平成 17 年 11 月 18 日
京都工芸繊維大学コンピュータ部

はじめに

このLimeも今年でようやく0x20号となりました。ついに32号、というところです。32というと原子番号でGe、ゲルマニウムにあたります。つまりこの32号のLimeというのは、ある意味でゲルマニウムなわけです。

ところでみなさんは、コンピュータ部というものがふだん何をやっているかご存知でしょうか。この部に入るまで、僕はコンピュータ部というものは、薄暗く汚らしい部室に、光るディスプレイ、散らばる電子部品、怪しげなプログラムと、それより怪しげな人々、といったものと切っても切れない関係にあると思ってきました。それでこそコンピュータ部というものです。ところが僕の、そのささやかな夢は、たった一人の掃除大臣によって打ち砕かれてしまったのです。彼の名はくっきー。コンピュータ部の裏ボスです。

しかし、それにしても世の中の偏見には困ったものです。コンピュータ部とは、ワープロソフトや表計算ソフトの使い方なんかを勉強するところだと考えている人がいました。ある特定の物事、特に最近流行りの日本文化に対する造詣や知識が深すぎる人たちの集まりだと考えている人もいます。実のところ、そのどれもが当たっているか外れているかのどちらかです。

我々はそのような偏見を持つべきではないのです。偏見を捨て、コンピュータオタクたちと共存すべきなのです。そして、そのためのLime32号です。これは、希望の書なのです。

平成17年11月16日
京都工芸繊維大学コンピュータ部部長 高井 真也

目 次

1 コンピュータ概論

電子情報工学専攻 2 回生 山本 大介

はじめに

6 年間書き続けてきた Lime もこれが最後の記事になりました。そこで、6 年間の総集編としてハードウェアからソフトウェアまでのさまざまな解説や自分の考えを書いていきたいと思います。

1.1 コンピュータ

1.1.1 スイッチ

スイッチの塊

コンピューターの仕組みとはどういうものなのだろうか？本の中では、半導体の電位がどうとか、トランジスタがどうとか、IC がどうとかいろいろ複雑なことが書いてある。しかし、コンピューターは構造こそ複雑だが、その原理はとても単純なものなのだ。

「コンピューターはスイッチの集合体」なのである。今、テレビゲームのボタンを押すとキャラクターがジャンプするとしよう。ボタンを押すとなぜキャラクターはジャンプしたのだろうか？テレビゲームのボタンを押すと、そのボタンの下にある回路に電流が流れる。その電流でゲーム機本体にボタンが押されたことを知らせるスイッチが押される。これにより、今度はコントローラーからゲーム機本体に電流が流れ CPU の中にあるスイッチを入れる。そのスイッチが入れられると今度は別のスイッチが入れられる。それを永遠と繰り返し、最後にはテレビ画面のキャラクターの上の部分を光らせるスイッチが入れられ、キャラクターの下の部分のスイッチが切られることによりキャラクターはジャンプするのである。

小学生の時に豆電球が光る回路を作ったことがあると思うが、基本的にはそれとまったく同じことなのである。テレビゲームの場合はそれがちょっと複雑になっただけなのだ。パソコンなどの CPU はこのようなスイッチが 1000 万個ぐらい入っていて、それが相互に複雑に繋がっているのである。スイッチだけで構成された回路を**組み合わせ回路**と呼ぶ。

スイッチの正体

さて、ここまでスイッチと言って来たがこのスイッチとは一体何なのだろうか？小学生の時の実験で作った回路のスイッチは金属と金属がひっついたり離れたりすることによって、実現されている。小学生や中学生の知識でも回路を複雑にすることができる。スイッチで電流を流す対象を電磁石にするのである。電磁石によって別のスイッチが動くよう作れば、一つのスイッチを入れただけで複数のスイッチを同時につけることができる。また、2 つのスイッチを直列につないで、2 つ同時につけないと動かないようにしたり、並列につないで 1 つだけつないでも動くようにするといったことも考えられる。電磁石によって別のスイッチを入れたり切ったりする装置は一般的に**リレー**と呼ばれ電子回路の基礎になっている。

しかし、電磁石は金属線をぐるぐる巻きにしなければならないので大きくなってしまう。しかも、電磁石は磁力が強くなるまでに時間がかかるという問題がある。そこで、昔から人々はこのスイッチをいかに小さくできるか、そして速く動かせるかを研究してきた。そして、真空管そしてトランジスタがそれに最適であることがわかったのである。以下ではそのトランジスタについて議論してみよう。

半導体

トランジスタはアメリカの AT& T のベル研究所で発明されたもので、電話会社で発明されたことからもわかるように、もともと電流を増幅する目的で開発された。しかし、これが今に至るまで最も小さくて最も速くできるスイッチの発明でもあったのだ。トランジスタは電流を増幅することもできるが、電流を遮断する機能も持っていたのである。つまり、これはスイッチなのである。

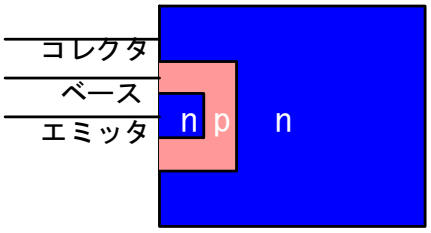


図 1.1: npn 接合型トランジスタ

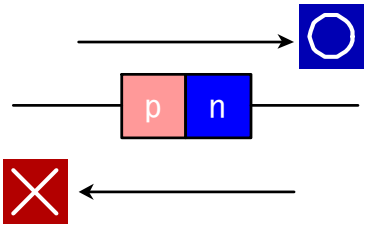


図 1.2: ダイオード

図 1.1 は接合型トランジスタの模型図である。トランジスタも図 1.2 のダイオードのように p 型シリコンと n 型シリコンが繋がっていて、p から n にしか電流が流れない。つまり、コレクタからエミッタに電流は流れない。しかし、npn 接合型トランジスタの場合ベースからエミッタに電流を流し込むと p 型半導体部分に n 型半導体の電子が流れ込んでくるので、n 型半導体に近い状態になる。すると、コレクタからエミッタにかけて電流が流れるようになる。

トランジスタを増幅器として使う場合はベースから流し込む電流の量を微調節してコレクタからエミッタに流れる電流を調節する。ステレオのアンプなどはこれを利用している。このような連続性のある利用法によって作られた回路を**アナログ回路**という。

しかし、コンピュータはトランジスタをスイッチとして使うので、微調整はしない。ベースに電流を流しこむか、流し込まないかの 2 つに 1 つである。この利用法によって作られた回路を**デジタル回路**という。コンピュータ言語が 2 進数によって記述されるのは、トランジスタの 2 つの状態を利用することから来ている。2 つの状態しか利用しないので、電気回路の問題としては**デジタル回路**の方がはるかに簡単な問題となる。

1.1.2 記憶装置

補助記憶

コンピュータの機能としてもう一つ忘れてはならないのが、記憶装置である。コンピュータには様々な種類の補助記憶装置がある。初期のコンピュータは紙テープに孔をあけるというものだった。孔がないのが 0、孔があるのが 1 という具合である。ASCII コードで `0x7F` (2 進数で `1111111`) が `DEL` (削除) コードなのは孔を全部あけると削除したことになるという歴史的経緯がある。

孔が開いているとき、スイッチをオン、ふさがっている時スイッチをオフにすれば先ほどのスイッチの組み合わせでどんな複雑な計算もできる。しかし、紙テープでは時間もかかるし場所もとってしまうので、現在はどんどん高集積化が進んでいる。

磁気テープ 紙テープの延長上で磁化しやすいものを塗布したテープを利用。再利用が可能になった。N 極か S 極で 1 と 0 を判断。

フロッピーディスク 記録できる量は小さくなるが、思い通りの場所に記録できる磁気ディスク。携帯性も上がった。

ハードディスク 磁化しやすい金属板そのものを利用。容量が格段に増えた。

光ディスク 高集積と携帯性のバランスがとれた上、大量生産しやすい。反射率が高いところが 0、反射率が低いところが 1 である。

フラッシュ 元々、一度しか書き込めない ROM を紫外線で初期化できるようになったものが、どんどん改良され記憶装置化したもの。小型化しやすい。

主記憶

さて、先ほどまで補助記憶と呼ばれていたものは電源が消えても保存がきくものであるが、読み書き速度が遅い。より速度が速い記憶装置として主記憶装置がある。プログラムの変数など、頻繁に利用されるデータがある。負荷の集中するインターネットのサーバー (google, Yahoo など) は頻繁にアクセスされるウェブページのデータも主記憶装置に入れている。

主記憶装置は回路を簡単化するのにも利用される。たとえば、掛け算の問題を解く回路を作るとき先ほどのスイッチの組み合わせだけで作ると、掛け算の全部のパターンを網羅した複雑な回路を作る必要がある。しかし、筆算の要領で一桁の掛け算の足し算にすれば回路は簡単にできる。その代わり、記憶装置への読み書きが行われるので解く時間は遅くなってしまう。時間を速くするのを

時間最適化、回路を簡単にするのを**空間最適化**というが、うまく中庸をとるのが回路設計者の腕の見せ所である。

1.1.3 入出力

コンピュータは入力と出力があってはじめて役に立つ。もちろん、円周率を計算するだけのコンピュータならば、出力だけのコンピュータというのも考えられなくはないが、ソフトウェアで実現することが多くなった今、入力と出力はコンピュータの必須条件である。コンピュータの中には補助記憶への書き込みを出力、読み込みを入力とみなしているものもある。この事は、コンピュータの概念をどこまで広げるかという意味合いもあるが、CPU と補助記憶を切り離すことによって高速化を図っているため、そう見なすことも多い。

入出力で重要なのが**インタフェース**である。インタフェースは**接合面**という意味であるが、コネクタの形状のような物理的な側面から通信時の電圧といった電気的な面、通信方法といった多岐にわたる接続方法の概念である。キーボードとコンピュータといった機械的なインタフェースもあれば、人間とキーボードというユーザインタフェースもある。

通常、ユーザインタフェースからの入力はコンピュータ同士のやりとりとくらべて格段に遅い。キーボードやボタンなどを押す間隔はせいぜい 1 秒間に 3,4 回が限度といったところである。ただ**ユーザインタフェースの入力に対する出力はできる限り速く行われなければならない**。人間は入力に対して出力がないと、壊れていると思ってしまうのだ。

1.1.4 インタフェース

私は現在のコンピュータのインタフェースには 2 種類あると考えている。対人間インタフェース (ユーザーインタフェース) と対コンピュータインタフェースである。これは、ハードウェアに限らずソフトウェアにも言える。なお、GUI はグラフィカルユーザインタフェースのことで、現在の一般的なパソコンの画面のように文字以外の表示ができるものである。

表 1.1: インタフェース

対人間指向	対コンピュータ指向
GUI	プロトコル
アプリケーション	モジュール
キーボード	ネットワークケーブル
ディスプレイ	無線

かつて、対人間インタフェースと対コンピュータインタフェースはしばしば重複していた。紙テープ時代のプロの技術者は紙の上の孔の配置だけで、命令の配列を理解していたと言われている。また、15 年ほど前の CUI （文字だけ表示するインタフェース）全盛時代は標準出力と標準入力という概念を用いて、ユーザへのインタフェースをそのままコンピュータにとりこませることができるようにし、2 つのインタフェースを構築する手間を省くと同時に、ユーザがインタフェースを結合

して複雑な処理が行えるようにするのがよいという思想があった。また、ソフトウェアはなるべく単純な機能をもつものをたくさんつくるのがよいとされた。

しかし、**時代は流れ GUI が主流になった今日、人間向けのインタフェースとユーザインタフェースは明確に切り分けられるようになった**。そのため、いったん人間向けのインタフェースになったら最後、それを再度コンピュータ向けにするのは逆に手間がかかるようになってしまった。そして、一つのソフトウェアが大量の機能を持つようになり、あらゆることができるようになった。以前であれば、メール送受信ソフトとメール閲覧ソフトは別であるのが普通であったが、どんどん統合が進み Windows に至ってはブラウザと統合するだけでなく文書作成ソフトさらには OS とまで統合する始末である。

しかし、それによってアプリケーションは格段に便利になった。ユーザはあまりコンピュータのしくみを知ることなくあらゆることができるようになったのである。ただ、便利になればなるほど、効率がよくなればなるほど、何か障害が発生したときに思わぬところにまで影響がでるようになってしまう。文書閲覧ソフトのバグのせいでメールが受信できなくなってしまうこともありえるのだ。また、大規模なソフトウェアは作れる事業者や組織が自ずと限られてくるため、インタフェースの画一化を生んだ。インタフェースの画一化は開発効率や運用効率は格段によくなるが、ウィルスなどの脅威に対して非常に脆弱になってしまう危険性もはらむ。そういう意味で、OS にしろブラウザにしろ第 2、第 3 の作り手が存在することは大きな意味がある。

1.1.5 ネットワーク

インターネット

現在、コンピュータ間ネットワークはそのほとんどが**インターネット**となっている。インターネットの特徴は中央集約サーバーが不要で、放射状にしか構成できなかったネットワークが網のように横のつながりも持てるようになったものと言われている。しかし、その考え方はインターネットが最初というわけではない。

電力の安定供給の観点から、このような**網の目状ネットワーク**の研究は昔からなされていた。北アメリカのエリー湖環状送電網はその典型で、エリー湖の環状送電線をはじめ、あらゆる電力会社の送電線や発電所が複雑にからんでいる。このような送電網は一箇所の発電所が停止しても、環状になっているため別系統から電力が供給されるようになっている。しかし、網の目状の送電網は想定されうる異常事態があまりにも多すぎるため、対応できる異常事態はある程度限られることになる。2003 年の北米大停電はこの想定の範囲外の異常事態が起きたため、連鎖的に機能が停止し大停電を引き起こしてしまった。

一方、日本の送電網は現在でも中央集中で**放射状ネットワーク**になっている。というより、放射状ネットワークにしてきたのである。これは中央集中型を基本に横方向にバックアップの線をとったほうが、制御が簡単なのである。ただ、このような電力供給網は綿密な計画と統合された意思決定が必要となる。日本の場合地域ごとに電力会社が 1 つしかないため[1]送電線の容量や発電所の施設などを一元管理している。これは、日本の安定した電力供給に大きく寄与している。

[1]現在は電力自由化で複数の会社が発電事業を行っているが、東京電力・関西電力等の既存電力会社は安定した電力供給

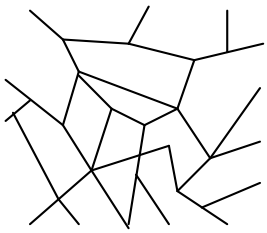


図 1.3: 網の目状ネットワーク

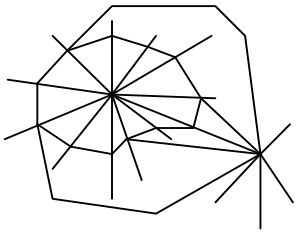


図 1.4: 放射状ネットワーク

このような事情から網の目状のネットワークは無計画に作られたネットワークを無理やりつなぐもので、本来は中央集中が理想であるという考えが強かった。しかし、軍事用に開発されたインターネットのプロトコルが民間開放されると、またたくまに広まった。これは国際的なネットワークを構築するのに綿密な計画と統合された意思決定が絶望的であるという側面とアメリカの文化発信力が影響していると思われる。国際的な通信網も本来は放射状のネットワークに横のバックアップ線が安定性の面からも理想的であり、(実際に、日本国内のインターネット網はそういう構成になっている) トラフィック管理も簡単なのである。

インターネットは情報を通信しようとすると各プロトコルに対応したヘッダ情報が負荷されるため、元々、通信線上は元の情報より多めの情報がやりとりされる (冗長性が高い)。しかし、近年は Winny などの匿名性を高めるために多数のコンピュータを経由した通信を行うソフトウェアが普及したことや、SPAM、ウィルスメール、DoS 攻撃など有効でない情報の通信がネットワークのほとんどを占めるようになっている。物理的なだけでなく論理上も網の目状ネットワークと化している今日では通信量の増大にさらに拍車がかかっている。その通信量に対応した設備の構築はかなりの費用がかかっていると考えられる。

ネットワークアプリケーション

ネットワークの普及と同時にさまざまなネットワークアプリケーションが登場した。インターネットのネットワークアプリケーションは、データをダウンロードしたりするだけのものや、分散処理の相互通信、情報の要求とそれに対する応答を受け取るものなどがある。いずれのネットワークアプリケーションの場合も通信を要求されたら早急に応答しなければならないため、ユーザインタフェースの構築に似ている。

実際、ネットワークアプリケーションを構築する場合 (特に複数の相手と同時に通信する場合) 別の**プロセス**[2]を作ったり**スレッド**[3]を作ったりする。それにより、現在のプログラムの処理やユーザインタフェースとネットワーク処理を分けることになる。

ここで問題になるのがデータの共有だ。OS 上で複数のプロセスがメモリを共有するときも共有の問題が発生するが、この時の共有の問題は、2 つのプロセスが 1 つの資源を扱おうとする場合に発生するもので、専用ハードウェアによってしか解決できないことがわかっている。

が妨げられるとして自由化に反対している。

[2]OS に記憶領域を割り当てられたプログラム

[3]プロセス上で、C の main 関数など最初に起動されるプログラムの流れとは別の流れ

ネットワーク上のデータ共有の問題はそれに加え、データの相互整合性というものがからむ。つまり、相手と同じものを共有していると思っているデータが実は相手のものと違う可能性があるのだ。通信情報の到達が保障できない通信路においては、何度確認をとろうとも整合性を取ることはできない。

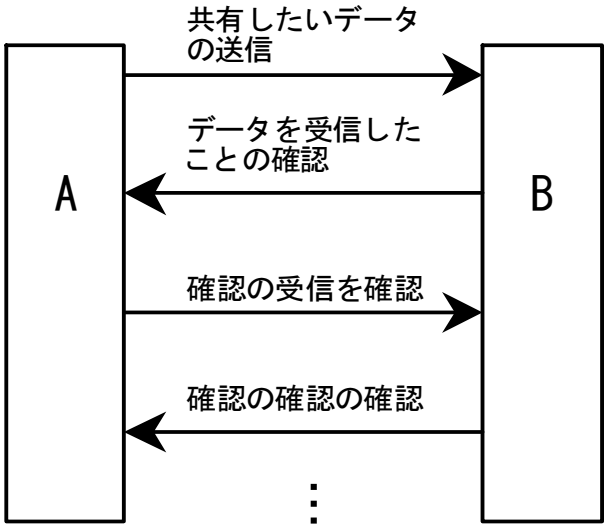


図 1.5: 終わりのない確認合戦

図 1.5 のように、A が共有したいデータを B に送信することを考える。データが届いたら B は A にその確認メッセージを送る。これで一見データの共有ができたように見える。しかし、確認メッセージが何らかの理由で届かなかったらどうなるだろう？A はデータが共有できているのかいないのか判断できない。そこで、さらに A が確認メッセージを B に送るようにすれば、B が確認メッセージが届いたかどうかを判断できるので A のジレンマは解消される。しかし、今度は B に確認メッセージが届かなかったらどうなるのか？このように、永久に確認メッセージをやりとりしなければ、データの共有は確認ができないのである。データの共有の保障は確認メッセージが送信された時刻までであり、その時刻以降については保証はない。現在ではネットワークの通信の高い信頼性がこのジレンマを解消している。

1.1.6 CPU

コンピュータが社会のありとあらゆる領域で使用されるようになり、コンピュータの中核をなす CPU はどんどん複雑になってきている。そして、近年コンピュータから幽霊やお化けがでるような小説や映画が出てくるようになった。その影響もあってか、CPU はかなり高度な計算や処理ができると勘違いしている人が多い。CPU ができる計算は今も昔も以下のとおりである。

- 2 つの整数の足し算、引き算

- 2 つの整数の比較
- 2 つの 0 または 1 の足し算、掛け算

これだけである。CPU は与えられた 2 つの数字とこれらの数少ない演算を繰り返し行うことしかできない。小数の演算は小数点の位置と数字の筆算によって求められるので、整数演算と変わりがない。また、複雑な命令を持つ CPU も存在するが、いずれも上の演算の組み合わせに過ぎない。3 つ以上の足し算などはあらかじめ 2 つを計算して残りの 1 つと足すという方法によって実現される。

CPU は与えられた 2 つの数字と命令からその結果を出力するだけで、ジャンプ命令や分岐命令、割り込み[4]で命令の順番を時々変えるだけである。こんなことをちょっと考えると、コンピュータから出てくるお化けというのがいかに滑稽なのかがわかるだろう。プログラム言語の高級化、プログラムのモジュール化、インタフェースの改善の繰り返しによって、複雑になった結果、コンピュータは何やら得体の知れないものにまで発展したのである。

1.2 マイクロコンピュータ

1.2.1 ユビキタス

CPU の高周波化、メモリの高集積化、ハードディスクドライブの大容量化がどんどん進んでいるが、5 年ほど前に比べるとその伸びは明らかに鈍化している。物理的限界に近づいたのが主な原因であると考えられている。画期的な技術の登場も考えられるが、それでも 100 倍も 1000 倍も改善することはないだろう。現在の技術動向は、既存のコンピュータをいかに小さくいかに低電力で動かすかに移っている。

そんな中注目されているのが**ユビキタス**という言葉で、社会のありとあらゆる場所にコンピュータをいれようという試みである。その場合現在のパソコンのように、ディスプレイがあって、キーボードがあって、マウスがあるというようなシステムとは限らない。そのようなものが一切ない世界では現在の GUI 指向のプログラムや OS はまったく役にたたない。

そのユビキタスを実現するものの一つに**マイコン**と呼ばれる CPU とメモリが一体となったチップのコンピュータがある。現在はありとあらゆる製品に内蔵されコンピュータの数で言えばパソコンの数をはるかに上回る。OS の資源管理が必要な場合もあれば不要な場合もあり、小型で省電力にするために極力プログラムを小さくすることもある。このようなコンピュータの世界では、日ごろのコンピュータのプログラミングなどとは違う別の世界を味わうことができる。

1.2.2 いざ、マイコンの世界へ

マイコンのプログラミングを進めていくと、必然的にコンピュータへの理解が進む。現代の複雑化したパソコンでは中のしくみをうかがい知ることができない。しかし、マイコンでは簡単にその世界に入り込むことができる。

[4]ユーザの入力やネットワークのデータ受信などで強制的に命令の順序を変える仕組み

マイコンのプログラミングはアセンブリまたは C によって行うことが多い。企業レベルであれば当然高級言語も使用されるが、趣味レベルだと C で十分だろう。なお、アセンブリは製造会社によりかなり違うため可読性を挙げるためにもなるべく C で書くことをお勧めする。私はテキサス・インスツルメンツの PIC、ルネサステクノロジの H8 と M16C などのマイコンでプログラムを書いていたが、初心者向けかつ安価なものとしてオークス電子 [4] のマイコンパッケージをお勧めする。やり方が 1 から丁寧に書いてあり、プログラムをある程度書ける人ならマイコン初心者でも簡単に開発ができるようになっている。

以下は PIC のプログラムである。PIC はアーキテクチャがとても簡単なため、プログラムに少しコツがいる。条件分岐で任意のアドレスに飛ばしたり、2 つのメモリから値をとって計算するというようなことができない。レジスタの中のデータに注視してプログラムを見ていく必要がある。

PIC16F84A のアセンブリプログラム例

```
; コメントは ; の後ろ

; ポート名、メモリアドレス対応表が多くのマイコンについている
; これを INCLUDE するとかなりプログラムが見やすくなる。C も同様。
; 命令の前の [TAB] を忘れずに
    INCLUDE "P16F84A.INC"

; 変数の宣言のようなところ。
; メモリのどこに割り当てるか指定する。
; C ならば自動割り当てだが #pragma で指定できるコンパイラもある。
; H8、M16C などは .EQU
HOGE EQU h'10'

; 命令をどこに配置するかの指定
; C ならば自動割り当てだが #pragma で指定できるコンパイラもある。
; H8、M16C などは .ORG
; PIC は メモリアドレス 0 からプログラムが開始されるが、
; そうでないものも多いので注意
    ORG     0

; C と同じラベル。goto などで飛ばせる。
    GOTO    START

; ORG 4 が割り込みの予約アドレスなのでいきなり 8 に飛ばす
    ORG     8
START
    CLRF    HOGE            ;HOGE を 0 にする
    MOVLW   B'11111111'     ; 代入用
    BSF STATUS,5            ;Bank 1 へ切り替え
    MOVWF   TRISB           ;PORTB を入力に設定
    BCF     STATUS,5        ;Bank 0 へ切り替え

    BTFSC   PORTB,5         ; もし PB5 が 1 であれば FIN へ
    GOTO    FIN

    MOVLW   100             ; 足し算用 100
    ADDWF   HOGE,1          ;HOGE=HOGE+100
```

```
; 終わり
FIN
    GOTO    FIN             ;後ろに抜けないようにする
    END
```

ここで、これまでの経験上、つまりやすいマイコンの罠について箇条書きする。

- 電源の投入確認！これはよくある。また電力が足りなかったり、ポートから電流を取り出しすぎていないか注意する。
- 接触不良。ライタと回路の間を行ったりきたりしている場合は特に注意！
- **オープンドレイン**ポートに注意！オープンドレインポートは出力が H と L ではなく、ハイインピーダンス (高抵抗) と L である。
- 割り込みに注意！割り込みは条件がそろうと突如としてプログラムカウンターが別の場所を指す。不要な割り込みは禁止すること。
- グローバル変数の定義時に初期値を代入しない。リンカの設定によっては ROM に焼付けられ値の変更ができなくなる。
- PIC の場合 bank という概念があることに注意。GOTO で飛べるアドレス空間が狭い。

これから勉強する人は注意しよう！

おわりに

6 年間、私はコンピュータ部で様々なものを学んできました。雑多なまとめ方になってしまいましたが、コンピュータは系統的に勉強しても何もおもしろくありません。縦横無尽にいろんなことに手を出すと思わぬ発見がいろいろとあります。この文章がこれからの学生の参考となれば幸いです。

参考文献

[1] 柴山 潔、1997、 「コンピュータアーキテクチャ」 オーム社

[2] 桜庭 一郎/大塚 敏/熊耳 忠、1986、 「電子回路」 森北出版株式会社

[3] 株式会社ルネサステクノロジ ホームページ
http://www.renesas.com/

[4] オークス電子 ホームページ
http://www.oaks-ele.com/

2 包狩

京都大学大学院 情報学研究科システム科学専攻 2 回生 越本 浩央

2.1 はじめに

自己紹介から始めようと思います. _という実にふざけた 1 文字アカウントを取ったのは私です. 言語マニアで学部時代は 20 以上のプログラムんぐ言語を覚えてたけど. 最近はぼけて大半が使えなくなってるのは私です. Python まんせー, とか,haskell まんせー, とか叫んでるのは私です (けどこれはオレ以外も言ってるじゃん). CG 検定 1 級なんていう怪しい資格持ってるのは私です. 御堂君が 2005 年現在所属している通称ディズニーランドに, 第一で配属希望を出したのは私です. 通称ディズニーランドで初めて (?) 忘年会を開催したのは私です. アレケに三日目だけフラっと現れてサッと去って行くのは私です. それから一時期部長だったけど, 他の優秀な人が色々やってくれて, 特に何もしなかったのは私です. ちなみにタコチャーハンはその時の遺物です.

ヘモグロビンの影響かどうか知りませんが, Lime は毎年ちゃんと書いてたりします. 去年は裏 Lime でした. 昔の文書を読むと核スイッチを押したい衝動に駆られますが, こういう恥を作れるのは学生の間だけだと思うので, 今年も懲りずに書いてみようと決心して筆を取りました. 今回はいつもにも増して忙しいのですが裏も書いてます. 裏はいつものノリなので, 興味がある方はそちらもどうぞ.

さてこの記事では,OpenBSD 上で pf を使った防壁構築をれくちゅあします. OpenBSD というのは元 NetBSD デベロッパだった Theo de Raadt を中心に, あらゆるコードの安全性を検証して実装されている*BSD の派生です. 実は, 他の*BSD がセキュアでないということではなくて, Theo が NetBSD 開発 ML で喧嘩して脱退し, 新たに自分の砂場を作るために生まれました. だから非常に独善的なシステムだと言えるかもしれません. ところがこのプロジェクトは成功しました. 既にバージョンは 3.8 です. 今回のリリースでは RAID ユーティリティが実装されています. また最新のセキュリティ技術の多くがこの OS 上で実装されています. その後に他の OS に移植されているツールも数多くあります. というわけで OpenBSD は趣味の OS などではなく, むしろプロ指向の大変優秀なフリーの OS だということです. 数多のツールの中でもとりわけ OpenBSD と蜜月の関係にあるのが pf です. パケットフィルタリングツールであり, いわゆる Firewall を実現します. 今回はこのツールの使い方を通して, 多機能な Firewall の構築を説明したいと思います.

話を始めるにあたっていくつか確認しておきます. 入門記事のつもりで書いているヌルい文章ですが, Internet などの初歩的な内容を既に知っていることを想定しています. 少なくともパケットフィルタリングが何か程度は調べてからお読み下さい. また*BSD を始めとする UNIX 系 OS の設

定を経験していることが望ましいです. あと題名が今再放送してる深夜アニメに似ていることは本文と一切関係ありませんが何か?

2.2 設定篇

OpenBSD インストール直後は, まだ pf が機能していません. /etc/rc.conf の中に pf=NO という項目があるので, これを pf=YES と書き換えます. 再起動後に /etc/pf.conf の内容を読み込み,Firewall が機能します. その場で pf を有効にしたい場合は pfctl -e を実行します. 特定の conf ファイルを読み込みたい場合は pfctl -f ./pf.conf のようにします. 毎起動時に読み込むファイルを特定するには /etc/rc.conf の中で pf_rules=/var/pf.conf のように書きます. pf の制御は conf の内容以外すべて pfctl を使います. 詳細は man を見て下さい.

2.2.1 ブリッジの設定

Firewall の構築に際して, どんなネットワークに, どう配置するのか, ということに決まりはありません. しかし, パケット管理を通してセキュリティを実現するプログラムであることを考えれば, 守りたい領域の内と外を分つ境界上に設置されるべきでしょう. そして越境するパケット全てに目を光らせ, ルールを適用させなければなりません. このため,Firewall 単体をネットワークにぶら下げるのではなく, ブリッジと一緒に実行し, パケットを橋渡しするときにフィルタリングするようにします.

ブリッジの設定には brconfig コマンドを使います. 例えば ne1 と ne2 というインタフェースを追加したい場合は次のようにします.

```
> brconfig hoge add ne1 add ne2 up
```

ここでの hoge はブリッジの名前です. インタフェースを削除したいときは add の代わりに delete です. またブリッジ機能を止めたいときは brconfig down を実行します. 毎回起動時にブリッジ設定を手打ちするのは面倒です. 設定ファイル/etc/bridgename に書き込みましょう. 上述の例と同様の設定は次の通りです.

```
add ne1
add ne2
up
```

2.2.2 pf.conf の編集

Firewall の基本機能はパケットフィルタリングです. 先に書いた通り, この動作の設定は pf.conf ファイルに記述します. プレインテキストなので vi なり ed なりで編集します. 内容はフィルタルールの列挙で, ファイルの上から順に適用されます. 特別な構文などは存在せず,1 行ごとにルールの指定をする形となります. しかし pf には様々なルールが実装されており, これから述べる通り多機能な Firewall を実現します.

簡単なルールファイルの例は次のようになります.

```
block in on ne1 from any to any
pass in on ne1 from any to any port 8080
pass out all
```

非常に簡単なので説明が無くても想像出来てしまいますが, この設定では ne1 に対して 8080 番以外のポートからのパケットを返答無しに弾きます. 一方全てのインタフェースについてパケットは何でも通します. 1 行目のルールでは ne1 について in を全てブロックしていますが, 2 行目の記述で修正されていることに注意して下さい. この例自体にはあまり実用性がありませんが, こういったルールを組み合わせることで Firewall を構築します.

ルールを書き換えた後はファイルの読み込みを忘れないで下さい. 現在のフィルタの一覧は次のコマンドで確認出来ます.

```
> pfctl -sr
```

2.3 記述篇

フィルタルールの記述方法を説明します. 非常に簡単な設定ファイルであると言いましたが, 幾らかの便利な書き方が存在し, 保守を容易にしてくれます. 中には応用上必要となる記述方法もあるのでよく理解してください.

2.3.1 マクロ

プログラミングを知っている人に説明は不要だと思いますが, 定文字列を複数箇所のルールに展開するために利用します. ご想像の通り, 何カ所にも記入する同じ文字列を一カ所にまとめることで, 特にメンテナンス作業を楽にしてくれます.

```
ext_if = "ne1"
block in on $ext_if from any to any
pass in on $ext_if from any to any port 8080
```

2.3.2 リスト

同じルールを複数のインタフェースやアドレスに適用したい場合, リストを利用すれば 1 つのルール記述で済ますことが出来ます. 前述のマクロにもリストは利用出来ます.

```
ng_addrs = "{192.168.1.1, 192.168.1.2, 192.168.1.3}"
block in on ne1 from $ng_addrs to any
```

マクロをリストに展開するときは注意が必要です.

```
host1 = "192.168.1.1"
host2 = "192.168.1.2"
hosts = "{" $host1 $host2 "}"
```

2.3.3 テーブル

リストと似ていますが, 全く異なります. テーブルはアドレスのセットにしか使えませんが, リストと比較して少ないメモリと CPU タイムを消費します. 大規模なアドレス群を高速に捌きたいときはテーブルを使用します.

```
table <goodguys> const { 192.168.1.0/24 }
table <badguys> { !192.168.1.0/24 }
table <spammers> persist file "/etc/spammers"
block in on ne1 from <badguys> to any
pass in on ne1 from <goodguys> to any
block in on ne1 from <spammers> to any
```

2 行目に注目するとエクスクラメーションマークが見つかります. これは指定したアドレスに対して排他的な領域にマッチします. またテーブルには const/persist の修飾子を付けることが出来ます. const はテーブルの内容を固定し, 動的な変更が出来なくなります. persist はテーブルを恒久的に保持し, 適用されるルールが無くても維持されます. このような修飾子は pfctl によるフィルタルールの動的制御を前提としたものです. 例えばテーブルの内容を追加, 削除, 閲覧するには次のようにします.

```
> pfctl -t spammers -Tadd 10.1.2.0/24
> pfctl -t spammers -Tshow
> pfctl -t spammers -Tdelete 10.1.2.0/24
```

2.3.4 アンカー

動的なルールの制御を行う場合, サブルールセットを評価します. サブルールセットを参照するにはメインのルールセット中にアンカーを記述します. アンカーを記述した部分には別ファイルを読み込んだり, pfctl によってルールを設定したりすることが出来ます.

```
block on ne1 all
pass out on ne1 all
anchor goodguys
load anchor goodguys:ssh from "/var/anchor-goodguys-ssh"
```

この記述があるファイル (メインルールセット) を読み込んだ場合, /var/anchor-goodguys-ssh (同様のフィルタルールが記述されている) を読み込み, その内容を goodguys に ssh というサブルールで連結します. load コマンドを使わず,pfctl を使って動的に読み込ませることも可能です.

```
> echo "pass in from 192.168.1.1 to any port 22" ¥
  | pfctl -a goodguys:ssh -f -
```

ファイルを動的に連結することも可能です.

```
> pfctl -a goodguys:ssh -f /var/anchor-goodguys-ssh
```

サブルールセットを読み込む場合は, マクロやリストのスコープが異なることに注意して下さい. メインルールセットで定義されている内容を参照することは出来ません. 逆も同じです.

なおアンカーには他に 3 つ存在します. それぞれが後述する別のルール (つまりフィルタ以外) に対するアンカーです.

- nat-anchor
- binat-anchor
- rdr-anchor

2.3.5 タギング

pf ではルールを適用したパケットにタグを付けることが出来ます. タグが付けられたパケットはタグ付済みのルールを適用することが出来ます. 処理手順に対して階層的なフィルタリングを行うことによって, より高度なセキュリティ概念を構築することが可能になります. このようなアプローチをポリシーベースフィルタリングと言ったりします. (つまりパケットごとにタグ付けをするルールと, タグごとにパケットを処理するルールです)

タギングは例えば次のように行います.

```
pass in on ne1 tag INET_HTTP keep state
pass out on ne1 tagged INET_HTTP keep state
block out on ne1 tagged ! INET_HTTP keep state
```

tag オプションによってタギングし,tagged オプションでタグのマッチを行います. タグのマッチにはアドレス表現で用いた”否定”も利用可能です. ここでは示されていませんが, タグが多重で付けられた場合, 最後に付けられたタグのみが有効となります.

タギング時に注意して欲しい点は keep state というディレクティブです. ステートフルなフィルタリングを行いたい場合,Firewall はルールの適用に際して, パケットが一連の接続の一部であることを認識しなければなりません. 例えば後述する NAT やリダイレクションがそうです. これらは暗黙的にで keep state されています. タギングする場合も同様です. 仮に keep state しなければシーケンスナンバーのチェックに引っかかり, キューからドロップされてしまいます.

ステートフルなフィルタリングを行う場合には更に検討すべき項目があります. SYN フラグを Firewall の側で明示的に確認しなければならない状況です.

```
pass in proto tcp all port 80 flags S/SA keep state
pass out proto tcp all flags S/SA keep state
```

逆に SYN フローディングを回避したいという要求も存在します. これには 2 通りの手段があります.

```
pass in proto tcp all port 80 flags S/SA modulate state
pass in proto tcp all port 80 flags S/SA synproxy state
```

2 つのルールは同じ目的を達成しますが, 前者は乱数シーケンスを用いてスプーフィングを回避し, 後者は Firewall 内で観察されるコネクションを張ります (つまりプロキシーとして機能します). synproxy は Firewall 上で keep state と modulate state を実行すると考えて下さい.

2.3.6 セットステート

Firewall 上でコネクションの状態を設定することが出来ます. 具体的には一度に保持するステートの数と, コネクションに許容するタイムアウトの長さです. これらはルールファイル内で明示的に設定することが可能です.

```
limit states 100
set timeout tcp.first 20
set timeout tcp.opening 20
set timeout tcp.established 10
set timeout tcp.closing 10
set timeout tcp.finwait 10
set timeout tcp.closed 10
```

さらにこれらの値を適応的に設定することが可能です. これには処理する下限と上限を設定し, その間を適応させることになります.

```
set timeout {adaptive.start 5000, adaptive.end 20000}
```

この設定値は次のように適応値を導出します.

$$\frac{adaptive\_end - number\_of\_states}{adaptive\_end - adaptive\_start} = scaling\_factor$$

2.3.7 ルール

フィルタルールの完全な書式は次の通りです.

```
action direction [log] [quick] on interface [af] ¥
    [proto protocol] from src_addr [port src_port] ¥
    to dst_addr [port dst_port] [tcp_flags] [state]
```

action にはルールの処理が入り, パケットの方向が direction で示されます. quick が指定されると, 後続するルールには関係なくそのルールが有効になります. af にはアドレスファミリとして inet か inet6 を指定出来ます. proto ではプロトコルを限定することが出来ます. これまでの例では説明しませんでしたが, アドレスの指定には否定の他に不等号による範囲も利用出来ます. port にはポート番号 (リストでも構いません) を設定出来ます. tcp_flags では check/mask の形式でフラグをチェックします. 先の例に出て来た S/SA なら,SYN or ACK フラグをマスクし, SYN フラグを確認するようになります.

2.4 基本篇

前節では詳細なルールの記述方式まで説明しました. ここからはより実践的なフィルタルールの例を示して行きます. 本節での対象は Firewall の基本機能が中心となります. しかしここを見るだけでも pf の高機能性を確認することが出来るでしょう.

2.4.1 ノーマリゼーション

パケットは彼岸からやってくるときに断片化されることがあります. 意図にしろそうでないにしろ. このようなパケットの取り扱いは厄介になります. パケットに限りませんが, 不十分な情報に基づく計算は非決定性問題というクラスに属し, 様々な問題を孕むことになります. pf ではこのようなパケットの再構築を行い, それが出来ないもの, 不正確なフラグを持つものを破棄します.

```
scrub in on ne1 all no-df
```

ne1 に入って来る全てのパケットを断片化します. no-df オプションは NFS のような断片化禁止ビットを持つパケットに対して, そのビットを取り去ります. scrub には他にも多くのオプションが存在します.

- random-id 識別フィールドを乱数に置換
- min-ttl num TTL の最小値を強制
- max-mss num MSS の最大値を強制
- fragment reassemble バッファ上で再構築 (デフォルト)
- fragment crop 二重断片化を破棄
- fragment drop-ovl 後続する重複も破棄 (デフォルト)
- reassemble tcp 方向が欠損したら TTL を最大で時刻を乱数

2.4.2 リダイレクション

pf にはパケットのルーティングを実現するリダイレクションの機能があります. これには 3 つのルールが存在します. 通信時に片側のアドレスを別のアドレスに書き換える NAT. 特定のノードへのルーティングを行うリダイレクト. 外部内部両側のアドレスを変換する BINAT. 前述した通り, これらは Firewall のステートフルコネクション上で機能します. 但し, わざわざ明示的に keep state を指示する必要はありませんし, してはいけません. またとても重要なことですが,NAT 自身の処理はフィルタルールより前に実行されます. そのため, 変換後のパケットがマッチするルールは最初のものとは異なります.

```
nat on en1 inet proto tcp from $prv_ad to any ¥
 -> $ext_ad port 10000:20000
```

単純に NAT を指示するルールです. アドレス変換だけでなくポートの割当も行えます. この例ではポートとして 10000 から 20000 の値が割り振られます. ところで繰り返しですが, 変換後にフィルタされます. これを回避したい場合, nat キーワードの直後に pass オプションを付けます. これでフィルタをバイパスします.

```
binat on ne1 from $web_int to any -> $web_ext
```

外部から内部へのパケットの変換だけでなく, 逆向きにも変換を施したい場合は BINAT を使います. これは内と外の異なるアドレスを 1 対 1 対応付けることを意味し, 例えば内部のサーバを外部のサーバとして見せかけることが可能です.

なお pf 単体では NAT の機能が有効になりません.sysctl を使って直接設定するか, /etc/sysctl.conf を編集します.

```
> sysctl -w net.inet.ip.forwarding=1
> sysctl -w net.inet6.ip.forwarding=1
```

NAT 処理の背後ではリダイレクションが機能しています. 外から来るパケットをマッチさせ, 適切な内部ネットワークへと再送します.

```
rdr on ne1 proto tcp from 10.5.1.0/24 to any port 80 ¥
 -> 192.168.1.1
```

この例では二つのプライベートネットワーク間で, 参照すべき HTTP サーバを設定しています. 中規模のイントラネットなどを想像すると丁度良いかもしれません. もちろんプライベートネットワークに限らず,SOHO のような形で外のマシンから参照させたいホストにも, より限定的にアドレスのマッピングを行うことも可能です.

逆のことも考えてみましょう. 二つの内部ネットワークから DMZ への参照は, 例えばこんな感じです.

```
rdr on {ne1, ne2} inet proto tcp from any to $web_ext port 80 ¥
 -> $web_ad port 8080
```

2.4.3 フィルタリング

当然のことながら pf にはフィルタリング機能があります. フィルタルールは quick が指示されない限り, 上書きされます. それを利用して次のような書き方をすることが可能です.

```
block in all
pass in on ne1 from any to any port >= 8000 os "Windows"
```

どう使うのかは微妙ですが,OS のフィンガープリントを確認することも出来ます. この例では Windows というエントリをマッチします. このエントリは/etc/pf.os の中で定義されていなければなりません. ちなみに上の例では通過させるポートとして 8000 番以上のものに限定しています.

フィルタリングはリダイレクションと併用することも可能です. 前項で説明したように, 変換テーブルの処理はフィルタより前に行われます. これを利用することで, パケットの分類とフィルタルールを分けて考えることが出来ます.

```
rdr on ne1 proto tcp from $prv_ad to any port {80, 8000} ¥
 -> $ch_ad port 8080
pass in on ne1 proto tcp from $prv_ad to $ch_ad port 8080
```

こうすることで 80 番や 8000 番のパケットをまとめて 8080 番としてフィルタ出来ます. もちろん 8080 番は 2 行目のルールとマッチします. これもある種のポリシーセキュリティと言えます. サブルールを使えばさらに柔軟な Firewall を構築可能です.

2.5 応用篇

パケットフィルタリング機能だけでなく,pf には更に高度な機能が存在します. これらの機能は前述の様々なルールと一緒に記述し, 非常に強力な Firewall の構築を助けてくれます. OpenBSD と pf の組み合わせがとても実践的なソリューションであることを見て行きましょう.

2.5.1 シェーピング

特定のポートやプロトコルに対して帯域を制限することが可能です. この技術をシェーピングとかキューイングと言い,pf では 3 種類のアルゴリズムが利用出来ます. OS がプロセスを処理する過程を思い起こして下さい. プロセスは実行待ちキューに格納されます. 割り当てられた CPU タイムを使い切ると, 実行中のプロセスはキューの後ろに回され, 先頭のものが実行されます. 実践的な OS では複数のレベルを持つ階層的なキューを持つことで, 速いレスポンスが必要なもの, 長時間に渡って処理が行われるもの, を区別して処理します. 同じことがネットワーク上のパケットでも言えます. pf ではルールファイルの中でキューを定義し, パケットをどのキューに入れるかを選択出来ます. さらにキュー上のパケットを処理するスケジューラアルゴリズムが選べます.

まずキューを定義します. キューには名前を付けて, スケジューラを含めたオプションを設定します.

```
queue std bandwidth 50% cbq(default)
queue ftp bandwidth 50% priority 3 cbq(red)
queue dmz bandwidth 10Mb cbq(red) queue {std, ftp}
```

ここでは std と ftp というキューを定義しました. 帯域幅はご覧の通り複数の指定法が使えます. 行の最後で利用するスケジューラを設定しています.cbq とは Class Based Queueing のことです. このスケジューラは階層的に帯域の割当を行います. この例では dmz というキューが親となり, 親に割り当てられた帯域 10Mb を std と ftp が半分づつ利用する形となります. また帯域幅だけでなく, キューの優先度を設定することが可能です.cbq では 0-7 の値を選んで下さい. なお最後に指定されている red キーワードは, キューが溢れる前にキューの平均長を調節することで, 輻輳を回避するよう指示します. さて, ここで定義した queue はインタフェースと対応づけなければなりません.

```
altq on ne1 cbq bandwidth 10Mb qlimit 100 queue dmz
```

割り当てるキューは幾つでも構いませんが, ここで設定された帯域を越えての接続は実現されません. そして最後に, どのフィルタルールにマッチしたものがどのキューに入るのかを書きます.

```
pass in on ne1 from any to any queue std
pass in on ne1 from any to any port {20, 21} queue ftp
```

pf で実装されているスケジューラは cbq だけではありません. 優先度だけに基づいてスケジューリングするものに priq(Priority Queueing) があります.

```
altq on ne1 priq bandwidth 50Mb ¥
 queue {dns, ssh, www, mail, other}
```

```
queue dns priority 14 priq(red)
queue ssh priority 13 priq(red)
queue mail priority 12 priq(red)
queue www priority 11 priq(red)
queue other priority 10 priq(red)
```

priq では単純にキューごとの優先度が設定出来るだけです. cbq のような階層的な制約は存在しません.

もう一つのスケジューラとして hfsc(Hierarchical Fair Service Curve) が利用出来ます. cbq と同様にキューの関係を階層化して帯域制限をかけることが出来ます. さらに hfsc ではキューごとに柔軟な帯域確保を行うことが出来ます.

```
altq on ne1 hfsc bandwidth 45Mb queue dmznet
queue dmznet hfsc(realtime(35% 5000 20%) ¥
 linkshare(50% 5000 20%) upperlimit(60% 5000 30%))
```

3 つのオプションによってキューを制約します. realtime オプションは絶対確保する容量です. 割当帯域の 20%から 35%に対して,5000 ミリセカンド以下の応答を保証します. linkshare は同じレベルのキューへの貸出しを許可する帯域幅です. upperlimit は自分の確保した分と他から貸し出された分の合計に対する制限です. つまりこの dmznet キューの帯域は次のような制約に従います.

$$band = realtime + linkshare \leq upperlimit$$

2.5.2 ロードバランシング

DNS サーバには応答をラウンドロビンで選択する仕組みがありますが, pf でも同様の機能を利用することが出来ます. またこれまでの説明でも明らかなように, リダイレクション等の機能と組み合わせることで, DNS と同様の機能が実現出来ます. もちろんテーブルの動的書き換えは可能です.

pf のロードバランシングではラウンドロビン以外にも色々な指示が可能です. 順番に見て行きましょう.

```
rdr on ne1 proto tcp from any to $ext_ad port 80 ¥
 -> 10.1.1.1/24 round-robin
rdr on ne1 proto tcp from any to $ext_ad port 80 ¥
 -> {10.1.2.1, 10.1.2.2, 10.1.2.3} random
rdr on ne1 proto tcp from any to $ext_ad port 80 ¥
 -> 10.1.3.6/29 source-hash
```

1 つ目の例はラウンドロビンそのものです.2 つ目はランダムにリダイレクトします. この両者のアプローチであっても, 同じホストの要求に毎回同じ応答を返すことが出来ません. 同じホストからの問い合わせを確認したい場合は,3 つ目のようなハッシュ値による対応を行います.

ロードバランシングのために動的なマッピングを必要としない場合もあります. アドレス群を, 並びを維持して他のアドレス群にマップするときは bitmask を使います.

```
rdr on ne1 proto tcp from 192.168.1/24 to any ¥
 -> 10.4.3/24 bitmask
```

このルールでは, 例えば 192.168.1.1 は 10.4.3.1 に, 192.168.1.2 は 10.4.3.2 にリダイレクトされます.

2.5.3 ロギング

最後の機能としてログの取り方を説明します. これはルールごとに設定が可能で, 次のようにします.

```
block in log on ne1 all
```

他の様々なルールでも同様に log ディレクティブでログが取れます. なお pf のログファイルはバイナリなので, ログを読むときには注意が必要です. これには tcpdump を使います. 例えば,

```
> tcpdump -r /var/log/pflog -i ne1
```

というようにしてログを参照します. このままでは全ての情報が出て来ますので, ポートやアドレスを指定して見ると便利です. 詳しくは tcpdump の man を参照してください.

ところで, 既に pf の Firewall は pfctl を用いることで動的に変更出来ると言って来ました. 具体的な方法も例を挙げました. さて, これにログの観察を加えるとより強力な Firewall が構築出来ます. 例えば, 定期的にログを観察し, 危険なパケットを検出します. 多くの場合, そのようなパケットは単独ではなく, これまでに見られなかった特異なパターンを成しているでしょう. ログの分析には文脈依存文法が必要になるかもしれません. ともかく攻撃が検知出来たとします. あとはそれをフィルタしたり, 場合によっては適切な応答を返すためにルールセットを書き換えます. ここでアンカーを利用し, フィルタやリダイレクションのサブルールを追加します.

2.6 おわりに

最後になったので白状します. この文書を書き始めた当初の目的は, 攻殻機動隊に登場するような攻性防壁を構築してみよう, というものでした. 既に (時間的な) 余白がありませんので, 攻勢防壁を作るのは読者の宿題としておきます.

たぶん今回の文書が私にとって最後の Lime になるのではないかと思います. 最後に味も素っ気も無いリファレンスのような記事を書いたのは, 当然今述べた理由もあるのですが, 現役として活動される部員の方達の生活環境の助力になればと思ったからです. 別に Windows を使うのが悪いというわけではありません. いやむしろ Linux だけではどうかと思いますが. 若い頃に考えている以上に世の中には色々な技術が存在し, 同時に様々な活躍の場面が存在します. 技術力以前に, これらの状況を体験しないことは悲劇としか言い様がありません. もちろん, 知らない世界へ足を踏み出すことは難しいと思います. そんなときに先達や先輩たちの文章が助けになると思います. 願わくば, この文章が誰かの興味を支える一役とならんことを...

3 アプリケーションプログラムにキーボードの入力が届くまで

電子情報工学科 1 回生 林 奉行

3.1 はじめに

アプリケーションプログラムを書くとき、キーボードからの入力は標準関数から知ることができます。キーボードに限らず入出力をするプログラムを書くとき、実際にデバイスが何をしているのかを意識する必要はあません。それはしかし大変便利な反面各地に大小のブラックボックスを生成することになるでしょう。そこで、実際にハードウェアや OS はどのように入出力制御しているのかをキーボードの入力があってからその入力がアプリケーションプログラムによって検知可能になるまでの処理を追ってみることにしました。

キーボードのキーが押されてからアプリケーションプログラムに届くまでには大まかに次のようなプロセスを踏みます。

1. キーボードのキースイッチが押される
2. キーボードはキースキャンコードをキーボードコントローラに送る
3. キーボードコントローラが割り込みコントローラに割り込み信号を送る
4. 割り込みコントローラが CPU に割り込み要求をする
5. CPU で割り込みが発生する
6. 割り込みハンドラに処理が渡される
7. 入力待ちのアプリケーションに処理が渡される
8. どのキーが押されたかを調べる

3.2 キースイッチが押される

一般的なキーボードはキーを押すと直下の電極が接触して通電し、キーが押されたことを感知する仕組みになっています。キー１つ１つにはスキャンコードと呼ばれる ID 番号が割り振られてい

ます。キーボードはキーが押されたことを感知するとメインボード上にあるキーボードコントローラというチップと通信してそのキーのスキャンコードを送信します。

3.3 キーボードはキースキャンコードをキーボードコントローラに送る

キーボードコントローラはメインボード上にある 1ChipCPU で CPU から見ると I/O として動作します。キーボードからはキーが押されたときは Make コード、放されたときには Break コードが発行されます。キーを押し続けると Make コードが連続して発行されます。

3.4 キーボードコントローラが割り込みコントローラに割り込み信号を送る

キーボードコントローラは Make コードや Break コードを受け取ると割り込みコントローラ (PIC[1]) に対して割り込み信号が出されます。

3.4.1 割り込みについて

『ヘネシー&パターソン コンピュータの構成と設計』[2]によると割り込みとは CPU 外部から引き起こされる制御の流れに予期せぬ変更を生じさせる事象です。具体的には周辺機器からの割り込み要求によって CPU は割り込みを発生させます。割り込みが発生すると CPU は現在の処理を中断して割り込み要求に応じた別の処理を行います。割り込みを処理する機能によって CPU はキーボードやマウスなどの周辺機器の変化に素早く反応することができるようになります。また、割り込みを利用することでプロセッサの処理能力を有効に利用することができます。周辺機器の処理速度は CPU のそれより格段に遅いため、その入力を待っているとせっかくの CPU の演算能力が無駄になってしまいます。例えば、ディスプレイに「a を入力してください。」と表示されてからキーボードで'a' と入力するまでに 0.5 秒かかったとすると、その間にプロセッサは Intel の Pentium を例に取ると 4000 万回以上 (筆者の PC による実測) [3]もの実数演算をすることができます。周辺機器の変化を割り込みによって CPU に通知することで CPU は入力があるまで他の処理を実行することができ、CPU の処理能力を有効に利用することができます。

[1]Programmable Interrupt Controller

[2]David A.Patterson/John L.Hennessy 著　成田光彰訳　日経ＰＢ社

[3]

ベンチマークソフト	姫野ベンチ (Win 版 実行形式 Sサイズ)
CPU	pentium　450MHz
メモリ	PC100 192MB
OS	Windows2000 Professional
実測結果	82.95687MFLOPS

3.5 割り込みコントローラが CPU に割り込み要求をする

キーボードコントローラは入力があると割り込み信号を出しますが CPU に直接つながっているわけではなく、割り込みコントローラを介します。割り込みコントローラは複数のハードウェアからの割り込み要求信号を中継する役割を持ちます。PC では CPU が割り込みを検知する線は１つしかありません。割り込みコントローラは複数の割り込み要求があったときにその優先度に基づいた順番で割り込み要求を CPU に伝えます。例えば、優先度の高い割り込み処理が実行されているとき優先度の低い割り込み要求があると優先度の低い割り込み要求を現在の割り込み処理がおわるまで待たせたりします。つまり、割り込み要求を管理します。

キーボードコントローラから割り込み要求があった場合、割り込みコントローラはその割り込みが許可されているかを確認します。許可されていれば CPU に割り込み要求信号を出します。

3.6 CPU で割り込みが発生する

割り込みコントローラから CPU に割り込み要求が出されると、CPU で割り込みが発生します。CPU も割り込みが許可されているか確認し、許可されていれば現在の状態を保存[4]します。それから割り込みベクタと呼ばれる割り込みハンドラのアドレス一覧が記録されたメモリ領域を参照して割り込み要求をしたデバイスに応じた割り込みハンドラのアドレスをしらべ、そのアドレスへジャンプします。

3.6.1 割り込みハンドラ

CPU で割り込みが発生したとき、現在実行中の処理を一時中断し、割り込みを要求したデバイスに応じた処理を行った後で再び中断していた処理を再開します。この処理を中断して呼び出されるプログラムのことを割り込みハンドラと呼びます。Linux や Windows の場合割り込みハンドラはドライバの機能の一部です。

3.7 割り込みハンドラに処理が渡される

キーボードの場合はキーボードドライバの割り込みハンドラが呼び出されます。キーボードドライバはキーボードコントローラと通信してキーの値を得て、その値をメモリのシステムエリアに格納して終了します。

3.8 入力待ちのアプリケーションに処理が渡される

OS からキーボード入力待ちのプログラムに処理が渡されるとそのプログラムは先ほどキーボードドライバが格納したキーの値を読み込みます。ここで、アプリケーションプログラムはキーボー

[4]プログラムカウンタやフラグレジスタなどの値をスタックなどに保存する

ドからのキー入力を受け取ったことになります。しかし、ここで受け取ったキー入力の情報はキーボードの'a' や'b' と言った文字コードではなく先ほどのキーのスキャンコードです。なので、変換する必要があります。

3.9 おわりに

今回アプリケーションプログラムにキーボードの入力が届くまでの処理を追ってみて、これだけ多くのプロセスを踏んでいるとは、と驚きました。(何か間違いがあるのでは、っとはらはらしていますが。) これからコンピュータを勉強していく足がかりになれば良いと思います。

参考文献

[1] David A.Patterson/John L.Hennessy、1996、 「コンピュータの構成と設計」 日経ＢＰ社

[2] テクノベイン社ウェブページ
http://www.technoveins.co.jp/

[3] Eric Raymond, JF Project 訳、「The Unix and Internet Fundamentals HOWTO」
http://www.linux.or.jp/JF/JFdocs/Unix-and-Internet-Fundamentals-HOWTO.txt

[4] japan linux.com
http://japan.linux.com/

[5] WisdomSoft
http://wisdom.sakura.ne.jp/

4 エディタを作ろう

電子情報工学科 3 回生 久保 達彦

4.1 エディタとは？

エディタとはテキスト文書を編集するソフトのことで, Windows で標準装備されているものとして「メモ帳」等があり, これを使って文書を作成したり, プログラムを記述したりすることができます。

有名なところで言うと, 秀丸や Emacs とかですかね。(自分は普段, Windows ではサクラエディタ, Linux では Emacs を使ってます。最近は専ら IDE[1]ですが)

普段何気なく使っているエディタですが, そんなエディタのプログラムを実際に書いてみようというのがこの記事のテーマです。言語は Java, GUI コンポーネントは主に Swing を使用しています。

なお, 各 API の詳細は省きます。J2SE の API 仕様[2]等を見て調べてください。

4.2 なんで Java?

まあ, Java の方が簡単に書けるかなあと思って書き始めたんですが, 実は Java は Java で面倒な部分も多々あったりします。

4.3 サンプル

サンプルには自分が趣味で書い (てい) た KUBOL エディタを用います。でも, 実装している機能は少ないです。あと, かなり稚拙なコードになってる上にバグが多々あります。でも一応ここで紹介している機能は問題ない (ハズ) です。

実物は図 1 を参照。

次にプログラムの根幹部分である MainFrame クラスを見てみます。

```
public class MainFrame extends JFrame{

    private JMenuBar menuBar; //メニューバー
```

[1]統合開発環境。エディタやコンパイラ, デバッガ等のツールを一つのインターフェースで統合して使えるような環境

[2]verson1.4→http://java.sun.com/j2se/1.4/ja/docs/ja/api/index.html
version5.0→http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/index.html?overview-summary.html

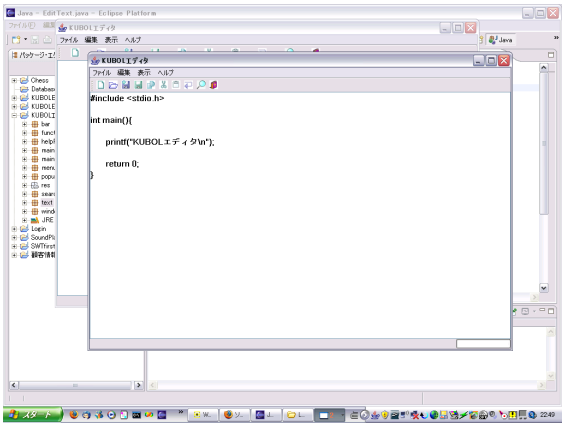


図 4.1: KUBOL エディタ

```
private FileMenu fileMenu; //ファイルメニュー
private EditMenu editMenu; //編集メニュー
private IndicationMenu indicationMenu; //表示メニュー
private HelpMenu helpMenu; //ヘルプメニュー

private PopupMenu popupMenu; //ポップアップメニュー

private EditText text; //編集領域
private JScrollPane sp; //スクロール
private LineNumberText lineText; //行の表示領域
private StatusBar statusBar; //ステータスバー
private ToolBar toolBar; //ツールバー

private static int num_mainFrame; //現在開いているエディタの数
private static Point pt;

static {
    num_mainFrame = 0;
    pt = new Point(50, 50);
}

//コンストラクタ
public MainFrame(){
    super("KUBOL エディタ");
    num_mainFrame++;
    System.out.println("現在開いているエディタの数" + num_mainFrame);

    lineText = new LineNumberText(); //行の表示領域
    text = new EditText(lineText); //編集領域

    menuBar = new JMenuBar(); //メニューバー
```

```
        popupMenu = new PopupMenu(text, this); //ポップアップ

        fileMenu = new FileMenu(this, text); //ファイルメニュー
        editMenu = new EditMenu(this, text); //編集メニュー
        indicationMenu = new IndicationMenu(this, popupMenu); //表示メニュー
        helpMenu = new HelpMenu(this); //ヘルプメニュー

        sp = new JScrollPane(text);

        statusBar = new StatusBar(text);
        toolBar = new ToolBar(this, text);

        pt.x += 50;
        pt.y += 50;

        construct();
    }

    //部品の組み立て
    private void construct(){

        menuBar.add(fileMenu);
        menuBar.add(editMenu);
        menuBar.add(indicationMenu);
        menuBar.add(helpMenu);

        //コンテナ
        Container c = getContentPane();
        c.add(lineText, BorderLayout.WEST);
        c.add(sp);

        c.add(statusBar, BorderLayout.SOUTH);
        c.add(toolBar, BorderLayout.NORTH);

        addWindowListener(new WindowClosing(this));

        setJMenuBar(menuBar);
        setBounds(pt.x, pt.y, 500, 500);

        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        setVisible(true);
    }

    public int getNumMainFrame(){
        return num_mainFrame;
    }

    public void dispose(){
        num_mainFrame--;
        super.dispose();
    }
}
```

```
```

ここでやっているのは, メニューバーやツールバーなどのコンポーネントを MainFrame の上に載せています。JMenu クラスを継承した FileMenu クラスや JPopupMenu を継承した PopupMenu クラスを初期化してそれらを add メソッドで MainFrame の上に載せるわけです。(実際にはコンテナですが) ツールバー等もこのように MainFrame 上に載せていきます。

ちなみに main メソッドで行っている処理はこれだけです。

```
    MainFrame mainFrame = new MainFrame();
```

4.4 機能

とりあえず, 執筆時で実装している機能は以下のようなものです。

・ファイル関連の機能

- 新規作成
- ファイルを開く
- 名前を付けて保存
- 上書き保存

・編集関連の機能

- コピー
- 切り取り
- 貼り付け
- 削除

まあ, エディタならこれぐらいの機能は付いていて当たり前ですね。むしろ少ないくらいです。

4.4.1 ファイル関連の機能

ファイル関連の機能の実装は以下の通りです。

```
    //FileFunction.java

    public class FileFunction{
```

```
    private static File file; //ファイル

    //ファイルを開く
    public void open(MainFrame mainFrame, JTextArea text){
        JFileChooser chooser = new JFileChooser();
        String line;

        if(chooser.showOpenDialog(mainFrame) == JFileChooser.APPROVE_OPTION){

            text.setText("");

            file = new File(chooser.getSelectedFile().getAbsolutePath());

            try{
                BufferedReader br = new BufferedReader(new FileReader(file));
                while((line = br.readLine()) != null){
                text.append(line + "¥n");
                }
                br.close();
            }
            catch(IOException ioe){}
        }
    }

    //名前を付けて保存
    public void write(MainFrame mainFrame, JTextArea text){
        JFileChooser chooser = new JFileChooser();
        String line;

        if(chooser.showSaveDialog(mainFrame) == JFileChooser.APPROVE_OPTION){
            try{
FileWriter fw = new FileWriter(chooser.getSelectedFile().getAbsoluteFile());
                PrintWriter out = new PrintWriter(fw);

                String s = text.getText();
                out.println(s);

                out.close();
                fw.close();
            }
            catch(IOException ioe){}
        }
    }

    //上書き保存
    public void overWrite(JTextArea text){

        try{
            FileWriter fw = new FileWriter(file);
            PrintWriter out = new PrintWriter(fw);

            String s = text.getText();
```

```
                out.println(s);

                out.close();
                fw.close();
            }
            catch(IOException ioe){}
        }
    }
```

新規作成

これは単に新しい MainFrame クラスのインスタンスを生成するだけでできます。

```
    new mainFrame():
```

ファイルを開く

open メソッドに注目してください。

呼び出されるとまず, JFileChooser の showOpenDialog メソッドでファイル選択ダイアログを表示させます。次に, text.setText("") でテキストエリアの内容を空にして chooser.getSelectedFile().getAbsolutePath() で選択ダイアログで選択したファイルを取得しています。その後, try の部分で選択したファイルを読み込んでテキストエリアに書き込んでいます。

名前を付けて保存

write メソッドに注目してください。

ここではまず, showSaveDialog メソッドを呼び出しています。

次に, FileWriter クラスと PrintWriter クラスのインスタンスを生成します。簡単に言うと, ここでどのファイルに書き込むのか指定しています。

chooser.getSelectedFile().getAbsoluteFile() とはファイル選択ダイアログで選択されたファイルのパスの絶対形式です。

その後, text.getText() でテキストエリアの文字列を取得して println メソッドでファイルにその文字列を書き込んでいます。

上書き保存

overwrite メソッドに注目してください。

コード自体は「名前を付けて保存」とほとんど一緒です。違うのは「名前を付けて保存」ではファイル選択ダイアログで選択したファイルのパスの絶対形式を FileWriter のコンストラクタに渡していますが, ここでは, 「ファイルで開く」の処理で使った File 型の file インスタンスを渡しています。

4.4.2 編集関連の機能

編集機能は簡単です。JTextArea クラスにはそのためにメソッドが用意されています。

コピー

選択した領域をクリップボード[3]にコピーするには copy メソッドを使います。つか, これだけです。

切り取り

これも cut メソッドで実現できます。

貼り付け

これもまた paste メソッドで実現できます。

削除

JTextArea クラスには delete メソッドはありませんが, replaceSelection メソッドを使えばできます。こんな風に。

```
    replaceSelection("");
```

4.4.3 編集機能のメソッドを自分で実装

ここで copy, cut, paste の実装例を紹介して内部の処理を見ていきます。

[3]複数のアプリケーションがデータを交換するために利用する共有メモリ。

コピー

```
public void MyCopy(){
    Clipboard clipBoard = Toolkit.getDefaultToolkit().getSystemClipboard();

    StringSelection contents = new StringSelection(getSelectedText());
    clipBoard.setContents(contents, this);
}
```

クリップボードにアクセスするには Clipboard クラスを使います。クリップボードには、OS が持つシステムレベルのクリップボードと Java のアプリケーション内で論理的に実装するローカルクリップボードがありますが, ここでは前者を対象にしています。Toolkit.getDefaultToolkit().getSystemClipboard() はシステムレベルのクリップボードを返すメソッドです。

次に, 文字列のクリップボード転送をサポートしている StringSelection クラスを使います。このコンストラクタに転送したい内容を渡しています。(getSelectedText() は指定した選択領域の文字列を返します。)

それから, clipBoard.setContents メソッドを使ってクリップボードに文字列をコピーしています。ちなみに this になってるのはこのメソッドを持つクラスが ClipboardOwner インタフェースを実装しているためです。

切り取り

```
public void MyCut(){
    MyCopy();
    replaceSelection("");
}
```

「切り取り」ではさっき実装した MyCopy メソッドはそのまま使えます。ようはクリップボードにコピーした後, 選択した領域を削除すればいいわけです。

貼り付け

```
public void MyPaste(){
    Clipboard clipBoard = Toolkit.getDefaultToolkit().getSystemClipboard();
    Transferable content = clipBoard.getContents(this);
       if (content != null){
             try{
```

```
            String dest = (String)content.getTransferData(DataFlavor.stringFlavor);
                        append(dest);
                  }catch (Exception e){}
            }
      }
```

コピーの時と同じく, getSystemClipboard() でシステムレベルのクリップボードを使います。getContents メソッドはクリップボードにコピーされた内容を返しています。そして, クリップボードの中身が空でなければテキストエリアにクリップボードにコピーされた文字列を書き込むわけです。

4.4.4 表示関連の機能

Java では Look & Feel[4]の変更ができるので KUBOL エディタではその機能も実装しています。使用する Look & Feel は UIManager.setLookAndFeel メソッドを使って変更します。

Windows 風の Look & Feel にするには, com.sun.java.swing.plaf.windows.WindowsLookAndFeel クラスを使います。

```
UIManager.setLookAndFeel(new WindowsLookAndFeel());
```

Java 風の Look & Feel にするには, javax.swing.plaf.metal.MetalLookAndFeel クラスを使います。

```
UIManager.setLookAndFeel(new MetalLookAndFeel());
```

ほかに Mac OS のルック&フィールである apple.laf.AquaLookAndFeel や
UNIX 風の com.sun.java.swing.plaf.motif.MotifLookAndFeel [5] があります。

4.5 ツールヒント

ツールバー上のアイコンにカーソルを合わせるとそのアイコンは説明などが表示されたりしますよね。これはツールヒントというものでこれを Java で実装するには JCompoment クラスの setTipText メソッドを使います。例えば, ツールバーの「名前を付けて保存」のアイコンでこれをやるには,

```
toolButton.setTipText("上書き保存 Ctrl+S");
```

[4]コンピュータの操作画面の見た目や操作感。
[5]UNIX の Motif→ Open Software Foundation が開発した GUI

のように書きます。toolButton は JButton 型のインスタンスです。

4.5.1 オリジナルツールヒントを作る

また, ツールヒントは自分で作ることができます。

しかし単にツールヒントを表示するだけならさきほどのように簡単にできますが, 自分で作る場合は結構面倒です。

どういうときにオリジナルのツールヒントを作るかというと, 例えばはさきほどのコードを

```
toolButton.setTipText("上書き保存 ¥r¥n Ctrl+S");
```

とやってもツールヒントは改行されません。改行するには自分でそういう機能を加えたツールヒントを自分で作る必要があります。ほかにツールヒントの背景色を変えたりするときにもオリジナルのツールヒントを作る必要があります。

改行するツールヒントを作る処理は以下のような感じです。

```
class MultiLineToolTip extends JToolTip{
    public MultiLineToolTip(){
        setUI(new MultiLineToolTipUI());
    }
}

class MultiLineToolTipUI extends MetalToolTipUI{

    private String[] strs;
    private int maxWidth = 0;

    public void paint(Graphics g, JComponent c){
        g.setColor(c.getBackground());
        g.fillRect(0, 0, c.getSize().width, c.getSize().height);
        g.setColor(c.getForeground());
        if(strs != null){
         for(int i=0;i<strs.length;i++){
     g.drawString(strs[i], 3, (g.getFontMetrics(g.getFont()).getHeight()) * (i+1));
            }
        }
    }

    public Dimension getPreferredSize(JComponent c){
        String tipText = ((JToolTip)c).getTipText();
        if(tipText == null){
            tipText = "";
        }

        BufferedReader br = new BufferedReader(new StringReader(tipText));
        String line;
```

```
            int maxWidth = 0;
            Vector v = new Vector();

            try{
                while((line = br.readLine()) != null){
    int width = SwingUtilities.computeStringWidth(c.getFontMetrics(c.getFont()),line);
                    maxWidth = (maxWidth < width) ? width : maxWidth;
                    v.addElement(line);
                }
            }
            catch(IOException ex){
                ex.printStackTrace();
        }

            int lines = v.size();
            if(lines < 1){
                strs = null;
                lines = 1;
            }

            else{
             strs = new String[lines];
             int i=0;
                  for (Enumeration e = v.elements(); e.hasMoreElements() ;i++) {
                    strs[i] = (String)e.nextElement();
                  }
            }

            int height = c.getFontMetrics(c.getFont()).getHeight() * lines;
            this.maxWidth = maxWidth;
            return new Dimension(maxWidth + 6, height + 4);
        }
    }
```

とまあ, 改行するだけなのになんか長いコードになります。というのも paint メソッドが出てきていることからわかると思いますが, 全部自分でツールヒントを描画するなんて面倒なことをしなければいけません。

オリジナルのツールヒントを作るというのはなかなか骨が折れるのでよっぽどこだわりがなければやる必要はないでしょう。(私みたいに「名前を付けて保存」と「Ctrl+S」の間には改行がなければ落ち着かないとか)

4.6 最後に

4.6.1 KUBOL エディタについて

今回, 紹介した KUBOLEditor はまだまだ機能が少なく, とても使えるレベルではありません。とりあえず, タブごとにファイルを編集できるようにするとか MDI なものにするとかキーマクロな機

能を実装するとかいろいろあると思うので今後も暇を見て改良していけたらなあと思っています。
それから, 汚いコードですがソースコードは以下の URL からダウンロードできます。
http://www.kitcc.org/ bokko/KUBOLEditor.zip

参考文献

[1] 赤坂玲音、「Java アプリケーション作成講座 Swing プログラミング徹底攻略」毎日コミュニケーションズ

[2] 高田美樹、「Java 完全マスターブック」技術評論社

5 耳コピしよ　～序ノ巻～

電子情報工学科 3 回生 若松 健

5.1 はじめに

「音楽を自動で楽譜にできらいいな」というのがこの記事を書くきっかけです。耳コピってなんぞやっていう人のために、耳コピとは音楽を人の耳で聞いて楽譜にする、つまり耳で音楽をコピーしよってことです。音楽を聞いて、ボーカルの人が歌ってるメロディーラインをカラオケとかで歌うことは、（正しいかどうかは別として）大抵の人はできると思います。これは、人間の素晴らしい感覚によって今歌っているのがどの音程か分からなくても歌えてしまうからです。しかし、声に出して歌うのではなく楽器で演奏したい場合はどうでしょう。音程がわからないと演奏できませんよね。そこで楽譜が必要になってくるわけです。PC[1]でこの楽譜を音楽から作るにはどうすればいいのか書いていきたいと思います。

5.2 音楽

そもそも音楽とは何なのでしょうか？広辞苑には以下のように書いてあります。

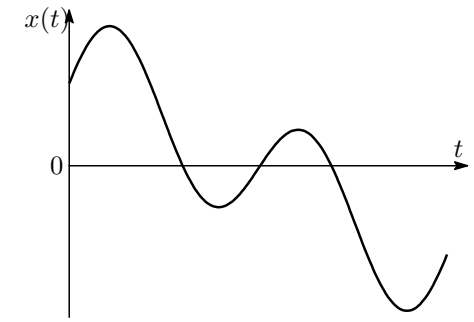
> 音による芸術。拍子・節・音色・和声などに基づき種々の形式に組み立てられた曲を奏するもの。　— 岩波書店　広辞苑第五版より一部抜粋

音楽は音の集まりであり、そこには拍子や音色などがあるわけです。音の正体は波（いわゆる音波）です。この音波は連続的に変化するのでアナログであるといいます。一方、音波を処理しようとする PC は、電気が流れているかどうか、つまりスイッチが ON か OFF かの 2 つの状態しかなく、離散的に変化するのでデジタル[2]であるといいます。このままでは、PC で音楽から楽譜を作ることはできないのでアナログからデジタルに変換してやる必要があります。これをアナログ・デジタル変換 (A/D 変換) といいます。

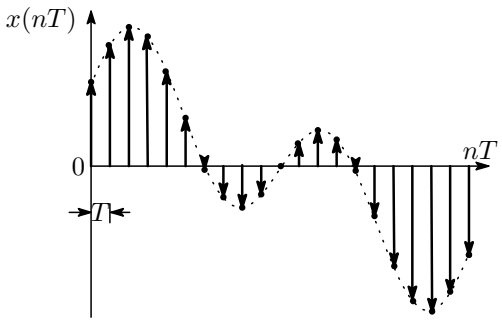
[1] Personal Computer の略。いわゆるパソコン。
[2] ディジタルと表記した方が英語の発音に近いが、この記事内ではデジタルと表記することにする。

5.3 A/D 変換

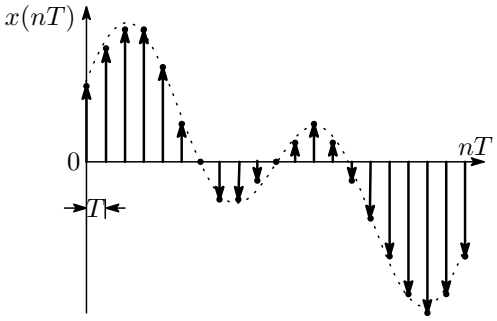
A/D 変換には、標本化、量子化、符号化の 3 つのステップがあります。それぞれのステップについて説明していきます。



(a) アナログ信号



(b) サンプル値信号



(c) デジタル信号

図 5.1: 3 種類の信号

5.3.1 標本化 (サンプリング)

標本化によって、図 5.1(a) のようなアナログ信号を図 5.1(b) のようなサンプル値信号に変換します。簡単に説明すると、一定の時間毎に信号の値を取っていくということです。アナログ信号 $x(t)$ は一定の間隔 T ごとに標本化されてサンプル値信号 $x(n) = x(nT)$ を得ることができます。一定間隔 T をサンプリング周期といい、その逆数 $f_s(= 1/T)$ をサンプリング周波数といいます。

5.3.2 量子化

標本化した信号は、時間については離散的になっていますが、信号についてはまだ連続的です。量子化によって、図 5.1(b) のようなサンプル値信号を図 5.1(c) のようなデジタル信号に変換します。量子化では、決められた量子化レベル間のサンプル値を四捨五入し、量子化レベルに対応させ

てしまいます。このために誤差が生じてしまいます。

5.3.3 符号化

量子化によって信号も時間も離散的に変換されたので、これを符号化すればデジタル信号として処理することができます。符号化は、量子化された振幅値を 2 進数のデジタルコードに変換することで、量子化レベルに対応するデジタルコードに変換します。

以上により、デジタル信号を得ることができ、PC で処理できるようになります。ここまでのことは、PC にマイクを挿して録音すれば終わりです。(録音するソフトが必要ですが、そこは既存のものを使うということで。)

5.4 フーリエ変換

音楽には、さまざまな音が混じっています。音の高さは、振動数の大小による音の性質の違いによって決まるので、どんな周波数が含まれているかが分かれば、含まれている音の高さが分かるわけです。信号のグラフでは、横軸が時間なので、時間変化による信号の変化しか見ることができず、周波数がどのように分布しているか分かりません。そこで、フーリエ変換を行います。フーリエ変換を行うとある時刻における周波数の分布を見ることができます。連続信号 $x(t)$ のフーリエ変換 (FT) を $X(\omega)$ とすると、フーリエ変換は次式で定義されます。

$$X(\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt \tag{5.1}$$

ただし、j は虚数、ω は角周波数、t は時刻である。また、離散信号 $x(n)$ の離散フーリエ変換 (DFT) を $X(k)$ とすると、離散フーリエ変換は次式で定義されます。

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}, \quad (k = 0, 1, \ldots, N-1) \tag{5.2}$$

しかし、DFT をそのまま利用してプログラムを書いたのでは、処理する量が膨大になってしまいリアルタイムでの処理は愚か、処理が完了するまでに相当の時間を必要としてしまいます。実際には高速フーリエ変換 (FFT) を利用します。FFT とは、DFT に要する膨大な計算量を大幅に減少させるアルゴリズムのことです。$x(n)$ と $X(k)$ の標本点数が N のとき、DFT では、複素乗算を N^2 回しなければならないが、FFT では、$(N/2)\log_2 N$ 回で済んでしまいます。FFT の原理を書くとかなりのページに式ばかりになってしまうので、興味のある方は参考文献 [1] 等を読んでいただきたい。

以上でどんな音が含まれているかを知ることができます。次に……えっ、時間がなくなったって!?マジで!? ということで、今後の課題を挙げて終わりにします。

- 音の長さをどうやって読み取る？

- 拍子、テンポをどうやって決める？
- 音色はどうする？
- 楽器毎の楽譜を作成するには？
- 音の強さや大きさはどうする？etc...

課題はたくさんありますが 1 つ 1 つクリアしていこうと思うので、来年の Lime をお楽しみに!

参考文献

[1] 大類重範、2004、「ディジタル信号処理」 日本理工出版会

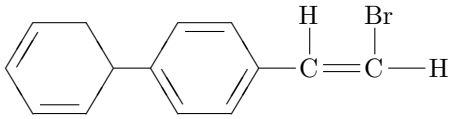
6 TEX で化学構造式を描こうっ！〜XƎMTEX・紹介編〜

物質工学科 3 回生 臼木 由美子

6.1 XƎMTEX とは

文書組版ソフトウェア、TEX。その TEX で化学構造式を取り扱いたい時に使うのが、XƎMTEX と呼ばれる一連のパッケージです。

XƎMTEX を使うことで、このような化合物も画像を貼り付けることなしに描くことができます。



今回の記事では、XƎMTEX の導入から簡単な化合物の描画までを紹介したいと思います。

なお、今回は紙面他の都合上 TEX 自体に関する説明は省かせて頂きます。既に、インターネット上に多くの方々による分かりやすい解説ウェブサイトが開設されておりますので、「TeX」または「LaTeX」等で検索してみて下さい。

この文書は、pLATEX2e + XƎMTEX4.03 の環境をもとに作成されています。

6.2 導入編（XƎMTEX の導入）

まず、XƎMTEX を使うためにはスタイルファイル（epic.sty）が必要になります。

Comprehensive TeX Archive Network（http://www.dante.de/cgi-bin/ctan-index）等から入手して、（TEX のインストールフォルダ）\share\texmf\tex へテキスト形式で保存して下さい。

続いて、XƎMTEX のパッケージを藤田眞作氏のウェブサイト（http://imt.chem.kit.ac.jp/fujita/fujitas/fujita.html）等から入手し、同様に（TEX のインストールフォルダ）\share\texmf\tex へフォルダごと解凍します。

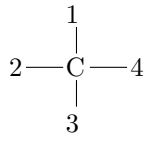
これらが終了したら、コマンドプロンプトから mktexlsr コマンドを実行すれば、導入は終わります。

XyMTEX を使いたい時は、使いたい文書の初め（\documentclass と \begin{document} の間）に
\usepackage{xymtex}
\usepackage{chemist}
と書いて下さい。後は通常通りコンパイルできます。

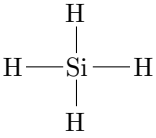
6.3 炭化水素の書き方（鎖状編）

6.3.1 炭素 1 つの化合物

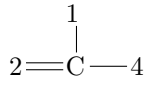
炭素 1 つの鎖状炭化水素は、\tetrahedral（四面体）というコマンドを使ってこのように書き表せます。置換基の番号は、上から反時計回りに 1、2、3、4 と割り振られています。
\tetrahedral{0==C;1==1;2==2;3==3;4==4}



もちろん、0==の部分を炭素（C) 以外にすれば、無機化合物も表せます。
\tetrahedral{0==Si;1==H;2==H;3==H;4==H}

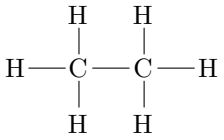


二重結合は、==の代わりに D==を使うことで表せます。また、省略した番号の置換基は表示されません。
\tetrahedral{0==C;1==1;2D==2;4==4}

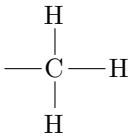


6.3.2 炭素 2 つ以上の化合物（鎖の繋げ方）

炭素 2 つ以上の化合物（エタンなど）を描く時は、このように入れ子にします。
\tetrahedral{0==C;1==H;2==H;3==H;4==1==H;2==(yl);3==H;4==H} }



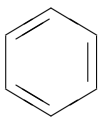
4 番の水素 (H) が、2 番から結合手の出ているメチル基
\tetrahedral{0==C;1==H;2==(yl);3==H;4==H}



に変わったと考えるといいでしょう。

6.4 炭化水素の書き方（環状編）

ベンゼン環を書き表すには、\bzdrv{}

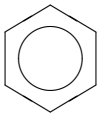


（縦置き）

または、\bzdrh{}

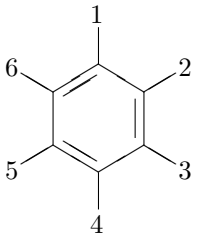
（横置き）

を使います。ベンゼン環の内側を、二重結合と単結合の代わりに、共役を表す丸で表したい時はこのようにオプション [c] をつけます。
\bzdrv[c]{}



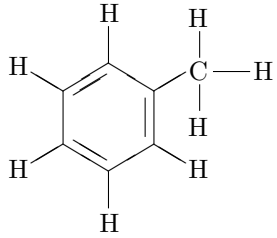
置換基を付ける時は、鎖状炭化水素の場合と同じように書き表します。但し先程と違い、置換基の番号は時計回りに割り振られています。

\bzdrv{1==1;2==2;3==3;4==4;5==5;6==6}

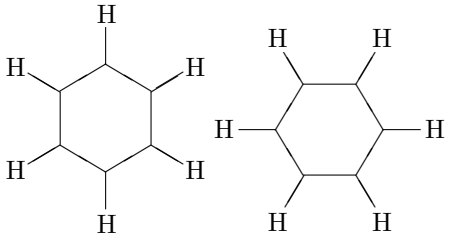


もちろん、このようなこともできます。

\bzdrv{1==H;2==\tetrahedral{0==C;1==H;2==(yl);3==H;4==H};3==H;4==H;5==H;6==H}

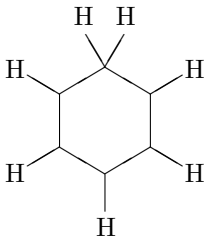


シクロヘキサンの場合は、\cyclohexanev（縦置き）、または \cyclohexaneh（横置き）で表します。\cyclohexanev{1==H;2==H;3==H;4==H;5==H;6==H}

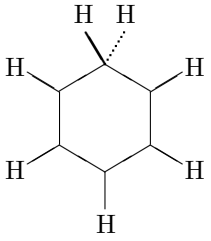


1 つの部位に 2 つの置換基が付く時は、Sa==と Sb==で表します。

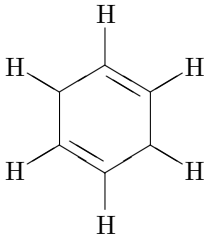
\cyclohexanev{1Sa==H;1Sb==H;2==H;3==H;4==H;5==H;6==H}



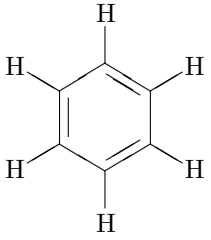
ここで大文字の SA==、SB==とするとそれぞれが点線と太線になり、立体結合を表せます。
\cyclohexanev{1SA==H;1SB==H;2==H;3==H;4==H;5==H;6==H}



二重結合のあるもの（シクロヘキセン）を描くには、オプションで二重結合になる結合を指定します。指定の仕方は、1 と 2 の間が a、2 と 3 の間が b…というようになっています。
\cyclohexanev[ad]{1==H;2==H;3==H;4==H;5==H;6==H}



このように、ベンゼン環を表すこともできます。
\cyclohexanev[ace]{1==H;2==H;3==H;4==H;5==H;6==H}



因みに、冒頭の化合物はこのように表されています。
\bzdrh{1==
\cyclohexaneh[ae]{4==(yl);};4==
\tetrahedral{0==C;1==H;2==(yl);4D==
\tetrahedral{0==C;1==Br;2==(yl);4==H}};}

6.5 最後に

XeLaTeX の新しい Version にはさらに多くの機能があり、多種多様な化合物を書き表せるようになっています。この記事での紹介はここまでとしておきますが、興味を持たれた方は XeLaTeX のマニュアル（英語です）や、入門書、参考書籍などを参照してみるとよいかと思います。

7 GEOM

電子情報工学科 5 回生 池野 直樹

7.1 GEOM についてのあれこれ

まとまるのかまとまらないのか知らないし、締め切りも過ぎてしまったんだけど、今更ながら FreeBSD の GEOM についてまとめてみる。どういう結論に行き着くのかもわからずに書きはじめて、どういう結論に落ち着くのか、そもそも結論なんて出るのか!? 乞うご期待。目標書き上げ時間まであと 10 時間、なんだかんだで実作業時間は 5 時間、2000 字くらい書ければいいよね？

7.1.1 そもそも GEOM ってなんなのさ？

ってことでぐぐってみると、

> 差込可能なストレージ層 GEOM により、新たなストレージ サービスをすみやかに開発して、FreeBSD のストレージサブシステムに きれいに組み込むことができます。GEOM は、ストレージサービスを 検出したり階層化して用いるための一貫して筋の通ったモデルを 提供しており、RAID とボリューム管理のようなサービスを 重ね合わせて使うことが容易になります。(http://www.freebsd.org/ja/features.html)

って話が出てくる。 わかったようなわからんような説明だわな。わかるところから意味をとってみると、GEOM により RAID とかディスクの暗号化といったストレージサービスをストレージサブシステムに組み込んで、それを通してストレージを使うための一貫したシステムということになる様子。やっぱりわかりにくいかも。 ともあれこの辺は何となくでいい様な気もする。

ともあれ普通であれば/dev/ad0s1 などにアクセスしていたのを、geom を使用することにより、その/dev/ad0s1 を暗号化してアクセスしたり、ad0s1 と ad1s1 をまとめて 1 つの論理ディスクとして使用したりすることができるようになる。ちなみに FreeBSD-5.0-RELASE 以降は普通の物理ディスクへのアクセスであっても geom を通してアクセスしているんだけど。

もっと詳しく知りたければ、man 4 geom で詳しい話は出てくるけど、あんまり役に立たない様な気がする。

7.1.2 geom でいろいろ

さて、geom がなんなのか何となくわかったような気分になったところで、geom で何ができるのかをみていこうと思う。

ここでもやっぱり man に聞くのが一番ということで、man 8 geom ってしてみると、下の方に現在利用可能な geom class ということで、いくつかの項目が載っている。

- CONCAT
- LABEL
- MIRROR
- NOP
- RAID3
- SHSEC
- STRIPE

これらを順番に見ていこう。 どれをみるのかはご丁寧にその下にコマンドについて書いてあるので、man から調査してみる。gconcat(8), glabel(8), gmirror(8), gnop(8), graid3(8), gshsec(8), gstripe(8)

CONCAT

ようするに concatenation、複数のディスクをまとめて一つの領域として使う方法で、いわゆる RAID0。やり方は man にも書いてあるけど簡単で、

```
gconcat label -v data /dev/da0 /dev/da1 /dev/da2 /dev/da3
newfs /dev/concat/data
mount /dev/concat/data /mnt
[...]
umount /mnt
gconcat stop data
gconcat unload
```

こんなかんじ。ちなみに da1、da2、da3 は容量が違っても OK。 そのかわりどれかの領域が死んだら、データのサルベージができなくなる可能性があるのでご注意。

LABEL

glabel コマンドを使って、/dev/ad0 なんかに usr ってラベルつけて、マウントできるようにするための仕組。使い方は man にもあるけどこんな感じで。

```
glabel label -v usr /dev/da2
newfs /dev/label/usr
mount /dev/label/usr /usr
[...]
umount /usr
glabel stop usr
glabel unload
```

ラベリングするだけでこれだけだとあんまり利用価値がないのかな。ディスクとかが多くて、ad3がどこにマウントされるべきなのかわからない、と言うようなときには便利なのかも知れない。

MIRROR

激しくおすすめの gmirror。 2 つ以上のディスクに同じデータを書き込んで、どちらかのディスクが壊れても、問題なくシステムが動き続けられるようにするもので、いわゆる RAID1。構築方法はやっぱり簡単でこんな感じ。

```
gmirror label -v -b split -s 2048 data da0 da1 da2
newfs /dev/mirror/data
mount /dev/mirror/data /mnt
...
umount /mnt
gmirror stop data
gmirror unload
```

gconcat もそうだけど gmirror も root ファイルシステムを gmirror にしたり、gconcat にしたりすることができる。この場合、OS 起動中はディスクが使用されているので、FreeBSD のインストールディスクを使用して、fixit で起動して gmirror 等の設定をおこなう必要がある。 やりかたはネットに幾らでもあるので、ぐぐる先生にでもお尋ねしてください。 ちなみに gconcat でいくつかの領域をまとめて、まとめた領域を使用して gmirror という事もできるらしい。 やったことないけど。geom はこうやって複数の処理を重ねることができるというのも特徴の一つで、重ねすぎてループが発生した場合には geom がその重複をきちんと検出してくれる仕組もある。

NOP

gnop はどうやら I/O エラーなんかを任意に作り出して、ほかの geom サブシステムのデバッグや性能調査に使用するものの様子。/usr/src/sys/conf/NOTES でも GEOM_ NOP は test class とされてるからそう言うもんでしょう。

RAID3

名は体を表す。とかそんな大げさなもんじゃないけど、GEOM による RAID3。ちなみに GEOM による RAID5 はまだ実装されていないが、開発はされてると言うようなことをどっかで読んだ。GEOM ベースの RAID5 が早いこと実装されて欲しいもんだ。

やり方は他と同じようなもので

```
graid3 label -v -r data da0 da1 da2
newfs /dev/raid3/data
mount /dev/raid3/data /mnt
...
umount /mnt
graid3 stop data
graid3 unload
```

RAID3 なので、ディスク領域は同じ大きさのが 2n+1 必要で、ちがう大きさなら小さいのにそろえられてしまう。ついでにどうやら一度つくった領域に新しくディスクを追加して領域を動的に増やすという事はまだ出来ない様子。今のところざっとネットを調べただけだけど、GEOM_RAID3 をつかってシステムの root を RAID3 にすることが出来るかどうかは謎。

SHSEC

いくつかのディスク領域をまとめて暗号化された一つの領域をつくるシステム。gshsec(8) に乗っている使い方として、ローカルドライブと USB メモリを利用して一つの gshsec 領域を作成すると、その USB メモリがない限りその領域に保存されたデータを読むことができなくなる。 この USB メモリは複数あってもいいようなので、重要なデータを複数の人の立ち会いの元でのみ閲覧できるというようなシステムに使える。 暗号化の方法が良くわからないので、どの程度安全かは不明だけど。gshsec で使用するディスクの容量は同じ大きさにする必要があるらしく、容量が違うもので gshsec すると小さいものにそろえられてしまうようだ。

領域の作り方は例によって

```
gshsec label -v secret /dev/ad0s1 /dev/da0
newfs /dev/shsec/secret
```

こんなかんじ。

STRIPE

CONCAT はディスク領域の大きさが違ってもまとめて一つのドライブとして使えたが、STRIPE は同じ大きさの領域をまとめて使うことになる。イメージ的には CONCAT がただ単に領域をくっつけて連続的にしているのに対して、STRIPE は領域を細かく分けて、細かく分けた領域を均等になるように配置している感じ。 計測したわけじゃないけどアクセス速度は STRIPE の方が早い。はず。 そのかわり STRIPE では一つのドライブが壊れるとデータの復旧は絶望的なのに対して、CONCAT の場合はサルベージ出来る可能性はある。 難しいと思うけど。先にも書いたけど GEOM はサブシステムを複数重ねることもできるので、STRIPE をするときには MIRROR をした方がいいでしょうね。

作り方はこんな感じ。

```
gstripe label -v -s 131072 data /dev/da0 /dev/da1 /dev/da2 /dev/da3
newfs /dev/stripe/data
```

```
mount /dev/stripe/data /mnt
[...]
umount /mnt
gstripe stop data
gstripe unload
```

7.1.3 まとめみたいな〜

つくった GEOM の領域などはそれぞれのコマンドで clear オプション等を使用して設定をクリアすることができる。また GEOM のメタデータは領域の最後に配置されるようだ。一杯一杯まで使ってるドライブだと GEOM にするとき何かエラーが出るかも知れない。

さてここまで急ぎ足で、おもに man に書かれていることを中心に GEOM に関して書いてきた。ここでは geom(4) に書かれている GEOM 関連のコマンドから概説してきたが、/sys/conf/NOTES にある GEOM 関連の options を見てみると、ずらーっとこれだけある。

```
options  GEOM_AES # Don't use, use GEOM_BDE
options  GEOM_APPLE # Apple partitioning
options  GEOM_BDE # Disk encryption.
options  GEOM_BSD # BSD disklabels
options  GEOM_CONCAT # Disk concatenation.
options  GEOM_FOX # Redundant path mitigation
options  GEOM_GATE # Userland services.
options  GEOM_GPT # GPT partitioning
options  GEOM_LABEL # Providers labelization.
options  GEOM_MBR # DOS/MBR partitioning
options  GEOM_MIRROR # Disk mirroring.
options  GEOM_NOP # Test class.
options  GEOM_PC98 # NEC PC9800 partitioning
options  GEOM_RAID3 # RAID3 functionality.
options  GEOM_SHSEC # Shared secret.
options  GEOM_STRIPE # Disk striping.
options  GEOM_SUNLABEL # Sun/Solaris partitioning
options  GEOM_UZIP # Read-only compressed disks
options  GEOM_VOL # Volume names from UFS superblock
```

みてみるといろいろある。ちなみに GEOM_ GETE というのをつかうとデバイスファイルをネットワーク経由でマウントできるようになるらしい。要するにサーバにある /dev/acd0 を、ネットワークを通じてクライアントの /dev/acd0 としてマウントできるとか。これはこれで面白そうな機能だ。

こんな風に GEOM を使うと、今までただ端にアクセスするだけだったディスク領域に様々な機能を付け加えることができ、利用のシーンが大きく広がるのではないかと思います。 サーバ運用している人には gmirror はおすすめだわ。興味あるところでは graid3 でのルートファイルシステムの運用ができるかどうかを実験して (gmirror で出来るんだから出来るのではないかと予想してるんだけど…)、graid3 の評価をしてみたいと思う。今後に乞うご期待。

編集後記

Lime の編集も 2 年目となりました。去年は、原稿の締め切りが早すぎたみたいで、記事数が少なく、また “裏らいむ” なるものも出現しました。“裏らいむ” とは、Lime の締め切りが早すぎて間に合わなかった人たちなどが作成されたものです。なぜ去年は締め切りが早かったかというと、印刷所に原稿を出して製本してもらっていたからです。印刷所の製本はきちんとしていていいのですが、やはり Lime の方の記事を増やさなければということで、今年は印刷所には出さずに製本しようということになり今に至ります。ギリギリまで待てばみんな出してくれる雰囲気なので、こういうのでもいいのかなと思ってたりします。

来年もたぶん自分が編集担当になるだろう（そろそろ後釜を見つけておかねば）。来年の Lime は今年よりも記事数が増えるよう努力したいと思うので来年の Lime もご期待ください。

平成 17 年 11 月 16 日 編集担当 若松 健

Lime Vol.32

平成 17 年 11 月 18 日 発行 第 1 刷

発行 京都工芸繊維大学コンピュータ部
http://www.kitcc.org/