう

とりあえずつくってみます。今回は Unix 系なら必ずと言っていいほど入っていてお手軽な ssh を使ってみました。sshd の設定で PermitTunnel を yes に設定、クライアント側が ssh -w0:0 などとすると VPN を作れるということなので、試しに自宅と MMA 側ゲートウェイ予定地の間に VPN を作ってみました。ssh におけるトンネルには 2 種類あり、上層部だけを転送するモードと完全にケーブルのような挙動をするモードがあります。今回は後者を。

```
home# ssh -w0:1 -oTunnel=ethernet -v mmagw
なんかいろいろデバッグメッセージが出てくる
mmagw# ifconfig tap1 192.168.14.1 netmask 255.255.255.0
home# ifconfig tap0 192.168.14.2 netmask 255.255.255.0
```



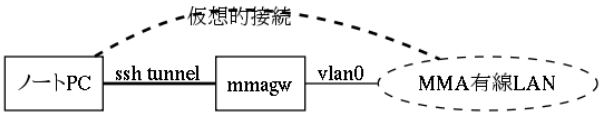
これだけです。ping するとしっかり通信していることが分かります。ただ、これでは別々のネットワークなので実用に移すためにはもう少々工夫が必要です。

4 実験 2 ブリッジしてみよう

次に、2 つ以上のネットワークを 1 つにまとめる「ネットワークブリッジ」の設定をしてみます。これがうまくいくと、MMA 内部の LAN を VPN 経由で遠隔地に飛ばすことができます。ネットワークブリッジは複数個のネットワークデバイスを指定するだけで簡単に作れます。[*2]さすがに家のサーバに MMA のネットワークが飛んでくるとまずいので部室にノート PC を持ち込んで実験します。

[*1] ネットワークブートには通常、有線 LAN と DHCP,tftp,nfs のサーバが必要です。
[*2] FreeBSD の場合。

```
note# ssh -w0:1 -oTunnel=ethernet -v mmagw
mmagw# ifconfig bridge create
bridge0
mmagw# ifconfig bridge0 addm vlan0 addm tap1 up
```



これで実験したところ、IPv4 のアドレスが飛んでこなかったのでおかしいとは思いつつ、IPv6 のアドレスは取れていたのでまあよしとしました。ここで指定している vlan0 は MMA で使っている仮想ネットワークデバイスです。1 つのネットワークを擬似的にいくつかのセグメントに分割するために vlan を導入しています。ちなみに、vlan0 は MMA 内部の有線 LAN 用セグメントなので、VPN には乗せずに無線 LAN 用のセグメントを VPN で飛ばす予定でした。この時はあくまでテストのつもりだったのですが …[*3]

5 実験 3 ブリッジ経由で DHCP を使ってみよう

さて、うまくブリッジが動いたように見えたのでとりあえず西 9 に持っていくためのゲートウェイ (以下 w9gw) を作ります。MMA で使っている HP のシンクライアントは 2GB のフラッシュを積んでいるので自力で起動可能です。このシンクライアントは 4 台あるので 1 台をゲートウェイとして供出しました。無線 LAN も搭載しているので学内無線に接続することも可能ともってこいの性能です。フラッシュメモリに FreeBSD をインストールし、無線を稼働させ[*4]、なんとか学内無線に接続させることに成功しました。同じ部屋にあるのに学内無線経由で接続するのはなんだか残念な気分になりますが実験なので仕方ない。今回は実運用と同じ構成の vlan1 をブリッジしました。

```
w9gw# ssh -w0:1 -oTunnel=ethernet -v mmagw
mmagw# ifconfig bridge create
bridge0
mmagw# ifconfig bridge0 addm vlan1 addm tap1 up
w9gw# dhclient tap0
w9gw# ifconfig tap0
inet 192.168.11.xxx netmask 0xffffff00 broadcast 192.168.11.255
```

ちょっとした事故[*5]の後、無事に MMA 内部の IP を取得することができました。これは同時にトンネルとブリッジを経由して DHCP サーバーへの接続に成功したことを意味し、ネットワークブートに一歩近づきました。

6 本運用に向けて

本運用ではこの端末を西 9 に持っていった上でハブを繋ぎ、そこに端末と無線 AP を繋ぐことでネットワークブートと MMA の無線ネットワークを実現する予定でした。しかし、完成を目前に大きな壁が立ちはだかることになりました。

- 無線 LAN 用セグメントからはネットワークブートができない
- nfs サーバのある有線 LAN 用セグメントはセキュリティ面で現地の無線に載せたくない
- かといって nfs がないとネットワークブートはできない

本来なら諦めて有線 LAN 用セグメントを飛ばしてしまいたくなるこの状況ですが、全てを一挙に解決できる画期的な方法が出てきました。

[*3] vlan1 の無線 LAN 用セグメントだと ssh の接続用に使ってしまっていたのでブリッジのおかげでつなげているのか分からず、テストにならないのです。
[*4] 無線稼働までに壮絶な戦いがあったことは言うまでもありません。
[*5] w9gw 側の tap0 を up し忘れていたなんて言えない

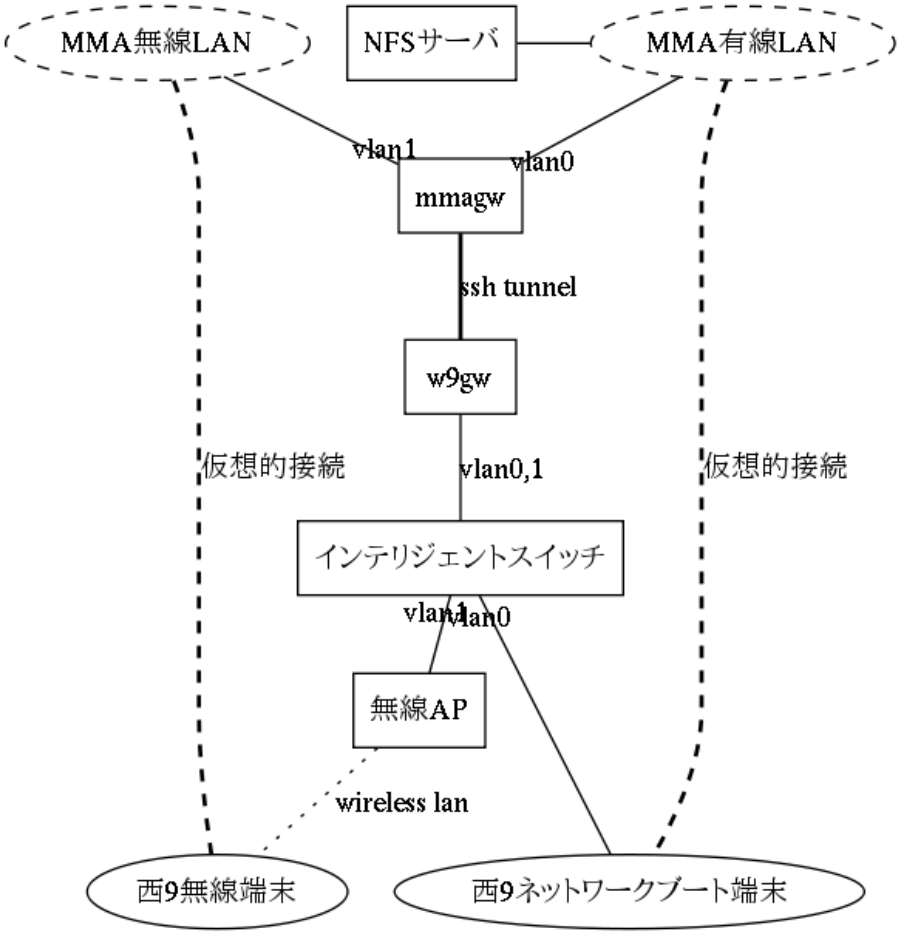
vlan 両方飛ばしちゃえばいいじゃない

vlan タグは現地にもインテリジェントスイッチを置いて外し、mmagw と w9gw の間は生の vlan を流せば現地でセグメントを分けることができます。じゃあ mmagw の vlan じゃない物理デバイスと tap をブリッジングすれば … と思ったのですが、思わぬ落とし穴。なんと、**学内ネットワークも同じ物理デバイスを共有する vlan** だったのです。[*6]

学内ネットワーク用の vlan まで tap デバイスに飛んでしまうと、tap デバイスの通信は学内ネットワーク用の vlan を通すので、無限ループとパケットの増殖を起こして最悪学内ルータを止めてしまいます。[*7]なので、物理デバイスを直接指定する方法は使えません。

使える物理デバイスがないなら増やせばいいじゃない

ということで、mmagw の繋がっているインテリジェントスイッチに MMA 内部の無線、有線 LAN のセグメント 2 つのみを流すポートを作り、それを mmagw の余っているネットワークカード[*8]に繋ぐことで MMA のセグメント 2 つの通信のみを拾うネットワークデバイスが完成しました。このネットワークデバイスと tap デバイスをブリッジすることで、めでたく西 9 に学内以外の vlan を VPN で送ることができます。あとは、w9gw で tap デバイスと自身の有線カードをブリッジしてインテリジェントスイッチに流しこめば無事に現地でネットワークブートもでき、無線 AP からは nfs に繋げない環境が作れるでしょう。



7　最後に

端末のネットワークブートが成功したかはこの記事には書けません。この計画の結末はぜひ調布祭当日に皆さんの目でご確認ください。

---

*6 こんな設定にしたのは誰かというとそれは私です

*7 その前に MMA 側のネットワークが落ちるとは思いますが。

*8 蟹さんのオンボード LAN

HTTPS 通信に対して透過的にプロキシを経由させる

ytoku (徳重佑樹)
ytoku@mma.club.uec.ac.jp

概要
プロキシが設定できない Android のために、透過的にプロキシサーバを通じて HTTPS 通信が行えるようにするためのプログラムを書きました。

1 事の始まり

Android 携帯電話が日本でも随分と普及してきた今日この頃、MMA では IS01 —通称メガネケース—が流行しています。私も HTC Desire を入手しました。さて、MMA はグローバル IPv4 アドレスを持っていないため、学外のウェブサイトを閲覧するには基本的に大学のプロキシサーバを経由する必要があります。しかし、実は Android にはプロキシサーバを設定することができないらしいのです。

そこで、まず HTTP 通信を透過的にプロキシサーバに転送する措置が執られました (yajima 先輩による)。これはゲートウェイとなっているサーバ (FreeBSD) 上に Squid(プロキシサーバ) をインストールして、pf[*1]上で外部の 80 番ポートへの通信を Squid に転送し、Squid 上で大学のサーバへのリクエストに書き換えるという方法でした。

しかし、以下の節で説明するような理由で、Squid では HTTPS 通信を透過的に転送することは出来ませんでした。

2 HTTP における実現方法

HTTP のプロトコルは、最初の一行にメソッドとパスとプロトコルを指定し、続けてヘッダのフィールドと値を指定するようになっています。ヘッダは空行が入るまで続き、空行の後が Content です。プロキシサーバを介して通信するためには、プロキシサーバに接続先のサーバはどこかを伝える必要があるので、パスの部分を絶対 URI 形式します。プロキシサーバはリクエストを解釈し、必要に応じていくつかのヘッダを加え、あるいは内容を書き換えて対象のサーバに通常の HTTP リクエストを発行します。

通常の HTTP リクエスト

```
GET /path/to/file HTTP/1.1
Some-Header: hoge
...
```

Proxy に対する HTTP リクエスト

```
GET http://host/path/to/file HTTP/1.1
Some-Header: hoge
...
```

Squid はコンパイル時に pf を用いて透過プロキシを実現する機能を有効にしてあり、接続元のアドレスとポートの組から本来どこのサーバに接続しようとしていたかを検索して、大学のプロキシサーバに対するリクエストを生成するようになっています。

さて、ここまでは暗号化されていない通常の HTTP の話でした。ところが、SSL 通信においては、プロキシサーバはこの方法は実現できません。なぜなら、この方法はプロキシサーバが一度リクエストを解釈して接続先を把握して、絶対 URI になっている部分をパスに書き換えたリクエストを生成する必要がありますが、通信が暗号化されているということは通信内容を内容を知ることが出来ないからです。

3 通常の、プロキシサーバを介した HTTPS 通信の流れ

プロキシサーバを介して HTTPS 通信を行うには CONNECT メソッドを用います。CONNECT メソッドはパスの代わりに接続先のホストとポート番号をとり、そのサーバへのトンネルを生成します[*2]。

[*1] FreeBSD の Packet Filter。ファイアウォールや NAT などの機能を持つ
[*2] セキュリティのために無効化されていたり、接続先ポートが限定されていることも多いでしょう。

HTTPS 通信を始めるための HTTP リクエスト

```
CONNECT host:port HTTP/1.1
Host: host:port
...
```

よって、HTTPS で透過的に大学のプロキシサーバを介するためには、Squid の代わりに、大学のプロキシサーバに向けて CONNECT メソッドを発行するプログラムを書いてやれば実現できます。

こうして見てみると実現が簡単ですが、なぜ Squid に実装されていないのでしょうか。そもそも Squid はキャッシュ機能を売りにしたプロキシサーバソフトウェアですので、恐らく、通信内容が分からないような状況をサポートする意義が無かったのでしょう。

4 具体的に Squid は pf からどうやって情報を得ているのか

プログラムを作るにあたって、まず最初に調べたのは Squid がどのようにして pf から本来の接続先を得ているかです。pf を用いた透過プロキシサポートを有効にするための configure のオプションは`--enable-pf-transparent` なので、これを元に調べていったところ、`src/ip/IpIntercept.cc` に

```
int IpIntercept::PfInterception(int fd, const IpAddress &me, IpAddress &client,
                                IpAddress &dst, int silent);
```

という型の関数があり、この中で本来の接続先を取得していました。この関数で何をやっていたかというと、

(1) `/dev/pf` を開く
(2) `struct pfioc_natlook` 構造体 `nl` を用意
(3) `nl.s*`に接続元 (クライアント) のアドレス・ポートを代入
(4) `nl.d*`に Squid 側のアドレス・ポートを代入
(5) `nl` を引数に `ioctl` で `DIOCNATLOOK` コマンドを呼び出す
(6) 本来の接続先を `nl.rd*`から取得

というものです。なお、`DIOCNATLOOK` コマンドのドキュメントが pf(4) にあったのでこれも参考にしました。

5 どのように通信を仲介するか

作成するプログラムは、大学のプロキシサーバにトンネルを作った後に、クライアントと大学のプロキシサーバの間の通信を仲介することになるわけですが、この通信は双方向に可能でなければなりません。そのため、単純に片方から読み出してもう片方に書き込むというわけにはいきません。ちゃんと実装しようとすればそれなりに面倒そうなプログラムですが、これは `connect.c` や `nc` が行うことそのものです。

と、いうか。

connect.c

```
 *   For a HTTP-PROXY connection:
 *     $ connect -H proxy-server:8080  host 25
...
    if (sendf(s,"CONNECT %s:%d HTTP/1.0\r\n", dest_host, dest_port) < 0)
        return START_ERROR;
...
```

man nc

```
-X proxy_protocol
        Requests that nc should use the specified protocol when talking
        to the proxy server.  Supported protocols are ‘‘4’’ (SOCKS v.4),
        ‘‘5’’ (SOCKS v.5) and ‘‘connect’’ (HTTPS proxy).  If the protocol
        is not specified, SOCKS version 5 is used.

-x proxy_address[:port]
        Requests that nc should connect to hostname using a proxy at
        proxy_address and port.  If port is not specified, the well-known
        port for the proxy protocol is used (1080 for SOCKS, 3128 for
        HTTPS).
```

元からありました。

そういうわけなので、作るプログラムでは本来の接続先だけ調べて、`connect.c` に丸投げ (`execl`) することにしました。また、簡単のために listen する部分は inetd に任せてしまい、プログラムは標準入出力でクライアントと通信することにしました。

```
> grep ssl-proxy /etc/services
ssl-proxy       3129/tcp
> grep ssl-proxy /etc/inetd.conf
ssl-proxy       stream  tcp     nowait          squid   /home/ytoku/bin/ssl-forward
```

6 今後の課題

これで HTTPS の通信も通るようになって幸せになれたかに思われました。事実、ブラウザを使った通常の HTTPS 通信は問題なく行えています。ところが Android のマーケットからのインストールが動作していません。tcpdump してみたところでは Port80 への HTTP 通信と Port443 への HTTPS 通信 (及び NTP など関係のない通信) 以外は行われていませんでしたので、どうやら HTTPS の転送が正しく動作していないのではないかという結論に至りました。残念です。原因究明とデバッグが今後の課題です。

アセンブリ入門講座

hiro1357

http://eserver.dip.jp/

eserver_dip_jp@yahoo.co.jp

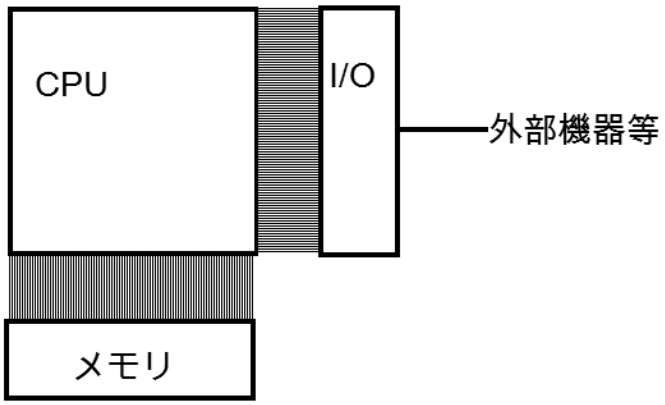
概要

アセンブリ言語は、コンピュータについて学ぶ上で基礎となる大変重要なものです。というのも、アセンブリ言語は、CPU が直接理解する機械語 (2 進数で表される) にほぼ 1 対 1 で対応する唯一の言語だからです。そんな難しいものを何でしなきゃいけないの?とか言わないでください。とにかく、やるんです。アセンブリ言語は、思っているよりも難しくありません。場合によっては、他のプログラミング言語よりも簡単です。

1　そもそもコンピュータって何？

… ということが分かってないと、アセンブリ言語なんて理解できません。普段は、なんとなく便利な機械として使っているだけですが、今回はいつもと違った方向からコンピュータを見ていきます。

コンピュータは基本的に、CPU、I/O、メモリの 3 つでできています。 CPU にメモリと I/O が接続され、I/O に外部の入出力装置が接続されます。 I/O は外部とやりとりをする部分です。CPU と外部との橋渡しをしてくれます。 CPU はメモリのプログラムを順番に読み込んで処理していきます。I/O からの情報が必要であれば、 I/O にアクセスします。



2　メモリ

0 番地	1 番地	2 番地	3 番地	4 番地	5 番地	6 番地	7 番地
入れろ A 2		入れろ B 8		足せ A B		書き込め 10 番地	

メモリは、ある一定ビット単位で区切られており、それぞれに番地という名前がついています。 そして、その中に命令やデータが順番に書き込まれています。さて、ここからが重要です。CPU はどのようにして命令やデータを処理しているのでしょうか。

3　CPU

CPU は命令デコーダ回路、演算回路、レジスタ、プログラムカウンタからなります。CPU はこれら の回路を使って次のような処理を繰り返し行っています。

(1) 入力 （プログラムカウンタの指し示すメモリアドレスから）
(2) 命令の解釈 （命令デコーダ）
(3) 演算 （演算回路）
(4) プログラムカウンタを加算
(5) 1 へ戻る

プログラムカウンタというのは、現在実行しているメモリの番地 (アドレス) を指すものです。プログラムカウンタの指す番地から、 命令やデータが読み込まれます。 命令デコーダは、読み込まれた命令を解釈し、データを適切な演算回路へ渡します。演算回路は、実際に演算を行う回路です。1 つの演算回路ですべての計算を行うことはできないので、加算、減算、乗算、除算などそれぞれ演算のための回路があります。（だから命令デコーダが必要）レジスタは、CPU で演算をするために、データを入れておくための場所です。CPU には (アセンブリ言語には) 変数というものがありません。その代わりに、有限個のレジスタと呼ばれる場所が用意されています。アセンブリ言語を覚えることは、レジスタとメモリの使い方を覚えることとほぼイコールになります。

4　プログラミング言語とは

メモリの中にプログラムが入っていて、それを CPU が順番に解釈して実行しているわけですが、自 分の必要とする計算を CPU にさせるには、プログラムを自分でメモリに書き込む必要があります。プログラムは自分で作成するわけですが、CPU が解釈できるプログラムは数値の羅列、しかも、1 と 0 だけの 2 進数になっています。これを人間が直接理解するのは大変困難です。 （たまに、16 進数であらわしたバイナリが読めちゃう変人が居ますが … ） そこで登場したのがアセンブリ言語と呼ばれるプログラミング言語です。CPU が理解できる命令に 人間が理解できる名前 （ニーモニックと言う） をつけたものです。分かりやすい名前をつけることで、 人間にもプログラムを理解しやすくなります。 お待たせしました。やっとここから本題です。

5　アセンブリ言語の特徴

アセンブリ言語は、CPU が理解できる命令に人間が理解できる名前をつけたものです。ですから、 アセンブリ言語で記述した命令は、CPU が理解できるバイナリコードとほぼ 1 対 1 で対応しています。CPU に直接命令をできると思っていいです。CPU を直接操作するので、上手にプログラミングをすれば、大変高速なプログラムを作成することができます。

6　アセンブリ言語の種類

アセンブリ言語の書き方には、Intel 形式と AT&T 形式の 2 種類があります。基本的な書き方は似ていますが、びみょーに書き方が違います。で、見た目はぜんぜん違います。 Intel 形式は、

```
mov eax, 1h
mov ebx, 1h
add eax, ebx
```

一番左側が命令になります。真ん中に書かれているものがレジスタ名で、ここに書かれたレジスタに 計算結果の数値が代入されます。右側が計算されるもうひとつのレジスタ名または数値になります。AT&T 形式は、

```
movl $1, %eax
movl $1, %ebx
addl %ebx, %eax
```

AT&T も一番左側が命令になります。Intel 形式と違い、真ん中に書かれているものが計算されるもうひとつのレジスタ名または数値になります。右側がレジスタ名で、ここに書かれたレジスタに計算結果の数値が代入されます。mov 命令にくっついている l は、オペランドの型をあらわしています。

```
l=long word=32bit,w=word=16bit,b=byte=8bit
```

命令を「オペコード」計算結果が代入されるレジスタ名を「デスティネーションオペランド」 計算されるもうひとつのレジスタ名または数値を「ソースオペランド」と呼びます。 今回は、書き方がシンプルな Intel 形式を使うことにします。

7 x86CPU のレジスタ

私たちが普段使っている PC の CPU は大半が x86 系の CPU だとおもいます。今回は、普段使っている PC でアセンブリ言語を勉強するので、x86CPU(32bit) に用意されているレジスタについて説明します。

x86CPU には、たくさんのレジスタが用意されていますが、ここではよく使われる基本的なレジスタを紹介します。

31 16	15 0	
32bit で eax 16bit で ax →	前半 8bit が ah	後半 8bit が al
32bit で ebx 16bit で bx →	前半 8bit が bh	後半 8bit が bl
32bit で ecx 16bit で cx →	前半 8bit が ch	後半 8bit が cl
32bit で edx 16bit で dx →	前半 8bit が dh	後半 8bit が dl
32bit で esi	16bit で si	
32bit で edi	16bit で di	
32bit で esp	16bit で sp	
32bit で ebp	16bit で bp	
32bit で eip	16bit で ip	
フラグレジスタ (各 1bit) OF SF ZF CF など		

a,b,c,d は汎用レジスタと呼ばれ、どのようなことに使ってもよいレジスタとなっています。(一応、a はアキュムレータ (演算用)、b はベース (メモリのベースアドレス用)、c はカウンタ (ループカウント用)、d はデータ (データ用) となっており、演算の内容によって は各レジスタごとにできるものとできないものが存在します。)

si,di はインデックスレジスタと呼ばれ、メモリアドレスの計算に使われます。

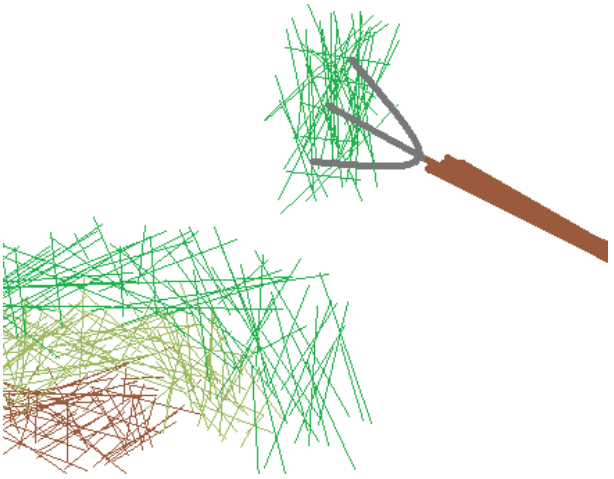
sp,bp も汎用レジスタですが、スタックのメモリアドレスを入れておくために使われます。

ip は命令ポインタ (インストラクションポインタ) と呼ばれ、現在実行中のメモリアドレスを入れてお くために使われます。このレジスタは、私たちが命令によって変更することはできません。

フラグレジスタは比較の結果や値の正負、桁あふれなどを示すために使われます。

8 スタックとキュー

アセンブリ言語では、スタックと呼ばれる方式でデータが管理されることが多いため、説明をしておきます。スタックというのはデータ構造の一つです。もともとスタックというのは積み上げられた干草のことです。積み上げられた干草は、最近新しく積み上げた部分は容易に取り出すことができますが、それよりも前に積み上げた部分はその上に積み上げられた新しい部分を取り除いてからでないと取り出すことができません。このような方式を FILO(FirstInLastOut) または LIFO と呼びます。スタックとよく対比されるデータ構造にキューというものがあります。キューは待ち行列とも呼ばれます。昼間、学食に長い行列ができますよね。先に並んだ人が早く昼食にありつけます。このような方式を FIFO(FirstInFirstOut) と呼びます。



9　アセンブリ言語の主な命令 32bit

(mul、div などビット数によって挙動が異なるものがあります)

命令	例	説明
転送命令	mov eax, ebx	ebx の値を eax に転送。ソースオペランドには数値も指定可。
交換命令	xchg eax, ebx	eax の値と ebx の値を交換。
AND 命令	and eax, ebx	eax と ebx の AND 演算を行い、結果を eax に転送します。
OR 命令	or eax, ebx	eax と ebx の OR 演算を行い、結果を eax に転送します。
XOR 命令	xor eax, ebx	eax と ebx の XOR 演算を行い、結果を eax に転送します。
NOT 命令	not eax	eax の NOT 演算を行い、結果を eax に転送します。
NEG 命令	neg eax	eax の符号 (プラスかマイナスか) を反転します。
TEST 命令	test eax	AND 演算を行い、符合のみを変化させる。(演算結果は捨てる)
加算命令	add eax, ebx	eax と ebx の足し算を行い、結果を eax に転送します。
減算命令	sub eax, ebx	eax と ebx の引き算を行い、結果を eax に転送します。
INC 命令	inc eax	eax の値をインクリメント (1 加算) します。
DEC 命令	dec eax	eax の値をデクリメント (1 減算) します。
乗算命令	mul ebx	eax と ebx を乗算し、結果の上位 32 ビットを ebx、下位 32 ビットを eax に転送します。
符号付乗算命令	imul ebx	符号付の乗算を行います。
除算命令	div ebx	上位 32 ビットを edx 下位 32 ビットを eax とした 64 ビットのデータを、ebx で除算して商を eax、 余りを edx へ転送します。
符号付除算命令	idiv ebx	符号付の除算を行います。
比較命令	cmp eax, ebx	eax と ebx を比較し、eax=ebx なら ZF フラグを 1 に、eax<ebx ならば、CF フラグを 1 に、eax>ebx ならば CF フラグを 0 にします。
ジャンプ命令	jmp LABEL	指定したラベルの位置へジャンプします。
JA 命令	ja LABEL	CF フラグと ZF フラグが 0 のときにラベルの位置へジャンプ。
JNA 命令	jna LABEL	CF フラグか ZF フラグが 1 のときにラベルの位置へジャンプします。
JC 命令	jc LABEL	CF フラグが 1 のときにラベルの位置へジャンプ。
JNC 命令	jnc LABEL	CF フラグが 0 のときにラベルの位置へジャンプ。
JZ 命令	jz LABEL	ZF フラグが 1 のときにラベルの位置へジャンプ。
JNZ 命令	jnz LABEL	ZF フラグが 0 のときにラベルの位置へジャンプ。
JCXZ 命令	jcxz LABEL	ecx が 0 のときにラベルの位置へジャンプします。
ループ命令	loop LABEL	ecx をデクリメントし、ecx が 0 でないならラベルの位置へジャンプします。
LOOPZ 命令	loopz LABEL	ecx をデクリメントし、ecx が 0 でないかつ ZF フラグが 1 ならばラベルの位置にジャンプします。
LOOPNZ 命令	loopnz LABEL	ecx をデクリメントし、ecx が 0 でないかつ ZF フラグが 0 ならばラベルの位置にジャンプします。
PUSH 命令	push eax	eax の値をスタックへ入れます。
POP 命令	pop eax	スタックから値を取り出し、eax へ転送します。
CALL 命令	call LABEL	現在の位置をスタックへ入れてラベルの位置にジャンプ。
RET 命令	ret	スタックから CALL 命令を実行した位置を取り出し、または ret 2h ジャンプします。オペランドに指定したバイト数のスタックを捨てます

10　アセンブリ言語で書いてみよう！

さあ、やっとここまで来ました。アセンブリ言語で実際にプログラムを書いてみましょう！ 今回は、Intel 形式でプログラムを書くので、“NASM” というアセンブラを使おうと思います。 NASM はインターネットで配布されているので、次のアドレスからダウンロードしてください。

`http://www.nasm.us/`

`http://www.nasm.us/pub/nasm/releasebuilds/2.08/dos/nasm-2.08-dos.zip`

NASM の使い方ですが、NASM はコマンドラインで動作するプログラムですので、次のような引数 を与えて使います。

```
nasm.exe input.asm -fbin -o output.bin
```

input.asm　テキスト形式のアセンブリソースです。
-fbin　アセンブルするためのオプションです。
-o　出力ファイル名を指定します。
output.bin　出力されるバイナリファイル名です。

では、適当なテキストエディタを開いて書き始めましょう。 まずは、 MS-DOS 上で動く Hello World プログラムを作ってみます。MS-DOS で動くプログラムは、 メモリの 100 番地から始めなければならないので、

```
org 100h
```

とします。これは、NASM に「100 番地からプログラムを始めてください。 」と指示するものです。次に、プログラム本体を書き始めるために、次のように書きます。

```
section .text
```

これは、この行以降に書いたものが、何なのかを表すために使用します。プログラム本体は必ず .text セクションに書きます。他に、.data セクション (データ)、.bss セクション (メモリ確保) というセクションがあります。.data セクションには、文字列などのデータをまとめて書きます。 .bss セクションには、書き換え可能なメモリ空間を確保するための指示をまとめて書きます。 次の行には、とりあえず

```
main:
```

と書いておきます。これはラベルと言います。ラベルはプログラム内のアドレスを表しており、 call 命令や jmp 命令などでジャンプ先を指定する際に使われます。 今回はジャンプする命令を使用しないので意味はありませんが、一応ここからが main 関数だと言うことで。

11 API

“Hello, World”という文字列を出力しなければならないわけですが、困ったことに CPU にも アセンブリ言語にも、文字列や文字を画面に表示させる命令はありません。大体、なぜ画面に絵や文字が映るのかというと ··· コンピュータにはビデオメモリと言うものが用意されているのですが、そこに表示したい内容を書き込んでおくと、表示装置が勝手にそれを読み取って画面に表示してくれることになっています。 しかし、ビデオメモリを直接書き換えて文字を表示するには、その文字の形を 1 ドットずつ書き込んでいかなければなりません。

いちいち文字の形を 1 ドットずつ書き込むなんてやってられない!!!

そこで、オペレーティングシステムの出番です。オペレーティングシステムと言うのは、その名の通り、コンピュータを操作 (オペレート) するためのものです。オペレーティングシステムには、コンピュータを操作するためのたくさんの関数が用意されています。オペレーティングシステムを使えば、よく実行される共通した操作はたいていAPI と呼ばれる関数にまとめてあるので、プログラムを書く手間が大幅に軽減できるのです。(API とは、Application Programming Interface の略です。) オペレーティングシステムは、コンピュータを操作するオペレータが使いやすい環境を整えるだけでなく、プログラマがプログラムを作りやすい環境を整えるためのシステムでもあるのです。

今回このプログラムを動作させる OS は MS-DOS ですので、MS-DOS の API を使うことにします。 MS-DOS で文字列を出力する API 関数は次のようになっています。

レジスタの状態を

```
AH = 09H
DX = 文字列の先頭のアドレス
```

としてから、int 21h という割り込み命令 (int = Interrupt です。Integer じゃないよ！) を実行します。この関数では’$ ’が見つかるまで文字を出力し続けるので、文字列の最後に’=$ ’をつける必要があります。

DX には文字列の先頭のアドレスを入れておく必要がありますから、まずは表示する文字列データを準備しましょう。 文字列などのデータは、.data セクションに書く必要がありました。プログラムの一番下に新しく .data セクションを用意しましょう。

```
section .data
```

文字列データは次のように書きます。

```
TEXTDATA: db 'Hello, World',0Dh,0Ah, '$'
```

TEXTDATA: は、文字列データの先頭を表すラベルです。 データに名前を付けたと思うとわかりやすいと思います。 db 命令はデータを格納する命令です。データの型に よって、dd、dw などがあります

```
dd=dword=32bit
dw=word=16bit
db=byte=8bit
```

0Dh、0Ah はそれぞれ改行コードの CR と LF をあらわしています。最後は’$ ’で終わります。データが用意できたので、先ほ どの API 関数を呼び出してみます。

```
section .text
main:
        mov ah, 09h
        mov dx, TEXTDATA
        int 21h
```

これで、MS-DOS の文字列出力を行う関数が呼び出され、’Hello, World’が表示されるようになりました。今回のプログラムはこれだけで終了ですので、MS-DOS にプログラムが終了したことを伝えなければなりません。プログラムの終了を伝えるには次のように書きます。

```
 mov ah, 4Ch
 int 21h
```

AH レジスタに 4C を入れて、int 21h を呼び出します。 今までに書いたプログラムのリストは以下の通りです。

```
        org 100h
section .text
main:
        mov ah, 09h
        mov dx, TEXTDATA
        int 21h
        mov ah, 4Ch
        int 21h

section .data
TEXTDATA: db 'Hello, World',0Dh,0Ah, '$'
```

実際にアセンブルして実行してみます。

12 ループ処理

MS-DOS の API には、文字を 1 文字だけ出力する API 関数もあります。

```
AH = 02h
DL = 出力する文字
```

としてから、int 21h を実行します。 先ほどのプログラムを、このAPI 関数を使用するように書き直してみましょう。まず、どのようにすれば、1 文字だけ出力する関数で文字列を出力できるのかを考えてみます。とりあえず、何も考えずに 1 文字ずつ出力していくと、

```
        mov ah, 2h
        mov dl, 'H'
        int 21h
        mov dl, 'e'
        int 21h
        mov dl, 'l'
        int 21h
        mov dl, 'l'
        int 21h
        mov dl, 'o'
        int 21h
        …
```

となりますよね。たくさんの繰り返し部分があります。これはループで処理できそうです。 DL に文字データの代わりにアドレスを指定しても、そのアドレスからデータを取ってきて表示してくれるので、アドレスをインクリメントしながら処理していくことにします。

```
main:
        mov ah, 02h
        mov bx, TEXTDATA
putchar:
        mov dl, [ds:bx]
        int 21h
        inc bx
        jmp putchar
```

まず、bx レジスタに、'Hello, World' の 'H' のアドレスを転送します。DL レジスタには、[ds:bx] というアドレスを転送します。[ds:bx] はメモリ上の絶対アドレスを指します。 inc 命令を使って、bx を 1 ずつ加算していきます。ループ処理をするために、putchar:というラベル と jmp 命令を使いました。

13 セグメントとオフセット

[ds:bx] はメモリ上の絶対アドレスを指す。と言いましたが、これは x86 系 CPU がメモリを セグメントと呼ばれる単位で管理しているためです。(1 セグメント=64 キロバイト) コロン「:」で区切られた左側 (ここでは ds) はセグメントアドレスと言って、セグメントの先頭の絶対アドレスを指します。コロン「:」で区切られた右側 (ここでは bx) はオフセットアドレスと言って、セグメントアドレスを基準とした相対的なアドレスを指します。

プログラムの話に戻します。先ほどラベル putchar:と jmp 命令でループを作りましたが、このままでは無限ループになってしまいます。ループを抜ける処理を書かなければなりません。前回使った API 関数では、'$' で文字列の出力を終了することになっていました。ですので、これを真似てみることにします。

今回のプログラムでは、メモリから 1 文字ずつ取り出して処理しているので、取り出した文字が '$' で あるかどうかを判定すればいいですね。 2 つの値の比較を行うには、cmp 命令でした。cmp 命令は、 2 つの値を比較し、フラグレジスタに 結果を入れることになっていました。今回は、メモリから取り出した値が '$' と同じかどうかを判定するため、ZF フラグを確認することにします。

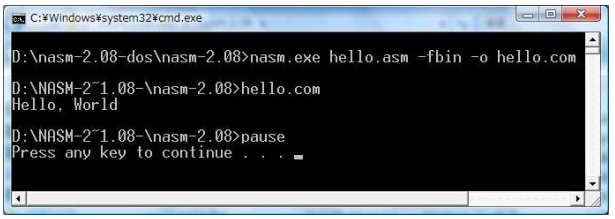
cmp 命令は 2 つの値が同じだった場合に ZF フラグを 1 にするので、ZF フラグが 1 のときに ジャンプする jz 命令を使います。

```
        org 100h
section .text
main:
        mov ah, 02h
        mov bx, TEXTDATA
putchar:
        mov dl, [ds:bx]
        cmp dl, '$'
        jz endchar
        int 21h
        inc bx
        jmp putchar
endchar:
        mov ah, 4Ch
        int 21h

section .data
TEXTDATA: db 'Hello, World',0Dh,0Ah, '$'
```

cmp 命令と jz 命令を書く位置に注意してください。int 21h を呼び出してからジャンプすると、終端 文字の '$' まで出力されてしまいます。

実際にアセンブルして実行してみます。



同じ動作をするプログラムなので、 動作している段階でははっきりとした違いはわかりませんね (^^;)

14 サブルーチン

せっかくなので、文字列の出力部分をサブルーチンにしてみましょう。ルーチンとは、ひとまとまりの作業、つまり、小さなプログラムのことで、プログラムの基幹部分として 動作するルーチンをメインルーチン、他のルーチンから呼び出されるルーチンをサブルーチンと呼び ます。 サブルーチンを作るためには、call 命令と ret 命令を使います。

```
        org 100h
section .text
main:
        mov bx, TEXTDATA
        call print
        mov ah, 4Ch
        int 21h
print:
        mov ah, 02h
putchar:
        mov dl, [ds:bx]
        cmp dl, '$'
```

```
        jz endchar
        int 21h
        inc bx
        jmp putchar
endchar:
        ret

section .data
TEXTDATA: db 'Hello, World',0Dh,0Ah, '$'
```

call 命令は、call 命令の位置をスタックへ格納し、ラベルへジャンプする命令です。 ret 命令は、スタックから call 命令の位置を取り出し、call 命令の位置へ戻る命令です。

とりあえず、サブルーチンにはなったようです。しかし、このままでは、このサブルーチンを他のプログラムへ移植した場合に困ったことになりそうです。なぜなら、このサブルーチンは dl と bx の値を書き換えてしまうからです。そのようなことにならないためには、dl と bx の値をどこかへ退避させておく必要があります。 そこで、push 命令と pop 命令を使います。 push 命令は、レジスタの値をスタックへ格納する命令です。 pop 命令は、スタックから値を取り出し、レジスタへ戻す命令です。

改良したプログラムのリストは次のようになります。

```
        org 100h
section .text
main:
        mov bx, TEXTDATA
        call print
        mov ah, 4Ch
        int 21h
print:
        push ax
        push bx
        push dx
        mov ah, 02h
putchar:
        mov dl, [ds:bx]
        cmp dl, '$'
        jz endchar
        int 21h
        inc bx
        jmp putchar
endchar:
        pop dx
        pop bx
        pop ax
        ret

section .data
TEXTDATA: db 'Hello, World',0Dh,0Ah, '$'
```

pop する順番を間違えないでください。push のときと逆の順番になります。スタックは最初に入れたデータが最後に出てくるので、入れるときの順番で取り出してしまうと、格納先が入れ替わってしまいます。

15 数値の演算

Hello, World が作れるようになったので、次は割り算をするプログラムを作成してみようと思います。え〜っと、割り算って言ったら、div、idiv、という命令があったよね。確かにそうですが、気づいてもらいたいことがひとつあるのです。足し算、引き算ができれば、特別な命令がなくても掛け算、割り算ができるんですよ!

というわけで、加算命令、減算命令を使って割り算をする方法を考えてみましょう。

まず、割り算はどのようにして計算していたでしょうか。とりあえず、40 ÷ 6 を計算する場合を考えてみることにします。

40 ÷ 6 を計算するときに、まず考えるのは、6 が 40 にいくつ含まれるか、ということですよね。で、余りがある場合には、6 × (含まれる個数) を 40 から引いて余りを出していました。余りは必ず、 含まれる個数 > 余り となっていました。(人によっては違う方法で計算しているかもしれませんが … ^^;)

6 は 40 に 6 含まれていますから、6 × 6=36 で 40-36=4。つまり、 6 余り 4 ですね。(6>4)

これを計算式に表すと、このようになります。

$$40 - 6 \times 6 = 4$$

これ、足し算と引き算で表せますね。

$$40 - (6 + 6 + 6 + 6 + 6 + 6) = 4$$

さらに、分配法則を適用して

$$40 - 6 - 6 - 6 - 6 - 6 - 6 = 4$$

気づきました?今のは答えが分かっていましたが、普段は答えは分からないので、

$$40 - 6 \times a = b$$

これだけではだめなので、条件 $6 > b$

$$40 - \overbrace{6 - 6 - 6 \cdots}^{a\ 回} = b$$

ということになりますね。 つまり、40 から 6 を引いた回数 a が商で、40 から 6 を a 回引いた残り b が余りということです。これはループ処理を使えば実現できそうですね。では実際に書いてみましょう。

まず、お決まりの部分ですね。

```
        org 100h
section .text
main:
```

計算する 2 つの値を用意します。40 は 16 進数で表すと、0x28。同様に 6 は 0x06 ですね。

```
main:
        mov ax, 0028h
        mov bx, 0006h
```

引き算をループするので、

```
divide:
        sub ax, bx
        jmp
```

このままでは無限ループですので、

```
divide:
        cmp ax, bx
        jc enddivide
        sub ax, bx
        jmp divide
enddivide:
```

おっと、このままでは商が得られませんね。商は cx に入れることにします。

```
divide:
        cmp ax, bx
        jc enddivide
        sub ax, bx
        inc cx
        jmp divide
enddivide:
```

割り算をするプログラムができました。

```
        org 100h
section .text
main:
        mov ax, 0028h
        mov bx, 0006h
divide:
        cmp ax, bx
        jc enddivide
        sub ax, bx
        inc cx
        jmp divide
enddivide:
```

が、割り算をするだけです。結果を見たい！（しかも、ＭＳ－ＤＯＳの終了処理してない）

となると、 結果を表示するプログラムを書かなければならないわけですが、 ここで問題が出てきます。 計算結果は ax と cx に入っているわけですが、それをポンと表示するわけにはいかないのです。なぜなら、ax と cx に入っているデータは数値であり、文字ではないからです。

結果を表示するために、数値を整数の文字列に変換するプログラムを考えてみましょう。 まず文字列としての数値は、例えば３１４１５９のように各桁に一文字ずつの数字で表されますから、 各桁の数値に分解しなければなりません。私たちが普段使っている数は１０進数ですので、数値を １０で割りながら、余りを求めていきます。すると、

３１４１５９÷１０＝３１４１５ … ９
３１４１５÷１０＝３１４１ … ５
３１４１÷１０＝３１４ … １
３１４÷１０＝３１ … ４
３１÷１０＝３ … １

このように、１０進数の各桁に分解することができますね。

とりあえず、プログラムを書き始めてみます。 まず、使いまわせるように、割り算を行うプログラムをサブルーチンにします。

```
divide:
        cmp ax, bx
        jc enddivide
        sub ax, bx
        inc cx
        jmp divide
enddivide:
        ret
```

整数を表示するサブルーチンを考えていきます。とりあえず、引数は dx で受け取ることにします。 演算に他のレジスタも使いたいので、まずレジスタの内容をスタックへ退避させておきます。

```
printint:
        push ax
        push bx
        push cx
        push dx
```

受け取った値を１０で割り算していくことを考えます。１０を１６進数で表すと 0x0A なので、

```
        mov ax, dx
        mov bx, 000Ah
        mov cx, 0000h
        call divide
```

CX には前の値が入っているので、０に初期化しておきます。 ２回目以降はループで処理していきます。何桁もの数値だった場合、レジスタだけでは保管場所が足りないので、スタックを使うことにします。スタックに何回 push したかを数えるために、dx を使うことにしました。

```
        mov cx, dx
        mov dx, 0000h
pintloop:
        mov ax, cx
        mov bx, 000Ah
        mov cx, 0000h
        call divide
        push ax
        inc dx
        jmp pintloop
```

このままでは無限ループになりますので、ループを抜ける命令が必要ですし、もしかしたら、一桁の 数値かもしれません。 cmp 命令で商が０かを調べるようにして、 命令を jnz 命令に置き換えます。

```
        mov cx, dx
        mov dx, 0000h
pintloop:
        mov ax, cx
        mov bx, 000Ah
        mov cx, 0000h
        call divide
        push ax
        inc dx
        cmp cx, 0000h
        jnz pintloop
```

最後に、数値を一文字ずつ表示していきますが、push した数値は、まだ数値のままであり、文字 ではありません。ＡＳＣＩＩという文字データの決まりでは、数字の０～９は１６進数の３０～３９ と決められているので、数値のデータに１６進数の 0x30 を足せばよさそうです。

1 文字出力する API 関数を使うため、DX にデータが入っていると困ります。CX であれば、もうこ れから使いませんし、loop 命令が使えるので便利です。ということで、DX の内容を CX に移すこと にします。

```
        jnz pintloop
        mov cx, dx
        mov ah, 02h
printic:
        pop dx
        add dx, 0030h
        int 21h
        loop printic
        pop dx
        pop cx
        pop bx
        pop ax
        ret
```

これで、数値を文字として表示するサブルーチンが完成しました！

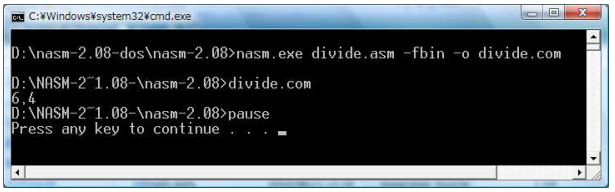
あとは、計算結果を表示させれば完成です。一度に商と余りを表示してしまうと、どこが境目か分からなくなりますので、商と余りの間に区切り記号を表示するようにしました。

今回作成したプログラムのリストは次の通りです。

```
        org 0100h
section .text
main:
        mov ax, 0080h
        mov bx, 0002h
        mov cx, 0000h
        call divide
        mov dx, cx
        call printint
        push ax
        mov ah, 02h
        mov dl, ','
        int 21h
        pop ax
        mov dx, ax
        call printint
        mov ax, 4C00h
        int 21h
divide:
        cmp ax, bx
        jc enddivide
        sub ax, bx
        inc cx
        jmp divide
enddivide:
        ret

printint:
        push ax
        push bx
        push cx
        push dx
        mov cx, dx
        mov dx, 0000h
pintloop:
        mov ax, cx
        mov bx, 000Ah
        mov cx, 0000h
        call divide
        push ax
        inc dx
        cmp cx, 0000h
        jnz pintloop
        mov cx, dx
        mov ah, 02h
printic:
        pop dx
        add dx, 0030h
        int 21h
        loop printic
        pop dx
        pop cx
        pop bx
        pop ax
        ret
```

実際にアセンブルして実行してみます。



16 BIOS コール

今までは、MS-DOS というオペレーティングシステム上で動作するプログラムを作成してきました。 では、オペレーティングシステムがなかったら、どうすればよいのでしょうか。

一般的な家庭向けコンピュータ（つまりパソコン）には、BIOS と呼ばれるプログラムが組み込まれています。BIOS というのは、必要最低限の入出力をつかさどるシステムで、コンピュータ本体やそれに接続された装置を管理しています。

BIOS は電源を投入した直後に瞬時に読み込まれ、実行されます。パソコンのスイッチを入れてすぐ、コンピュータに異常がないかチェックする画面が表示されますよね（最近はメーカーロゴの場合も多いですね。）あれです。

BIOS がコンピュータの様々な入出力を管理しているので、コンピュータの入出力は BIOS にお任せすればよいことになっています。

今回は、その BIOS の関数を使ってみようと思います。

本来、画面に何かを表示するには、1 ドットずつ地道に指定していかなければならないのですが、BIOS にはとりあえず文字を表示させる関数があります。

```
AH = 0E
AL = 出力する文字
```

としてから、int 10h を実行します。

とりあえずというのは、BIOS の関数は基本的に CPU が３２ビットで動作を始めると使えないからです。

今回は、前回作った Hello World プログラムを BIOS 用に改造することにします。

以下のリストは、以前作った文字列表示関数です。

```
print:
        push ax
        push bx
        push dx
        mov ah, 02h
putchar:
        mov dl, [ds:bx]
        cmp dl, '$'
        jz endchar
        int 21h
        inc bx
        jmp putchar
endchar:
        pop dx
        pop bx
        pop ax
        ret
```

これを BIOS の関数に対応させます。AH は 0E、AL に文字データ、BIOS 呼び出しの指示は int 10h ですから、このようになります。

```
print:
        push ax
        push bx
        push dx
        mov ah, 0Eh
putchar:
        mov al, [ds:bx]
        cmp al, '$'
        jz endchar
        int 10h
        inc bx
        jmp putchar
endchar:
        pop dx
        pop bx
        pop ax
        ret
```

今回は BIOS を使いますので、プログラムは１００番地からではなく、０番地から始めます。 DS、ES はセグメントレジスタで、いずれもデータの格納場所を指しています。07C0 を入れている のは、07C0 以降のセグメントにこのプログラム（ブートセクタ）が読み込まれるためです。 CPU にはプログラム終了などの命令はありませんので、最後を無限ループにしておきます。 ブートセクタは、５１２バイトである必要があり、最後の２バイトは 55 AA である必要があるので、 db 命令 (NASM に対しての命令) を使って０ (Null) で埋めてあります。 今回のソースのリストは以下の通りです。

```
        org 0
section .text
main:
        mov ax, 07C0h
        mov ds, ax
        mov es, ax
        mov bx, TEXTDATA
        call print
endprog:
        jmp endprog
print:
        push ax
        push bx
        push dx
        mov ah, 0Eh
putchar:
        mov al, [ds:bx]
        cmp al, '$'
        jz endchar
        int 10h
        inc bx
        jmp putchar
endchar:
        pop dx
        pop bx
        pop ax
        ret

section .data
TEXTDATA: db 'Hello, World',0Dh,0Ah, '$'

;Fill
       db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
       db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
       ～省略～
       db 0x00, 0x00, 0x00, 0x00, 0x00, 0x55, 0xAA
```

BIOS への対応、あっけなく終わってしまいました。というのも、BIOS がきちんと関数を用意して くれているからです。(BIOS がと言うより、BIOS を作った人ですね。ありがたや)

実際に動かしてみましょう！といっても、普段使っているパソコンで動かすと、どのように動作するか分かりませんので、 エミュレータを使うことにします。（どうしても実機で試したい！と言う方は、ハードディスクをはずしてからお試しください。その場合は自己責任でお願いします。 ）

x86CPU をエミュレートするエミュレータなら何でもいいのですが、 フリーで軽量でインストールも 不要な qemu という x86 エミュレータがありますので、今回はそれを使います。

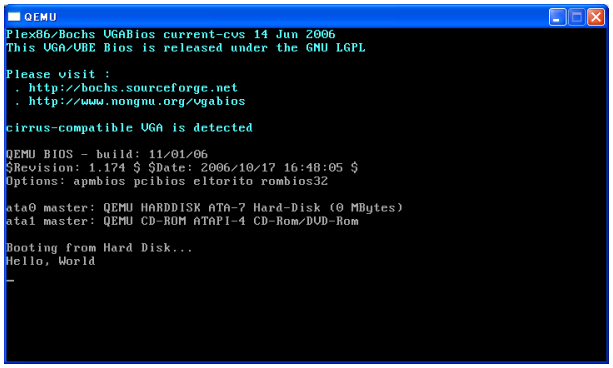
http://homepage3.nifty.com/takeda-toshiya/qemu/ からダウンロードしてください。

完成したプログラムをアセンブルし、

```
nasm.exe hellobios.asm -fbin -o hellobios.bin
```

実際に実行してみます。(コマンドは実際には一行です)

```
qemu.exe -L . -m 128 -hda hellobios.bin
           -soundhw all -localtime -M pc
```



17　さいごに

このテキストは、部会の勉強会で配布するために作成したものです。僕が「アセンブリ言語」に興味を持ったきっかけは、「コンピュータはなぜ動くのか」という本です。

この本を読んで、アセンブリ言語がコンピュータを学ぶ上で重要なものであることを知り、アセンブリ言語に関する本を探して実際に試しました。

アセンブリ言語によって、計算機 (コンピュータ) が計算機と呼ばれるゆえんが分かり、大変面白く感じました。この面白さを伝えたいと思い、部会で話をすることにしました。

実際に説明をしてみて、なぜそのようになるかという質問に答えられなかったり、間違いを指摘されたりと、自分の知識の足りなさを身をしみて感じました。自分の足りない部分を補うためにも、自分の知らない知識に触れるためにも、たくさんの本を読んで、いろいろなことに挑戦していきたいと思いました。

※このテキストにもたくさんの間違いが含まれていると思います。もし、この間違いにツッコんでくださる方は eserver_dip_jp@yahoo.co.jp までメールをいただけるとありがたいです。

最後まで読んでいただきありがとうございました！

参考文献

[1]『アセンブリ言語の教科書』 愛甲健二 データハウス ISBN4-88718-829-3

[2]『コンピュータはなぜ動くのか』 矢沢久雄 日経 BP ISBN4-8222-8165-5

[3]『IA-32 インテル・アーキテクチャ 32 ソフトウェア・デベロッパーズ・マニュアル』http://www.intel.co.jp/jp/download/index.htm

[4]『アセンブラ入門』http://www5c.biglobe.ne.jp/~ecb/assembler/assembler00.html

[5]『アセンブラ ～MS-DOS の世界～』http://www5c.biglobe.ne.jp/~ecb/assembler2/assembler10.html

[6]『Linux のアセンブラー GAS と NASM を比較する』http://www.ibm.com/developerworks/jp/linux/library/l-gas-nasm.html

[7]『The Netwide Assembler』http://www.nasm.us/

[8]『QEMU on Windows』http://www.h7.dion.ne.jp/~qemu-win/index-ja.html

OCaml 入門

fujii

fujii@mma.club.uec.ac.jp

概要

機会[*1]があって勉強中の OCaml について書いてみる。最終的には高階関数を使って数列を作ってみたり。

1 OCaml とは

OCaml(Objective Caml) は、フランス国立情報学自動制御研究所 (INRIA) で開発された関数型言語で、ML[*2]の方言にあたります。処理系は Windows や Mac OS X なら `http://caml.inria.fr/index.en.html` から入手できます。Unix-like な環境なら、直接入手するよりもパッケージ管理システムを用いた方がよいでしょう。

2 とりあえず使ってみる - 演算、型推論

`ocaml` コマンドを実行するとインタプリタが起動します。まずはお決まりの hello world から:

```
> ocaml
        Objective Caml version 3.12.0

# print_endline "hello, world";;
hello, world
- : unit = ()
```

#がインタプリタのプロンプトです。何か適当な数式を入力してみます。

```
# 3 + 4;;
- : int = 7
```

OCaml インタプリタでは、セミコロン 2 つで入力の一区切りを示します。さて、入力すると次の行に評価された式の型と値 (この場合 `int` 型 (整数) の 7) が表示されます。変数も使ってみます。変数に値を束縛[*3]するには `let` を使います:[*4]

```
# let a = 4;;
val a : int = 4
# let pi = 3.14;;
val pi : float = 3.14
```

型についての情報を何も入力していないにも関わらず `a` の型は `int`、`pi` の型は `float`(浮動小数点数) であると解釈されています。OCaml には型推論の機能があり、型を明示的に書く必要はありません。[*5]では、変数を使って適当に計算をしてみます。

```
# pi * a * a;;

Error: This expression has type float but an expression was expected of type
         int
```

エラーが出ました。「この式 (`pi`) は `float` 型だが、期待される式の型は `int` である」といったところでしょうか。OCaml では

[*1] 情報工学工房。情報理工学部 情報・通信工学科 (I 科) の選択科目のひとつ
[*2] Meta-Language、イギリスのエディンバラ大学で開発された関数型言語
[*3] 変数に対して値を対応付けること
[*4] 変数名や関数名は小文字またはアンダースコアで始まる必要があります。
[*5] 明示的に書くこともできます

演算子の取る値の型が厳格に定まっていて、演算子*は int 型の値 2 つを取ります。つまり*では float を使った掛け算はできないということです。float の掛け算には演算子*. を用います。では気を取り直して…

```
# pi *. a *. a;;

Error: This expression has type int but an expression was expected of type
         float
```

…OCaml では暗黙の型変換が行われません。すなわち、同じ float 型どうしでなければ*. で掛け算をすることはできません。int 型の値を float 型の値に明示的に変換するには float_of_int を使います:[*6]

```
# pi *. float_of_int a *. float_of_int a;;
- : float = 50.24
```

ようやく計算できました、良かった良かった。ちなみに、*に対する*. と同様に、+. や-.、/. があります。

3 関数定義 - 部分適用

関数定義も let による束縛で行います。

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 5 (-3);;
- : int = 2
```

int 型の値 2 つを受け取り、それらを加えた結果を返す関数 add を定義しています。int -> int -> int という表記が「int 型の値を 2 つ受け取り、int 型の値を返す」ことを表しています。負の数を渡すときは () でくくる必要があります。

また、関数を部分的に適用することができます:

```
# add 5;;
- : int -> int = <fun>
```

型を見ると int -> int、int を受け取って int を返す関数となっています。これを新たに関数として定義してみます。「int 型の値を一つ受け取って 5 を加える関数」add5 です。

```
# let add5 = add 5;;
val add5 : int -> int = <fun>
# add5 (-3);;
- : int = 2
```

4 リスト

リスト (list) は同じ型の値を並べたものです。次のように再帰的に定義されます:

- 空リスト [] はリストである
- l がリストなら、l の先頭に要素 e を追加したもの e :: l もリストである

実際にリストを作ってみます。

[*6] 逆の int_of_float など、なんとか_of_なんとかの形の関数がたくさんあります

```
# let l = 1 :: 2 :: 3 :: [];;
val l : int list = [1; 2; 3]
# let ll = [[1; 2]; [3; 4]; [5; 6]];;
val ll : int list list = [[1; 2]; [3; 4]; [5; 6]]
```

[a; b; c; d] は a :: b :: c :: d :: [] の略記法です。リストのリストも作れます。

5 パターンマッチ

リストを用いて複数のデータを扱ってみます。まずはリストを受け取りその要素数を返す関数 length を考えます。リストを扱うにはリストから値を取り出す必要があります。それにはパターンマッチを用います。

```
let f lst = match lst with
     パターン 1 -> 式
   | パターン 2 -> 式
   | ...
```

このように、パターンを|で区切って書きます。実際に例を見た方が早いでしょう:

```
# let rec length lst = match lst with
      [] -> 0
    | x :: xs -> 1 + length xs;;
val length : 'a list -> int = <fun>
```

自分自身を呼び出す再帰関数を定義する際は、let の後に rec と書く必要があります。この例では、lst が空リストなら 0 を返し、x :: xs、つまりある値 (ここでは x) の後ろにリスト (ここでは xs) がつながったものなら xs の長さに 1 を加えたものを返しています。

パターンには定数を直接書いたり、ワイルドカード (_) を使うこともできます。例として、0 から 6 の数を受け取り対応する曜日の名前を返す関数 weekday_name を書いてみます。

```
# let weekday_name n = match n with
      0 -> "Sunday"   | 1 -> "Monday" | 2 -> "Tuesday"  | 3 -> "Wednesday"
    | 4 -> "Thursday" | 5 -> "Friday" | 6 -> "Saturday" | _ -> "Wrong number"
val weekday_name : int -> string = <fun>
# weekday_name 3;;
- : string = "Wednesday"
# weekday_name 256;;
- : string = "Wrong number"
```

パターンマッチは書かれた順に上から行われるので、0 から 6 のどれにもマッチしなかった場合のみワイルドカードパターンにマッチします。

6 高階関数 - 匿名関数

関数型言語である OCaml では、関数を引数に取ったり関数を返す関数、高階関数を定義できます。ここでは、「リストと、「リストの要素 1 つを受け取って演算を施した結果を返す関数」とを受け取り、リストの全ての要素に関数を適用したリストを返す」[*7]関数 map を定義してみます。

[*7] 日本語が意味不明ですね…

```
# let rec map f lst = match lst with
      [] -> []
    | x :: xs -> (f x) :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map weekday_name [0; 1; 2; 3; 4; 5; 6];;
- : string list =
["Sunday"; "Monday"; "Tuesday"; "Wednesday"; "Thursday"; "Friday"; "Saturday"]
```

先ほど定義した weekday_name を使ってみました。さて、関数に関数を渡す際、その場で名前のない関数 (匿名関数) を定義して渡すこともできます。匿名関数は"fun 引数 -> 式"という形で定義することができます。

```
# map (fun x -> x * 2) [0; 1; 2; 3; 4];;
- : int list = [0; 2; 4; 6; 8]
```

匿名関数はそのまま適用したり、束縛することもできます。

```
# (fun x y -> x * y) 2 4;;
- : int = 8
# let square = fun x -> x * x;;
val square : int -> int = <fun>
```

最後に、高階関数を用いた数列の生成をやってみます。「整数を受け取って対応する数列の項を返す」関数 f と整数 n を受け取り、第 1 項から第 n 項までのリストを返す関数 sequence を定義してみます。

```
# let sequence f n =
    let rec sub_seq f i n =
        if i > n then [] else (f i) :: sub_seq f (i + 1) n
    in sub_seq f 1 n;;
val sequence : (int -> 'a) -> int -> 'a list = <fun>
# sequence (fun x -> 2. ** float_of_int x) 10;;
- : float list = [2.; 4.; 8.; 16.; 32.; 64.; 128.; 256.; 512.; 1024.]
# sequence (fun x -> 2 * x - 1) 10;;
- : int list = [1; 3; 5; 7; 9; 11; 13; 15; 17; 19]
```

"let 関数名 引数 = 式 1 in 式 2"という形式で、式 2 の中でのみ有効な局所関数が定義できます。[*8]sub_seq は第 i 項から第 n 項までのリストを再帰的に生成する局所関数で、これを用いて sequence を定義しました。使用例として、2 のべき乗数と奇数のリストを作ってみました。

7 おわりに

以上、初学者がお送りする入門編でした。この記事を書くこと自体が、むしろ自分の勉強になった感があります。本稿で触れられたのは OCaml という言語のごく一部に過ぎません。今後もっと勉強して行きたいと思います…

参考文献

[1] 浅井　健一、プログラミングの基礎、サイエンス社、2007 年

[2] 五十嵐　淳、プログラミング in OCaml、技術評論社、2007 年

[3]「チュートリアル - OCaml.jp」 http://ocaml.jp/チュートリアル

[*8] 同様に、"let 変数名 = 式 1 in 式 2"という形式で局所変数も定義できます。

Haskell 入門

kyogoku42

kyogoku42@mma.club.uec.ac.jp

概要

Haskell という言語がある．**関数型**の，値の計算に**遅延評価**を採用している言語である．ここでは 関数型であること・遅延評価について，簡単な紹介を行う．

1　関数型の言語とはなんだろうか.

関数型の言語とは，(その名のとおり) 関数を中心にそえた言語である．**手続き型**　のプログラムと**関数型**のプログラムの例をあげ，対比を行おう．

いま「クイックソート」と呼ばれるデータを整列するアルゴリズムを例にとる．

ここではアルゴリズムの詳細には立ち入らない．知る必要のあることは次のことだ:

1. まず Quicksort は，うけとった配列 A のなかから適当な要素 P を抜き出す.
2. P をのぞいた A の要素を，P より小さいもの S と P と等しいか大きいもの L にわける.
3. 結果は Quicksort(S) ++ P ++ Quicksort(L) である.[*1]

1.1　手続き型の言語

これを，手続き型の言語で記述すると，以下のようになる.[*2]

```
Quicksort(A)
        if Length(A) > 1 then
                S ← []
                L ← []
                for i ← 1 to Length(A) do
                        if A[i] < A[0] then
                                S ← S ++ [A[i]]
                        else if A[i] ≧ A[0] then
                                L ← L ++ [A[i]]
                A ← Quicksort(S) ++ A[0] ++ Quicksort(L)
        return A
```

[*1] ここで，S と L に対して再帰的に関数を適用していることに注意.

[*2] 言語ごとの細かな差違を潰し公平を期すために，よくある記法に従った疑似言語で記述する.

これに [3,4,2,1,5] をあたえた場合を考えよう．手続き型言語では，変数に対し操作を行うことにより，プログラムの処理が進行する．

```
Quicksort([3,4,2,1,5])
S ← [2,1]
L ← [4,5]
        Quicksort([2,1])
                S ← [1]
                L ← []
                Quicksort([1])
                        S ← []
                        L ← []
                        結果 ← Quicksort([]) ++ [1] ++ Quicksort([]) = [1]
                結果 ← Quicksort([1]) ++ [2] ++ Quicksort([]) = [1,2]
        Quicksort([4,5])
                S ← []
                L ← [5]
                Quicksort([5])
                        S ← []
                        L ← []
                        結果 ← Quicksort([]) ++ [5] ++ Quicksort([]) = [5]
                結果 ← Quicksort([]) ++ [4] ++ Quicksort([5]) = [4,5]
結果 ← Quicksort([2,1]) ++ [3] ++ Quicksort([4,5]) = [1,2] ++ [3] ++ [4,5] = [1,2,3,4,5]
```

1.2 関数型の言語

これに対し，関数型の言語 (今回は Haskell) では，以下のように記述される

```
quicksort []     = []
quicksort (x:xs) = quicksort smaller ++ [x] ++ quicksort larger
        where
                smaller = [a | a ← xs, a < x]
                larger  = [b | b ← xs, b ≧ x]
```

同様に [3,4,2,1,5] をあたえた場合を考えよう．

関数型の言語における処理は，関数に引数を代入していくことにより進行する．ここでいう関数とは，数学でいう関数と同じで，引数をあたえると返値が一意に定まるものを指す．(ここで，quicksort [ ] = [ ] であることは定義である．)

```
qsort [3,4,2,1,5]
= qsort [2,1] ++ [3] ++ qsort [4,5]
= qsort [1] ++ [2] ++ qsort [] ++ [3] ++ qsort [] ++ [4] ++ qsort [5]
= [1] ++ [2] ++ [3] ++ [4] ++ [5]
= [1,2,3,4,5]
```

2　関数型である利点

手続き型の言語では，すべての計算はその名の通り**手続き的**に実行され，手続きの順番が計算結果に影響する．この結果，すべての値は，関数に渡される前に完全に評価されなければならない．

一方，(純粋な) 関数型言語では，計算は**決定論的**に進行する．関数は同じ値を渡せばかならず同じ結果を返し，その結果は（停止性についての問題をのぞけば）一般に計算順序によらない．このことは，関数に渡す値が（関数に渡された時点で）完全に評価されている必要がないことを示す．

3　遅延評価

関数の値が計算順序によらず一意に決まることにより，無限のデータを扱う，といったことが可能になる．必要になり次第，必要になったところのみを計算することが可能だからだ．これを**遅延評価**という．以下の例では，無限の素数列を扱う．

```
Prelude> let factors n = [x | x <- [1..n], n `mod` x == 0]
Prelude> let isprime n = factors n == [1,n]
Prelude> let primes = [n | n <- [1..], isprime n == True]
```

これは，無限の素数列を生成するコードである．このうえで，たとえば 100 より小さい素数をすべて列挙するには，こうしたらよい:

```
Prelude> takeWhile ((>=) 100) primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

4　注意

Haskell を使う場合，GHCi を使うことになるだろう．

http://hackage.haskell.org/platform/

注意したいのは，インタプリタである GHCi 上では，関数定義は **let** を頭に記さなければならないことだ．
ほか細かな注意としては，以上に示したコード中の特殊文字を若干置き換えなければならないことか．以下に置き換えの一部を示す．not equal が /= であることに注意したい．

置き換え前	後
→	`->`
←	`<-`
≠	`/=`
≧	`>=`
≦	`<=`
λ	`\`
¬ (否定)	`not`
∘(合成)	`.`

置き換え表

Let's note CF-W2 の HDD 交換

wakisaka
wakisaka@mma.club.uec.ac.jp

概要
交換が困難とされている Let's note の HDD 交換に挑戦した時の記録です。

1 Let's note CF-W2 とは

まずは基本性能と特徴ですが、CF-W2 は 2003 年発売の B5 型ノート PC です。上にパカっと開く DVD-ROM ドライブが見た目 的な特徴です。CPU は Pentium M の 1.0GHz、メモリ 256MB、HDD40GB と今となっては微妙な性能で標準の Windows XP SP1A から SP3 にすると大変重く使いづらいのですが、Windows を入れなければどうということはない、Fedora13 をインストールして割と快適に使っていました。[*1]メモリを増設[*2]して 768MB にし、今でもやはり苦労する無線 LAN を設定して快適に使えるようになった矢先に事件は起こりました。

2 IO error

Windows 時代はけっこうな確率で BSoD[*3]を出していたこの PC ですが、Fedora にしたら安定して動いていたので油断していたところ、SMART が不良セクタの増加を検知。それでもまだだましだまし使っていましたがちょうど手を置く位置の下にある HDD から異常な振動、音がしはじめ、端末エミュレータが IO error と表示してフリーズ。とりあえずスイッチを 4 秒押しして強制終了、その後は何事も無く動いたのでとりあえず自宅のサーバにデータを全て退避し、クラッシュ覚悟で無理矢理使う事にしました。ついに死亡したのは数週間後の 10/8(金)、MMA の部会での軽い発表を行う日でした。発表直前から 2,3 度異常振動を起こしては強制終了を繰り返していましたが、発表中は無事に動き、終わった瞬間にまた異常振動。さすがにもう無理だろうということで、交換に踏み切りました。

3 交換の問題点

さて、交換をためらっていた理由です。普通、ノート PC は裏側のネジ 2,3 本を外して蓋をあけると普通の 2.5 インチ HDD がすぐに出てきて交換可能な構造になっています。ところが、小型化と省電力化を追い求めすぎた結果、Let's note シリーズは全般的に HDD が大変取り出しづらくほぼ全分解が必要な上、中の HDD は 2.5 インチではありますが普通より低電圧で駆動する仕組みになっています。HDD が不調なのは分かっていたため事前調査したところ、先人の方々はこの電圧の違いをピン折りによって無理矢理解消し、大容量の HDD への換装を行っていました。そんな恐ろしいことできるかと一度は諦めていましたが壊れてしまってはやむを得ないので交換に挑戦することにしました。

4 dry-run

交換するにあたってはまず、「分解できる」「組み立てられる」の条件が必須となります。10/11(月) にとりあえず実際の交換を伴わない分解組立テストを行いました。いくつかのサイトを見て回った結果、ポイントは以下の 4 点に絞られました。

- スライドスイッチが飛び出さないよう、開けた状態でガムテープで固定する
- DVD-ROM ドライブが閉じないように気をつける
- ネジの位置が分からなくならないようメモをとりながら
- キーボードに付いているグリス[*4]に注意

[*1] 実はメモリ足りないと言われテキストインストールになってしまい、後付で X を入れました
[*2] このメモリも MicroDIMM DDR とあまり無いもの
[*3] コードを読むとメモリ関連。swap の異常と気づいたのは後の話でした。
[*4] CPU はキーボード経由で放熱します。

指示に従いスイッチを固定、裏返していざ分解。DVD-ROM ドライブを開け忘れたことに気づいて後から穴をつついて開けたりしつつネジを外していきます。ネジがとにかく硬い。細かい。持ち上げづらい。画面やキーボードを壊さないように慎重にやりつつもかなりの力を必要とする作業でした。さて、裏側のネジをほぼ全部 (10 本程度) 外したら表に戻してキーボードを剥がします。薄っぺらいキーボードがツメ 2 箇所とネジ 3 本、両面テープで固定されているのでまずツメを外してから両面テープをバリバリと剥がしていきます。キーボードの上側の固定が外れて手前に返せるようになるので本体とキーを繋ぐコネクタを外して完全に取り去ってしまいます。キーボードが外れるとさらに大量のネジが見えます。いくつか外す必要のないネジがあるので見極めつつ結局ほぼ全て外します。ここで問題発生。裏側のネジを全て外してしまったため画面のヒンジを固定しているネジがなくなり、画面が開いてしまい左と右のヒンジの角度が違う状態になってしまいました。無理矢理直して本番は裏面のヒンジ固定ネジを外すのは最後にすることにしました。どうせキーボードの下のネジを取った後はもう一度裏返すので裏側のネジを残しておいても問題ありません。表のネジを外すにしたがってだんだん本体上部が浮き上がってきます。しかし、表で固定していると思われるネジを全て外しても完全には外れません。どうやらまだ固定されている模様。正体は表と裏だけ外してもまだ甘いと言わんばかりにある側面の固定。まず、無線アンテナ部分の蓋が上下を外れないように固定しているのでかなり強引に外します。次に、電源ユニットと思われるあたりに付いているネジ付きの蓋。これの正体はわかりませんがとにかく外します。これで思い当たる部分は全て外したので開封！と行きたいところですがなぜかまだ外れませんでした。なんと、画面コネクタの左右にあるケーブル固定用ネジまでもが上下の固定に使われていました。これを外すとついに上下が離れ、HDD が姿を現します。とりあえず実験分解はここまでですが、元にもどすまでが HDD 交換です。ここまでの作業を全て逆回しで行うことになります。無線アンテナ部分の蓋は開けるのが面倒だったので開けっ放しにしておくことにしました。どうせ翌日には HDD 交換本番で開け直すことになります。動作確認には影響はありません。ここでキーボードのコネクタが一時的に外れてしまい、後々に影響を及ぼすことになりますがそれは次のセクションで。

5　本番

分解してから戻して動くことが確認できたので実際の交換作業に入ります。HDD は動作実績の挙がっていた HTS421210H9AT00(100GB,4200rpm) を購入しました。[*5]予行の手順にしたがって、また改善点を活かしつつ順番に分解していきます。二度目なのでどこを外すかは完全に分かっているのでさっさと分解して HDD を取り出せる段階まで持っていってしまいます。新しい HDD の準備としてまずコネクタのピンを折る作業を行う必要があります。写真のようにピンを 2 本[*6]を折って電源供給を遮断します。

旧 HDD を取り出してこの HDD を装着。裏返して基板側を上にし、写真にあるよく分からないシールを旧 HDD から剥がして付け直してセットします。ケーブルを上下に気をつけつつ接続し、やはり逆回しの手順で上下をくっつけます。認識しなくて開け直しという事態になった時のために無線アンテナの蓋と画面ケーブルソケットのネジはそのまま外し、バッテリーを取り付けて通電。BIOS に入って認識しているか確認します。プライマリ マスター:100GB の表示を確認してほっと一息。と思いきや左キーが効かなくなってしまいました。何度か叩くと効くのでとりあえず放置して OS のインストール作業に入りました。OS は

[*5] SSD は無線と電波的に干渉するらしいので見送りました
[*6] 41 番と 44 番ピン

Fedora13 を入れ直します。自宅サーバにネットワークインストール用の設定がされているのでインストール元を設定しようとしたところ、またキーボードを認識しない状態に。どうしようもないのでもう一度開けてキーボード部分を取り外し、コネクタを見てみると若干斜めに入っているような状態になっていました。キーボードを入れてしまうとコネクタ部分は見えなくなってしまうので、確認のしようがありませんが力づくで押し込んだところカチッ[*7]という音を立ててはまり、キーボードを正常に認識するようになりました。動いたところでもう一度 Fedora13 のインストールを行って無事に全ての作業が完了しました。ディスクに余裕が出来、メモリも増えたので WindowsXP とのデュアルブートも可能でしたが、学内ではどう考えても Windows を使う用事がなかったのと設定が面倒なので放棄してやっぱり Fedora 一本で使うことにしました。

6　これから挑戦する人に

あまり居ないとは思いますが参考程度に。前述の注意点はもちろん、次のことにも注意した方がいいと思います。

- 紙に図を描き、ネジを図に配置していく
- ヒンジを固定しているネジを外すのは最後
- キーボードの両面テープが汚れないように

一番最後の両面テープは超重要です。張り付きが悪くなるとふかふか浮き上がって非常に叩き心地の悪いキーボードになります。

7　後日談

両面テープをなんども付け外しした影響か、キーボードがふかふかする状態になってしまっていますがまあ問題なく動いているのでいいとします。データは全て[*8]自宅サーバにバックアップしてあったので必要なものを拾い上げてコピーして終了。バックアップの重要性、特にクラッシュ後の復元の早さを再認識しました。旧 HDD は釘抜きとトンカチで叩き割ってプラッタを粉々にした上で燃えないゴミへ。ジャンク市に混ぜたりしてませんよ。自宅サーバの更新に先立ってのアップグレード実験で Fedora14 になったりはしましたが今日も元気に動いています。

ドライブ

モデル名:	ATA HTS421210H9AT00	シリアルナンバー:	HKA69CARCKA8PM
ファームウェアバージョン:	HABOA750	World Wide Name:	0x5000cca514c7e0e6
位置:	PATA ホストアダプタ の ポート 1	デバイス:	/dev/sda
書き込みキャッシュ:	有効	回転速度:	–
容量:	100 GB (100,030,242,816 バイト)	接続:	ATA
パーティション:	マスターブートレコード	SMART 状態:	● ディスクは正常です

*7 というよりはゴリッ

*8 本当に HDD 全体を rsync で。/dev や/proc まで間違ってコピーしていました

残念な目的の為の自作 PC

shimazaki
shimazaki@mma.club.uec.ac.jp

概要
一人暮らしの部屋にテレビと録画機を両方揃えるのは値段が高く、TV チューナーを搭載した PC を自作した方が遥かに安上がりであり、液晶も大型化できる。また、PC であるからもちろん、ネットサーフィンも可能でありレポートも書ける。今回はなるべく安く、だけど高性能に、TV 以外でも使える、ということを目標においた。
なお、当初の予算は 10 万円に設定してある。

1 PC 構成の選定

自作するにあたって、一番重要なのはパーツ構成を考えることである。持論を展開するならこの作業が自作の醍醐味であり一番楽しいと思っている。

1.1 CPU

まず選択したのは CPU である。そもそも、一般に家庭で使われるパソコンに使われる CPU は INTEL 社と AMD 社での寡占化が凄まじく、基本的にこの 2 社のうちから選択することになる。当然、対応するソケットが違う訳で、この選択によって他のパーツ選定も大まかな方向性が決まる。そもそもの目的が「なるべく安く」ということもあり、今回は迷うことなく AMD 社を選択した。同社は価格性能比に優れていることで有名である。また、最近ではクアッドコアが増えてきたが、いかんせん予算が少ない。そこで BIOS 設定でクアッドコア化も狙える「Phenom II X2 555 Black Edition」を選択した。

1.2 マザーボード

次にマザーボードの選定である。ここは「固体コンデンサを使っている」「ヒートパイプが付いている」「サイドポートメモリが付いている」という項目を主に重視した。AMD 社の CPU を採用するマザーボードの多くは優秀なオンボードグラフィックスを搭載している。しかし VRAM はメインメモリから使用し、ユーザーが使用できる領域は減ってしまう。これが今回サイドポートメモリにこだわった理由である。このことから、今回は MSI 社の「785GM-E65」を採用した。このマザーには SATA ケーブルが 1 本しか付属しないため、 1 台目として組もうとすると光学ドライブと HDD が同時に接続できず詰むことになるので要注意である。

1.3 メモリ

そして、メモリの選定であるが、ここは個人的なこだわりで Corsair 社のメモリを使うと以前から決めていた。今回使用したのは 2GB の DDR3 を 2 枚、「TW3X4G1333C9A」を使用した。ヒートシンクが付いており、比較的安価に、かつ信頼性のいいものだと判断した。自作当時（2010 年 3 月）は 1 万円を優に超えていたものだが、現在（2010 年 11 月）では半額以下となっている。「欲しいときが買い時」などと言われるが、値段を見てしまうと悲しくなる。

1.4 HDD

HDD であるが、今回の本来の目的が録画であることからブート用と録画用の 2 台構成とした。なお、当時は何故か WD 信仰者であった為 2 台とも WD 社製のものが採用されている。また、ブート用 HDD はその用途から少し奮発して 7200rpm のものである。型番はブート用、録画用それぞれ「WD5000AAKS」、「WD10EARS」である。これもメモリ同様、自作当時の 6 割程度まで値段が落ち込んでいる。値段を調べつつ執筆しているが、正直泣きたくなってくる。

1.5 光学ドライブ

光学ドライブだが、ここは完全に割り切った。というか全く考えずにスーパーマルチドライブで最安値のものを選択した。将来的に BD ドライブへのグレードアップを考えており、OS インストール以外は使い道がなかったためである。

1.6　ケース

PC ケースだが、これも事前に決めてあった。ANTEC 製の「Three Hundred」である。これも録画用、もっと言えば主に深夜、寝ている間に起動していることを考えて冷却重視にしてある。

1.7　チューナー

肝心のテレビチューナーだが、今回はピクセラ社製の「PIX-DT096-PE0」を採用している。というのも、多くのレビューサイトには他社製のチューナボードでは録画失敗の確率が高いという評価が多く見られた一方、ピクセラ社の製品は録画失敗が少ないと評価が高かったためである。事実、録画に失敗したのは半年間で 1 回程度であった。また、予算が少ないといいつつ、W チューナーを採用したのは深夜アニメで被った場合のことを考えてのことである。なお、今回のパーツ選定に当たってはもちろんながら深夜アニメの録画を念頭に置いている。

2　各パーツの組み立て

PC の組み立ての関しては、近年自作そのもののハードルが下がり、容易なために割愛する。

3　インストール後の設定

インストールした OS は windows7 の 6 4 ビット版である。いきなり Linux を…と言いたいところだが、入学前のことであり、そこら辺には疎かった、また、TV チューナーの対応からして選択肢にはなかった。

インストール直後に BIOS 設定を弄り、さっそくながら 4 コア化にチャレンジした。……のだが、これがあっさり終了し起動後確認するとクアッドコアとして認識。B55 プロセッサーという表記となり、CPU ファンの自動制御が効かなくなるなどの現象が起こるが、既知のことなので問題はない。ただ、コア温度の情報が取得できなくなることが痛手か。また、こうも簡単に 4 コア化ができてしまうと、欲を出してオーバークロックなんてものに挑戦してみたくなる。あれ、本来の目的は録画機として安定動作することじゃなかったっけ?

今回は BIOS 上から設定するのではなく、「K10stat」というソフトウェアを用いてクロック周りを弄った。このソフトウェアはクロック周波数、コア電圧、クロック変化のタイミングなどを負荷に応じて 4 段階に切り替えられる。録画機として比較的長時間起動しているものであるから、消費電力はなるたけ抑えたい。ただし 4 コア化している時点で消費電力が上昇しているという指摘に関しては聞かなかったということにしておく。したがって、低電圧低クロック、低電圧高クロックという両方に挑戦するというなかなか楽しい設定となった。これにより、低負荷時はクロックが 800Mhz まで低下し、高負荷時は最大で 3.6GHz まで上昇する。なお、電圧を盛ってやると 3.8GHz まで起動することには成功したが、不安定であった為に 3.6GHz で常用することにした。

4　拡張、そして何所へ向かうのか

組み立てから約 1 か月。4 コア化やオーバークロックといった運用にも関わらず安定動作していた矢先、事件は起こった。

「録画番組がひどい横縞……なにこれインターレース?」

このころはまだ録画機としての役割の比率が高かったため、とても気になる症状ではあった。もちろん真っ先に調べる。グーグル先生に聞いてみる。

……原因判明。

どうやら TV チューナーと RADEON との相性らしい。GeForce 製のグラフィックボードに変更すれば改善するという報告から急遽秋葉原へ。そして帰りには何故か 1 万円ほどのグラボが。搭載チップは GTS250。録画という用途だけには無駄すぎる。この頃から大幅な拡張に走り始めた。

組み立てから半年にもかかわらず、録画用 HDD が埋まってしまった。どうしてこうなった……。TS も抜けなければ消去することも絶対にしたくないので HDD 増設の方向で調整へ。そして再びの秋葉原。……気づいたら今度は 1.5TB の HDD が手元にあった。もちろん WD 製。今年 3 台目の WD である。2TB じゃないのは GB 単価が安かったからである。それにしても HDD

安くなったものだ。この時点で HDD の総容量は 3TB。うむ、しばらくは大丈夫だろう。あと半年くらいは。
この後に BD ドライブの増設もしたが、これは本来予定されていた拡張である。秋葉原を散策中に 1 万円を切る BD ドライブを発見。並行輸入品だったり、キャッシュの表示に不穏な文字が見えたがどうせ BD の視聴だけなので気にしないことに。

そして今に至る。これでしばらくは戦える…はず。

5 まとめ

- 自作は予算と用途！
- 買いたい時が買い時！
- 残念だっていいじゃない！

PrivateIP しか振られていない自宅に外部からアクセス

sawada
sawada@mma.club.uec.ac.jp

概要
諸事情により ISP からクローバル IPv4 が自宅に振られていないという場合に、なんとかして外部から自宅にアクセスしようというお話です。

1 はじめに

みなさんの自宅には ISP からグローバル IPv4 が割振られているでしょうか?マンションなどの集合住宅の共同回線を利用している場合は自宅にプライベート IPv4 しか割り振られていない場合があると思います。私の自宅の場合がもろにその状況です。この場合 IPv6 が使えたらまだ良いのですが、私の場合は v6 も使えない状況でした。これではちょっと外出した際に自宅のコンピュータにアクセスできないので不便だったりします。しかし、筑波大学の「グローバル・固定 IPv6 アドレス割当型トンネル接続実験サービス」と言うものを利用すると、自宅に IPv6 でアクセスできる環境が出来上がります。今回は基本的な導入方法と私がどのように利用しているかを紹介したいと思います。

2 どんなサービス？

「グローバル・固定 IPv6 アドレス割当型トンネル接続実験サービス (http://v6ip.tsukuba.wide.ad.jp)」は学術目的で公開されているサービスです。「PacketiX VPN」というソフトによって面倒な設定をすることなく IPv6 が使える環境を構築できます。とても大雑把に説明しますと、VPN で筑波大のサーバに接続してローカルの仮想ネットワークインターフェイスカードに IPv6 を振ってもらうことで、筑波大のサーバ、回線を経由して IPv6 接続を実現します。

3 メリット・デメリット

- メリット
  (1) 外部から自宅にアクセスが不可能な環境でも IPv6 さえ使えれば世界中どこからでも自宅にアクセス可能。
  (2) 自宅 PC で IPv6 が使えるようになる。(例えば IPv6 しか振られていない MMA のサーバに直接アクセスが可能)
  (3) サービスは無料で使える。
- デメリット
  (1) 試験サービスなので若干不安定。サービス自体がいつまで続くか分からない。
  (2) 迂回して接続するので速度が劣る。
  (3) 学術目的のサービスなので一人でリソースを食うようなことをすると迷惑がかかる。

と、若干問題もありますが今までアクセス不能だった自宅にアクセスできるのは大きいです。試してみる価値はあると思います。

4 導入

では早速導入していきましょう。今回は OS に Ubuntu(Debian) を使って説明します。お約束ですが、**導入・使用は自己責任でお願いします。**

4.1 sshd 設定

自宅にアクセスするのにはまず通常は SSH を利用するでしょう。ということで SSH を使えるようにします。

```
$ sudo apt-get install openssh-server
```

インストールが終われば SSH でログインが出来るようになります。初期設定だと root ログインが有効になっているので無効にします。適当なエディタで`/etc/ssh/sshd_config` を開き、`PermitRootLogin yes` を `no` にします。ただしこのファイルは

スーパーユーザでないと編集できません、sudo コマンドを使って編集してください。終わったら sshd を再起動させます。

```
$ sudo /etc/init.d/ssh restart
```

試しに ssh でログインしてみましょう。

```
$ ssh localhost
```

ログインできましたか?

4.2 PacketiX VPN Client 導入

残念ながら Linux 用の PacketiX VPN Client には GUI がありません。ですが一度設定してしまえば後は簡単です。

まずはサービスにユーザ登録をしましょう。サービスのトップページ左のメニューの「ユーザ登録」から指示にしたがって登録してください。登録が終わったら接続情報の書かれたメールが届くので大切に保管してください。

次に SoftEther から PacketiX VPN Client をダウンロードします。PacketiX VPN Client 2.0 を選択し、環境にあったアーキテクチャを選択してダウンロードします。ここでは Linux 32bit (x86) を使用して説明します。ダウンロードしたものをここでは /src フォルダを home 直下に作成して中に入れておきます。ではダウンロードしたファイルを展開します。

```
$ cd src
$ tar -xf vpnclient-5280-rtm-linux-x86.tar.gz
```

次にコンパイルに必要なライブラリをインストールします。

```
$ sudo apt-get install zlib1g-dev
$ sudo apt-get install libreadline5-dev
```

コンパイルします。次のように入力し、指示に従います。

```
$ ./.install.sh
```

最後にディレクトリを移動してインストールは完了です。

```
$ sudo mv ../vpnclient /usr/local/vpnclient
```

4.3 PacketiX VPN Client 設定

PacketiX VPN Client を起動して設定をします。

```
$ sudo /usr/local/vpnclient/vpnclient start
$ /usr/local/vpnclient/vpncmd
```

「2. VPN Client の管理」を選択します。次に localhost に接続するので何も入力せずに Enter を押します。設定はコマンドを入力して次の様に行います。

```
VPN Client>NicCreate
NicCreate コマンド - 新規仮想 LAN カードの作成
仮想 LAN カードの名前: tlan

VPN Client>AccountCreate
AccountCreate コマンド - 新しい接続設定の作成
接続設定の名前: tsukuba
接続先 VPN Server のホスト名とポート番号: v6ip.tsukuba.wide.ad.jp:443
接続先仮想 HUB 名: ACVPN
接続するユーザー名:（登録したユーザ名）
使用する仮想 LAN カード名: tlan

VPN Client>AccountPasswordSet
AccountPasswordSet コマンド - 接続設定のユーザー認証の種類をパスワード認証に設定
接続設定の名前: tsukuba
パスワードを入力してください。キャンセルするには Ctrl+D キーを押してください。
```

```
パスワード:（メールで送られてきたパスワード）
確認入力　:（確認）
standard または radius の指定: standard
```

設定が終わったので接続してみましょう。

```
VPN Client>AccountConnect tsukuba
```

はい！これで接続できたはずです。ifconfig で確認してみましょう。

```
vpn_tlan  Link encap:イーサネット　ハードウェアアドレス **:**:**:**:**:**
          inet6 アドレス: 2001:200:****:****/64 範囲:グローバル
          inet6 アドレス: fe80::****/64 範囲:リンク
```

上記の様に inet6 アドレス　範囲:グローバル　が取得出来ていれば成功です。そしてそのアドレスがあなたのマシンの IPv6 アドレスです。そのアドレスに外部からアクセスすれば SSH でアクセスが可能なはずです。このアドレスは固定アドレスで何度接続し直しても同じマシンなら取得できるアドレスは変わらない様です。クライアント側の.ssh/config などに書いておくと便利です。しかしこのままだと接続する度に色々とコマンドを打たないといけないので不便です。というわけでシェルスクリプトを作ってしまいましょう。

```
#!/bin/bash
sudo /usr/local/vpnclient/vpnclient start
/usr/local/vpnclient/vpncmd /CLIENT localhost /CMD:AccountConnect tsukuba
```

後はこれを home に vpn.sh などの適当な名前をつけておけば./vpn.sh の一発で起動が完了するので便利です。

4.4 DNS 設定

設定は終わったのでもう外部からのアクセスが可能になりました。ここで終わっても良いのですが、いざ外部からアクセスする時に IPv6 のアドレスをいちいち入力するのは面倒ですし、そもそも覚えられるわけありません。そこで DNS の設定をしてドメインを貰っちゃいましょう。もちろん無料で出来ます。今回は「Domain@けんどもネット」(https://domain.kendomo.net) さんを利用させていただきます。

まずは domain.kendomo.net にログインしなければなりません。その前にログインには openID(http://www.openid.ne.jp) が必要ですので、取得しておきましょう。ログインしてニックネームの設定など必要事項を設定したら、メニューからドメイン名追加をします。注意事項や指示に従って好きなドメインを取得しましょう。次にメニューのドメイン名管理から設定ページを開き、先ほど登録したドメインのレコード設定を開いてください。レコード設定ページでタイプを「AAAA」にし、データ欄に自分の IPv6 アドレスを入力して追加を押します。これで設定完了です。

ではアクセスしてみましょう。例えば example.dna.jp というドメインを取ったなら、IPv6 の使える環境で次の様にコマンドを入力します。

```
$ ssh example.dna.jp
```

接続できましたか?ドメイン名があるとカッコいいし便利ですね！

5 最後に

いかがでしたでしょうか?プライベート IP しかなくて絶望に伏していた私はこれのおかげで非常に助かっています。

余談ですが、私の家では、

```
Windows マシン <= LAN => Ubuntu マシン <= VPN => 筑波大
```

という感じに繋げて使っています。Windows マシンがメインなんですが、結構電気を食うので使いたいときだけ wakeonlan で叩き起こして使っています。Windows には VNC サーバが入っているので Ubuntu を踏み台にして SSH でローカルにポートフォワードすることで VNC を使っています。しかしやはりなんとなく不安定なのが難点です (^^;

最後に、筑波大のサービスの存在を教えてもらった yajima 先輩と wakisaka 君に感謝します。ありがとうございました。

fluxbox のオレスタイル

hayakawa

hayakawa@mma.club.uec.ac.jp

概要

普段使用するコンピュータには ubuntu をインストールし、愛用しているのだが gnome は重たいと感じることが多々ある。そのため、軽量のウィンドウマネージャーの一つである fluxbox を使用することが多い。そこで fluxbox のスタイルを作ってみた。今回使用した OS は ubuntu10.04 です。

1 準備

/usr/share/fluxbox/style/orestyle というファイルを作成する。このファイルに設定項目を書くことでスタイルを作って行く。

touch /usr/share/fluxbox/style/orestyle　もしくは

$ touch /.fluxbox/styles/orestyle

これだけで menu の style に orestyle というスタイルができる。実際に選択すると、スタイルとして機能していることが確認できる。この状態から、自分好みの設定を加えていくことで自分でスタイルを作ることができる。

2 書き方

スタイルの設定を書くときには各項目についてそれぞれ指定していくことになる。例えば次のように指定する。

menu.frame.font:　tahoma-8:bold

上のように一つずつ指定することもできるが、

*.font: tahoma-9

のようにすれば、すべての font に対して指定することができる。例えば font の設定は次のようにする。

```
menu.*.font:                    tahoma-10:bold
toolbar.clock.font:             tahoma-9:bold
toolbar.workspace.font:         tahoma-10:bold
toolbar.iconbar.focused.font:   tahoma-9:bold
toolbar.iconbar.unfocused.font: tahoma-8
```

3 設定項目

fluxbox 上の各パーツにはそれぞれ名前が割り当てられている。大きな枠組みとして、”window,toolbar,workspace,menu”などがありさらに細かく名前が付けられている。例えば、”window.title”といったようまた、次のような効果等の指定をすることができる。

- font:フォントの設定
- color:各パーツの色
- justify:テキストやイメージをどこに配置するか。
- effect:効果の指定
- shadow:影の指定

次のようにしてグラデーションの設定もすることができる。

```
menu.frame:         flat gradient
menu.frame.color:   #1A1A1A
menu.frame.colorTo: #777777
```

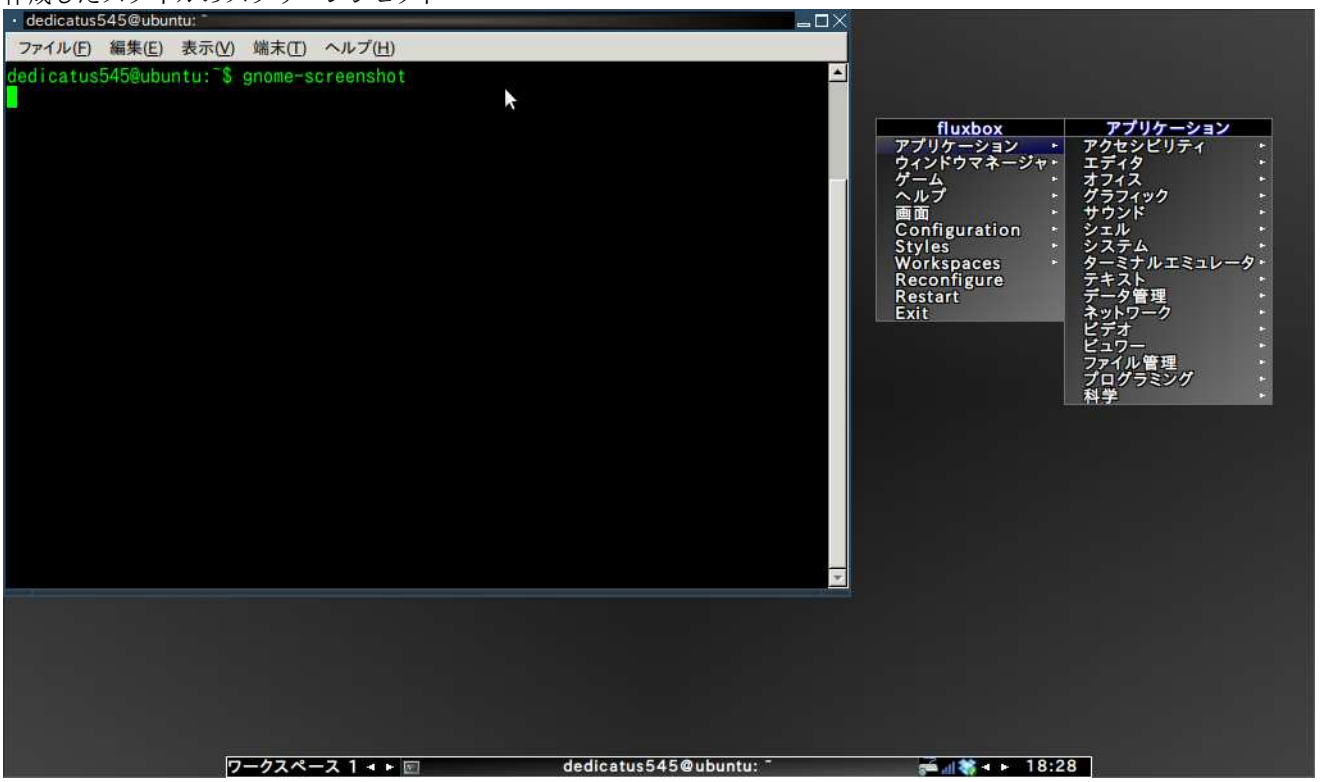
グラデーションの陰影は”flat,raised,sunken”から選択できる。また、ウィンドウなど、選択されているかどうかによって、設定を切り替えることができる。iconbar では focused,unfocused で示し、その他は focus,unfocus で示す。ここに示した以外の効果等の設定がいくつも存在する。

4　作成したオレスタイル

```
*.font:                          tahoma-9
*.color:                         #000000
*.colorTo:                       #C0C0C0
*.unfocus.color:                 #dadada
*.unfocus.colorTo:               #808080
*.unfocused.color:               #dadada
*.unfocused.colorTo:             #808080
#########FONT#########
menu.*.font:                     tahoma-9:bold
toolbar.clock.font:              tahoma-9:bold
toolbar.workspace.font:          tahoma-10:bold
toolbar.iconbar.focused.font:    tahoma-9:bold
toolbar.iconbar.unfocused.font:  tahoma-8
window.*.font:                   tahoma-8
#########MENU#########
menu.borderColor:                #6F6F6F
menu.borderWidth:                1
menu.bullet:                     triangle
menu.bullet.position:            Right
menu.hilite:                     flat
menu.hilite.color:               #4C4C4C
menu.hilite.colorTo:             #4C4C4C
menu.hilite.textColor:           #ffffff
menu.frame:                      flat
menu.frame.justify:              left
menu.frame.color:                #1A1A1A
menu.frame.colorTo:              #1A1A1A
menu.frame.font.effect:          halo
menu.frame.focus.color:          #00FF00
menu.title:                      flat
menu.title.justify:              center
menu.title.font.effect:          shadow
menu.title.font.shadow.x:        1
menu.title.font.shadow.y:        1
menu.title.font.shadow.color:    #0000FF
#########WINDOW#########
window.button.*.color:           #2B4B65
window.borderColor:              #2B4B65
window.borderWidth:              2
window.handleWidth:              3
window.label.focus:              flat gradient rectangle
```

```
window.label.focus.color:          #000000
window.label.focus.colorTo:        #555555
#########toolbar#########
toolbar.borderWidth:               1
toolbar.border.color:              white
toolbar.*.justify:                 center
toolbar.iconbar.focused:           flat gradient
toolbar.iconbar.focused.color:     #000000
toolbar.iconbar.focused.colorTo:   #555555
#########backgroung#########
background.color:                  #111111
background.colorTo:                #444444
background:                        SunkenGradientPyramid
```

作成したスタイルのスクリーンショット



5 最後に

今回は先に述べた事柄をベースにこのようなスタイルを作成した。作成した時点では、このスタイルに満足しているが、少しずつより自分色に改善していきたいと思う。

参考文献

[1] Okumatsu_Hiroshi Website,http://www.geocities.jp/okumatsu_hiroshi/Fluxbox_Tips.html

[2] Fluxbox Style Guide ArchWiki,https://wiki.archlinux.org/index.php/Fluxbox_Style_Guide

[3] fluxbox information,http://www2.odn.ne.jp/add10/fluxbox/

高度情報通信ネットワーク社会における人間存在の超自然的な在り方

yamazaki

yamazaki@mma.club.uec.ac.jp

概要

無駄に高度に発展した通信網は、今や日本全国だけではなく地球全土をも覆ってしまい、何れは外宇宙へと進出するだろう。
だが、我々は、今や籠の中に居る囚われの豚[*1]でしかないのだ。

1 序章

希望はすこぶる嘘つきであるが、とにかくわれわれを愉しい小道を経て、
人生の終わりまで連れて行ってくれる。

これは誰の言葉だっただろうか。ラ・フォンテーヌか。それともラ・ロシュフコーか。ついつい Wikipedia で調べてみると、どちらも巻き毛でイカツイおっさんなので、正直どうでも良い。

こんなおっさんが嘘つきかどうかも分からないが、とにかく私は愉しくもない無駄な時間を経て、今日の終わりまで連れていかれてしまいました。

”命… 夢… 希望… どこからきて どこへいく?” などというケフカめいたことを言っても、紙と神[*2]の無駄遣いなので、本文に入りたい。とは言うものの、果てさて、ここまで書いたは良いのだが、一体何を書きたいのかということをここで自問自答してみるという愚行に走る。

2 懸念の想起

一番最初に考えたことは「スマートフォンの形状 –通話しやすい形とは何か–」
Xperia にしろ、IS シリーズにしろ、iPhone にしろ、あのゲーム & ウオッチみたいな形状で通話するのは厳しいのではないだろうか。なので、通話しやすい形状とはどの様な形になるのか、ということを考えたかった。だが、破滅的にデザインセンスとデッサン力が無いため、20 分で放棄するアイデアとなった。このアイデアはいずれどこかでやりたいと思うので、Blender などのソフトの使い方を知ることが肝要だろう。なんだかんだいって、ガラパゴス化した携帯は使いやすいのではないだろうか。

次に考えたことは「基礎化学実験 **C**urry」
鍋の中身がオーバーフローしない様に、念入りに具材の圧縮率を上げるだとかの、変な言い回しの多用。ただ、これは他者と被る可能性が高すぎなので、10 秒で丹念に頭から拭い去った。とはいえ、意外と誰も書いてはいなかった。真面目な人間が数多く、その為にばからしいことなど誰もしない。個人的には、レポートの規定枚数に届かないからといってカレーのハウツーを書くなんて行為の真似をしたくないというのがあったのだが。まぁいい。このアイディアはいい奴だったよ。

最後に考えたことは「iPhone アプリの制作行程」
ただ、プログラミングのプの字すら分からないプー太郎には制作行程なんて土台無理なおはなし。Objective-C の使い方どころか、C 言語のポインタの使い方すらも知らない。それに、Mac 端末などという人智を超えた貴重品なぞ、清貧な一学生には接点もある筈もなく。情報基盤センターに Mac 端末はあるが、見えません。見ていません。見たらダメ。iPhone SDK とか言われても、スーパードンキーコングなのかスーパードクター K なのかという始末。そんなレベルであるからにして、製作に取り掛かる段階の大幅な手前で断念した。

---

*1 某ラーメン屋で出される餌が好きな人種ではない
*2 きしん まじん めがみ

3　感覚的ヒロイズム

諦めの連続で結局はこのよく分からない文章に辿りついたが、資本主義の豚を批判したいわけではない。共産主義は甘美だが、歴史を紐解いてみても幻想でしか無いということがはっきりと分かる。こんなことを書くと、割と本格的に MMA[*3]は宗教団体だとか、ある種の思想[*4]を持った集団だと思われかねない[*5]。

割とまともに考えはしてみたのだが、何分プログラミングを始めたばかりの矮小な存在には円の描写ですらまともに出来ていない為に、部誌として満足いくものを書ける筈もない。始めたばかりの LabVIEW だとか、Erlang にしても、書くことなど何一つ存在しない。かと言って、電子工作で何かを作り出すということも出来やしないし、する気力もない。何故ならば、実験レポートという、至上の拷問が寝ても醒めても私を苛むのだから。

ここまで書いて、実はカレーの作り方が一番まともで、一番まとまったものだったのかもしれないと思い始めてしまった上に、二転三転する話題を文章中で展開することによって更に意味不明さを増してしまった。二転三転する度にあーだこーだと考えを巡らせ、浪費の時間に追われ追われて一日の終わりへと発散する。

考えてみて、良いと思う案を出しては紙に書き出し、それをインターネット上で調べて調べて調べて調べて……。調べれば調べる程に無理であるということと、無謀であるということ、時間が刻一刻と失われていくということが錯綜し、考えはカレーという数値に収束してしまう。

結局はそういうことである。インターネットという電脳世界はいやに広い。寝転がりながらも、ソファに横たわりながらも、歯を磨きながらも、お風呂に入りながらも、料理を作りながらも、その広い広い世界に触れることが出来る。何となく思いついたことをインターネットで検索し、無理だという結論に至る。この過程こそが、無駄の中の無駄、本当の無駄ではないだろうか。ワンガリ・マータイさんも時間モッタイナイとか言ってきそうなレベルな程に、無駄であったのだろう。

4　結論

これまでの全てを要約すると、たったの一文となる。

Don't think, Feel!

無駄なことを考えるよりもまず、感性と惰性で物事を考える。これこそが、人間存在の超自然的な在り方だと、私は考える。ここで題意を持ってきたが、実質そんなに考えもせずに付けたので、この結論に収束するとは思っていなかった。まさしく要約のとおりになってしまったという、稀有な例ではないだろうか。

下手に変なことを考えて、調べて時間を潰してしまうよりは、何も考えずにやってみることも一番なのだろう。C 言語にしてみても、考えるよりもまず手を動かさないとプログラムは出来上がらない…気がする。つまりは、そういうことであろう。きっと。

もう少し空白があるので、付け足しをしてみようと思う。
よくインターネット上での煽りで

LAN ケーブルで首吊って云々

というものがあるが、今の時代は無線 LAN が台頭してきている。無線で首を吊ることが出来れば、日本の未来は明るいように思えるのは、きっと今日は日曜日だからだろう。こんなこと書くよりも、月曜日を楽しく過ごす 10 の方法だとかを書けば良かったと、私は思う。

[*3] Manual Metal Arc：被覆アーク溶接の意
[*4] vi 教と emacs 教の対立は、知らない間に起きているのかもしれない
[*5] 否、思われている

UNIX 系 OS を使う人々を分ける二つの戦争

松宮　遼

matsumiya@mma.club.uec.ac.jp

概要

Unix 系 OS は昔はサーバー用途で使うことしかできなく、普通の人達はその名前を知ることすらなかったであろうが、最近になってワークステーション向けの Unix 系 OS が出始めたことによってユーザーが増えてきた。しかし、そのことによって Unix 系 OS ユーザー間で大きく分けて四つの派閥ができてしまった。今日はその派閥を分けてしまった二つの戦争を紹介する。

1　カーネル戦争

1.1　概要

決して某 KFC の白髪のじっちゃんのことではない。OS の基盤のことで、同時に宗教である。このカーネルは大きく分けて二つ存在し、常に対立している。大学で UNIX 系 OS を学び、即戦力になれると思って SI 系企業に就職してから「企業と大学が別のカーネルだった」という事実に気付いてももう遅い。会社は宗教の違いにすぐに気付きすぐにその新入社員を干すことだろう。

1.2　BSD カーネル

Unix の流れをそのまま受け継いだカーネル。MMA のサーバーもこれを使用している。利用者の中には GUI を毛嫌いしている人種も存在している、硬派といっても間違いではないだろう。

1.3　Linux カーネル

Unix の流れから分岐し、Linus という人が開発したカーネル。こちらの方は GUI などの新しい流れには比較的寛容的、BSD を硬派とするならこっちは軟派である。ちなみに筆者も Linux カーネルの信者（Debian 派）である。

2　エディタ戦争

2.1　概要

カーネル戦争においては無宗教でも、こちらには必ず属さなければならない。これも一つの宗教戦争である。電通大内部でもこの宗教戦争はよく見られる。

2.2　emacs

無宗教で電通大に入ると一年の前期で emacs の洗礼を受ける。この為電通大生にはこれを使う人が多い。便利な機能が多く使いやすいが、動作が重く実はこれを書く数日前まで電通大の計算機で大暴走していたという噂が … 。

2.3　vim

完全な CUI ベースなので動作は非常に軽い。ちなみに今この文章を打つのに使っているのも vim である。使いこなせればコレほど使いやすいエディタはないが、使いこなすまでが少し大変。一年の後期のプログラミングの授業で vim を知らない学生の前で、普通に vim を使って授業を進めている講師がいるとかいないとか … 。

3　最後に

今回は Unix 系の触れてはいけないところを触れた。もしかしたら（ピーーーーー）棟の屋上から重力加速度の実験に参加させられるかもしれない。万が一、そのようなことがあったのならばこの文章を思い出してくれると安心して成仏できるだろう。

iFrog

革命的で魔法のような変形。しかも、信じられない素材で。

iFrog

noy
noy@mma.club.uec.ac.jp

概要
革命的で魔法のような変形。しかも、信じられない素材で。

1　大きな驚きをあなたに

iFrog は偶然誕生しました。僕が大好きなマックの箱を投げたり潰したり、ちょっと工夫して完成しました。

2　最高の素材

- マックフライドポテト M サイズの箱 1 個

3　信じられない製法

3.1　潰す

図 1　尻を折り込む

図 1 の丸をつけた部分（これからは「尻」と呼びます）を、箱の内側に折り込みつつ箱を潰します。しかし、尻には点線がありますが、それはあまり気にしないでグシャッと潰します。尻が内側に折られて入れば特にこだわる必要はありません。

3.2　頭を畳む

頭というの図 2 の丸をつけた部分のことです。ちょうど頭の凸が体の凹とぴったり合うように折ります。

3.3　股を作る

図 3 写真の線に沿って谷折りします。箱の方にも線があるのでわかりやすいと思います。

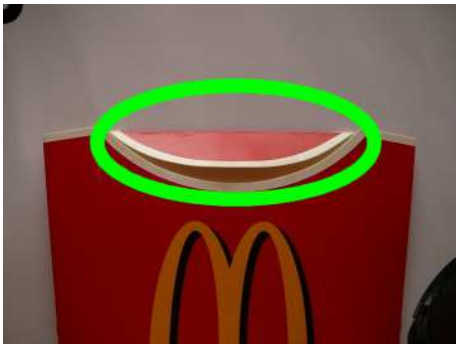


図 2　頭を畳む

図 3　股を作る

図 4　足を作る

3.4　足を作る

図 4 写真の線に沿って谷折りします。ここをなんども折ると却って反発が減りあまり動かないので、気をつけてください。

3.5　完成

線に沿って折っている内に完成です。まるで最初からこのためにあったかの様だと思いませんか?図 5 の丸で囲った部分を

図 5　完成

押して弾くと跳びます。うまく作れると空中で回転します。これで、みんなの視線が、あなたに。

4　iFrog をパーソナライズする

さあ、飛び立とう。

4.1　丸みをつける

図 6　丸みをつける

図 6 のように iFrog の脚となる部分に丸みをつけてやるとバネができて、より跳ぶようになります。

4.2　頭を立てる

図 7 のように頭の真ん中に折り目をつけると、簡単に股が広がってしまうのを防げます。

図 7　頭を立てる

飛躍するアクセサリは、全国の Mc'Store でお求め頂けます。

百萬石 2010 年 秋号 © 電気通信大学 MMA

2010 年 11 月 19 日　初版第 1 刷発行　【本書の無断転載を禁ず】
2010 年 11 月 20 日　初版第 3 刷発行

著　者　fujii, hayakawa, hiro1375, kyogoku42, matsumiya, noy, sawada, shimazaki, wakisaka, yamazaki, ytoku
編集者　ytoku
発行者　電気通信大学 MMA
発行所　電気通信大学 MMA 部室
〒182−8585 東京都調布市調布ヶ丘 1−5−1 サークル会館 2 階
http://www.mma.club.uec.ac.jp/ (IPv6)
http://w3.uec.ac.jp:8081/club/mma/ (IPv4)
印刷所　電気通信大学 MMA 部室
製本所　電気通信大学 西 9-115
表　紙　hiro1375