G1. 情報: 第1部

第1日　環境設定, Cプログラミングの復習

1. はじめに

G1〜G3 の課題は各 4 回, 合計 12 回分で一続きの内容となっている.

- UNIX 一般, Linux (演習で用いる環境) の基礎
- プログラミング一般 (デバッグ方法等), C プログラミングの基礎

から始まる. 中核部分では,

- ファイル入出力, 音の入出力,
- 音を対象としたデータ処理, デジタル信号処理
- インターネット, ネットワークプログラミング

を学ぶ. 最後には面白い応用プログラムとしてインターネット電話 (1 対 1 の通話), あるいは電話会議システム (多対多の通話) を完成させる事を目指す.

いきなりインターネット電話を作るといわれると, 一見難しそうに思えるかもしれない. しかし, 要は音を録音する, 再生する, 音データをネットワークで転送する, の 3 つができれば良いのである. 前者については, 演習用の Linux 環境では特定のファイル (/dev/dsp) を普通のファイルのように読み書きするだけで, 思いのほか簡単にできる事を知るだろう. だから, ファイル入出力のプログラミングができていれば即, 音の録音や再生ができる. 後者は, データが音声であろうとなんであろうと, ネットワークを介してデータを送るという, ネットワークプログラミングの基本を学べばできる. したがってファイル入出力, ネットワークという, 実用プログラミングの基本を身につけてしまえば, もう一頑張りできっとできる事で, 実験期間の最後の方にそれが「動いた!」という喜びをぜひ多くの人々に体験して欲しいのである.

本テキストはそれなりに自己完結的に書かれているので, 説明を順に読んで内容を理解してほしい. その途中, 実際に作業や考察する事を要求する項目には, 以下の印のいずれかがつけられており, その位置づけは以下のとおりである.

準備課題 : 簡単な設定作業の類 (例えばネットワークへ接続するための設定) や, 後で課題を実行するための準備となる簡単なプログラミング. 必須項目と考えるべきだが, もし, チームメンバ全員がすでにそれらの項目に慣れきっており, 「目をつぶってでもできる」というような場合はとばしても構わない.

本課題 : 各回に数問設定されている, いわば「その日の目標」とでもいうべき課題. これらについては全班がプログラムを完成させること. 進度把握のため, プログラムは電子的に提出する. ただし, これらひとつひとつにレポートを書いて提出する必要はない.

選択課題 : 余力のある人には是非やってもらいたい課題. 内容的には重要だが, 仮にできなくてもその先でつまづく事はないだろうという意味で, 選択課題としている.

基本的には, 準備課題または本課題と印がつけられた項目を順にこなして行くのがよい.

実験期間の最後の 2 回を用いてできたプログラムの発表会を行う. 後日レポートを提出する.

目標の置き方:　実験期間を通しての最終課題は, 前述のとおりインターネット電話 (1 対 1 の通話), あるいは電話会議システム (多対多の通話) を C のプログラムとして完成させる事である. これを動かす, できればそこで何らかのオリジナリティを発揮して, 発展的課題に挑戦する事が, 学期を通しての全体目標である. 各回では, そのために必要な事を学ぶという目的意識を持って臨むとよい.

各回の実験時間の使い方:　最終課題達成に向け, 実験の各回ではその回用に設定された「本課題」を, 時間内に完成させる事を目指す. そのために本テキストを読んで予習, 心の準備, 必要ならば過去に習った項目の復習をしてくる必要があるのは言うまでもない. 実験の時間内に, 教員が必要な項目を講義形式で説明する時間帯もある. その時間帯は講義時間と同様, 話を聞くことに集中する. それ以外の時間帯は実験に取り組む. 普通はまず最初に, その日の目標である本課

題を頭にいれ, その後, その日の準備課題を順にクリアしていき, 最終的にその回の本課題をクリアする. 準備課題は, 本課題をクリアするのに理解すべき必須項目を少しずつ導入しているもので, これを順にクリアしていけば本課題に取り組み易くなるはずである. しかし, 班員全員がそれを読んで, 「簡単すぎてしかたがない」と思えるような場合は飛ばして, いきなり本課題に取り組んでも構わない.

注意: 決して「面倒臭いから」「早く済ませたいから」という理由で準備課題を飛ばさないこと. あくまで, すでにそのような課題は日常的にやっており, それらの項目は目をつぶってでもできる, と言う人まで形式的にそれらをこなす必要はない, という意味で飛ばしてもよいと言っている. 実際には本課題をやるには, 準備課題相当の事をやらなくてはいけないようになっている場合がほとんどであり, 飛ばしたところで作業量が削減されるわけではない. したがっていきなり本課題に取り組むのは, それをいきなり見せられただけで, 何をやったらいいかがすぐに分かってしまう人向けの選択肢である. そうでない人は準備課題を順々にこなしていくのが, 「急がば回れ」で結果的に早く本課題をこなせるのではないかと思う.

チームワークについて: 課題は, 普段 4 人一組の班を 2 つに割って行う. つまり, 1 チーム 2 人のチームを作って行う (奇数人の班は 2 人または 3 人のチーム). そして 1 チームでひとつのプログラムを完成させればよい. そのために, チーム内でコミュニケーションを取りながら協力して取り組むこと. 決して, 各自がてんでにテキストを読み, 気が向いたところで連携も取らずにコンピュータに向かい, 話もせずに黙々と取り組む, などという状態に陥る事のないようにすること.

基本スタイルとして, どうすればいいかを二人が理解したら, どちらか一人がプログラミングをする, もう一方の人はそれを背後から見てツッコミを入れる, あるいは質問をする, というスタイル (ペアプログラミング) が良い. それを, 時々プログラミングをする係を交代しながら行う. こうして, お互いがコミュニケーションを取りながら, 正しいプログラム, 良いプログラムに関する議論をしながら作業を進めていくこと. お互いがやる事をよく理解し切った後でなら, ある程度の「分担」をしてもよいが, 「負担を半分にする」目論見で最初から課題を半々に分けるとか, ましてや「今日は A 君, じゃ, 後はよろしく」などという分担の仕方をしないようにする.

特に, チーム内でプログラミングに対する慣れの差が大きいときは, 多少時間がかかるのは承知で, 特に序盤は不慣れな人が積極的に作業を行い, もう一方がそれにツッコミを入れるというスタイルを取ってほしい. また, 初級者は自分の慣れが少ないという事で遠慮したりせず, 怪しい部分を見つけたら, このプログラムはここがおかしいのではないか, このプログラムはなぜ正しいのか, などの議論を仕掛ける事を心がけて欲しい (案外相手もよく分かっていなかったりするものである).

時間が余ったら: 各回の進度の目安を与えるために本課題が設定されているが, G1-G3 の内容は一つながりになっており, あくまで全体として一つのプログラムを完成させる事が目標になっている. したがってある回に時間が余ったら先へ進めばよい. 第 8 回に必須課題までの内容を説明し終える予定であり, その必須課題をこなした後どのような発展的課題に取り組むかは, 各チームの裁量を重視する. ある回に時間が余ったら, 是非その余った時間を利用して, 先の回の予習をする, できるのであれば作業をする, 発展的課題について構想を練る, 議論をする, などをしてほしい. 必須課題が早くできてしまえばその分発展的課題に多くの時間を割く事が出来る.

2. プログラミング以前の準備

2.1 ログイン

演習環境は, 学科が貸し出している PC の Linux (Ubuntu) を用いる. 貸し出している PC は Windows も立ち上がるようになっているが, **本演習の内容は, 音データの入出力, ネットワーク関係のコマンドなど, 多くの場所で Linux を前提としている.** 慣れていない人もこの機会に学ぶこと.

準備課題 1.1 Linux を立ち上げ, 自分のユーザ (特に作っていなければ, denjo ユーザ) としてログインせよ.

2.2 ネットワークへの接続

準備課題 1.2 演習用のワイヤレスネットワーク ZENKIJIKKEN へ接続せよ. 確認がてら, 本課題の HP がある, 以

下の URL へアクセスせよ. 以降ここからお知らせを流したり, 資料やプログラムを提供する事もある.

http://g1g2g3.logos.ic.i.u-tokyo.ac.jp/

2.3 シェル

いろいろなコマンドを実行するのにもっとも基本的な道具は, いわゆる**コマンドライン**で, 通常, **シェル**と呼ぶ. シェルを起動するには, ランチャから「端末」アプリケーションを選ぶ.

準備課題 1.3 画面左のランチャから「端末」を選択して, 端末ソフトを開け. もしランチャに見当たらない場合, 画面左上隅の Dash ホームボタン + terminal で見つかるはずである.

2.4 Emacs エディタ

ファイル (プログラムやデータ) を編集するのにはエディタを用いる. エディタにはいろいろな種類があるが Emacs を使いこなせるようになるとよい.

準備課題 1.4 画面左のランチャから「Emacs」を選択して, Emacs エディタを開け. もしランチャに見当たらない場合, 画面左上隅の Dash ホームボタン + emacs で見つかるはずである.

3. C プログラミングの環境設定

Linux で C プログラミングを行うための基本フォームを説明する. 4 学期に駒場で使った Mac と共通部分が多いので, よく復習するとよい.

3.1 C コンパイラ

Linux では, 標準的な C コンパイラとして cc または gcc というコマンド (実はどちらも同じ物) を用いる. もっとも単純には, C プログラムがかかれたファイル (例: ini.c) を用意し,

```
$ gcc ini.c
```

とする (実際に入力する部分は **gcc ini.c** のように太字体で示されている. $はシェルのプロンプトに向かって入力する事を指示する印で, これ自身を入力するわけではない). これで a.out という実行可能ファイルができるが, その名前を指定したければ, -o オプションを用いて,

```
$ gcc -o ini ini.c
```

とする. できた実行可能ファイルは,

```
$ ./ini
```

のように実行する. "./" を忘れて,

```
$ ini
```

だけだと,

```
$ ini
bash: ini: command not found
```

というエラーになるので注意.

準備課題 1.5 以下のプログラムを Emacs で ini.c という名前で作り保存せよ. それをコンパイルし, ini という実行可能ファイルを作れ.

```
main(int argc, char ** argv) {
  char * gn = argv[1];
  char * fn = argv[2];
  char g = gn[0];
  char f = fn[0];
  printf("initial of '%s %s' is %c%c.\n", gn, fn, g, f);
}
```

ここでたくさんの警告が出るが一旦気にせずに先へ進む.

コンパイルできたら, 以下のようにコマンドを実行して確認せよ.

```
$ ./ini Daisuke Matsuzaka
initial of 'Daisuke Matsuzaka' is DM.
```

引数が足りないと以下のような出力 (Segmentation Fault) になる.

```
$ ./ini Daisuke
Segmentation fault
```

これも一旦気にせずに先へ進む.

3.2 ファイル作成と周辺の基礎

念のため, 上記を実行するための Emacs の操作と基礎概念の復習 (4 学期に Mac でやったものとほとんど同じ).

(1) **Emacs でファイルを作る:**

C-x C-f (Ctrl キーと x を同時に打ち, そのまま Ctrl キーを押しながら f を打つ)

とすると

Find file: ~/

というのが下に現れるので, ~/ 以降にファイル名 (ここでは ini.c) を入力して Enter.

Find file: ~/**ini.c**

~は自分の, 「ホームディレクトリ」を表しており, 上記の操作でホームディレクトリ直下にファイルが出来る.

(2) **Emacs でファイルを保存:**

C-x C-s

(3) **ファイル操作の GUI:** Mac のファインダや Windows のエクスプローラに相当するものは, 画面左のランチャから開く. ini.c を作ったらそれが見えるはず. または端末上での操作から GUI に移行したい場合, 端末で,

```
$ nautilus .
```

とすると, カレントディレクトリを GUI 表示してくれる.

(4) **シェルの基本コマンド:** シェルを起動した直後,

```
$ ls
```

とするとホームディレクトリのファイル一覧が表示される．ini.c を含め, 上記の GUI と同じファイルが見えるはずである．

ディレクトリを理解する　今後, すべてのファイルをホームディレクトリの下に作っていくというわけにはいかないので, 適宜ディレクトリを作る, 作ったディレクトリ内に Emacs でファイルを作る, 作ったディレクトリ内のファイルをコンパイル, 実行する, という事が出来るようになっておいてほしい．そのために必要な, 復習しておくべきコマンドと概念を列挙しておく．

(1) **パス名:** ファイルやディレクトリの名前の事．

- /から始まるパス名は, **絶対パス名**と呼ばれる．例えば/home/tau/ini.c は,
  - ルートディレクトリ (システムに唯一存在する, 頂点となるディレクトリ) の中の,
  - home というディレクトリの中の,
  - tau というディレクトリの中の,
  - ini.c というファイル (ひょっとしたらディレクトリかもしれない．名前だけからは判断できない)

  を意味している．たとえ話としては, 2 号館実験室を, 「大宇宙銀河系太陽系地球日本国東京都文京区本郷 7-3-1 工学部 2 号館 4F 新館実験室」というようなもの．
- /から始まらないパス名は, **相対パス名**と呼ばれる．ini.c のように単独でファイル名を指定しているのも, 相対パス名の一種である．相対パス名は以下で説明する「カレントディレクトリ」を起点とした位置を指定している．例えばカレントディレクトリが/home/tau であれば, ini.c は, /home/tau/ini.c を意味し, enshu/mk_data.c は, /home/tau/enshu/mk_data.c を意味している．いちいち「大宇宙銀河系太陽系地球日本国東京都文京区本郷 7-3-1 工学部 2 号館 4F 新館実験室」という代わりに, 日本国内では「東京都文京区本郷 7-3-1 工学部 2 号館 4F 新館実験室」と言えばよいし, 普段工学部 2 号館にいる人は「新館実験室」でよいのと同じ．

(2) **カレントディレクトリ:** 起動されているプロセスごと保持されている．シェルの場合, pwd で表示できる．

```
$ pwd
/home/tau
```

シェルを立ち上げると, 初期状態ではカレントディレクトリは自分のホームディレクトリとなっており, それは実験環境では, /home/⟨ ユーザ名 ⟩ となっている．

(3) **ディレクトリを作る:** ディレクトリを作るには,

```
$ mkdir ⟨ ディレクトリ名 ⟩
```

例:

```
$ mkdir g1g2g3
```

は (g1g2g3 が相対パスなので), ⟨ カレントディレクトリ ⟩/g1g2g3 というディレクトリを作る．

(4) **Emacs でファイル作成:** Emacs をはじめとして色々なソフトでホームディレクトリを~/ と表記する習慣があり, ファイルを作成した際の,

```
Find file: ~/ini.c
```

は, ホームディレクトリ下に ini.c, つまり (ホームディレクトリが/home/tau であれば), /home/tau/ini.c を作るという事になる．もちろんここで,

```
Find file: ~/enshu/mk_data.c
```

とやれば, /home/tau/enshu/mk_data.c ができる．

(5) **カレントディレクトリの変更:** シェルでカレントディレクトリを変更するには,

```
$ cd ⟨ ディレクトリ名 ⟩
```

とする. また,

```
$ cd
```

は,

```
$ cd ⟨ ホームディレクトリ ⟩
```

の略記.

例:

```
$ cd g1g2g3
```

なお, カレントディレクトリを D にする事をしばしば, 「D に移動する」とか, 「D に行く」などと言う.

(6) **ディレクトリの内容を表示:**

```
$ ls ⟨ ディレクトリ名 ⟩
```

で ⟨ ディレクトリ名 ⟩ にあるファイルやディレクトリ一覧が表示される.

```
$ ls
```

は,

```
$ ls ⟨ カレントディレクトリ ⟩
```

の略記.

(7) **コピーや移動:** cp や mv などでファイルをディレクトリ間でコピーしたり移動したりして, 作ったファイルの「整理」ができるように. 慣れないうちは GUI でやってもよい.

ディレクトリ構造と, 「カレントディレクトリ」の概念を頭に入れて, ファイルがどこへ行ったのかきちんと把握出来るようにしておくこと. 以下のような手順がよく現れる.

```
$ cd              # ホームディレクトリへ移動
$ mkdir enshu     # enshu ディレクトリを作る
ここで Emacs で ~/enshu/xyz.c を作成
$ cd enshu        # enshu ディレクトリへ移動
$ ls
xyz.c             # xyz.c ができている事を確認
$ gcc xyz.c       # コンパイル
$ ./a.out         # 実行
```

3.3 Emacs 内でのコンパイル

プログラムの編集 → コンパイル → エラーを修正するためにまた編集 → またコンパイル → ··· というサイクルを効率的に行うために, Emacs の中で gcc を実行する方法をぜひ身につけてほしい.

それには端末で gcc を実行する代わりに, Emacs で

```
M-x compile
```

(発音: 「メタエックス コンパイル」) というコマンドを実行する (このコマンドの実際の入力方法が分からなければ下記参照). すると, Emacs 下部の細いところ (**ミニバッファ**という) に, コマンドを入力するための以下のようなプロンプトが現れる (**図 G1.1.1**).

なお, プロンプトが現れている状態でやっぱりやめたくなったら, **C-g** で脱出する (コマンド実行前の状態に戻る). Emacs では一連のコマンド操作をやっている最中に訳が分からなくなったら, 大概の場合 **C-g** で脱出できる.

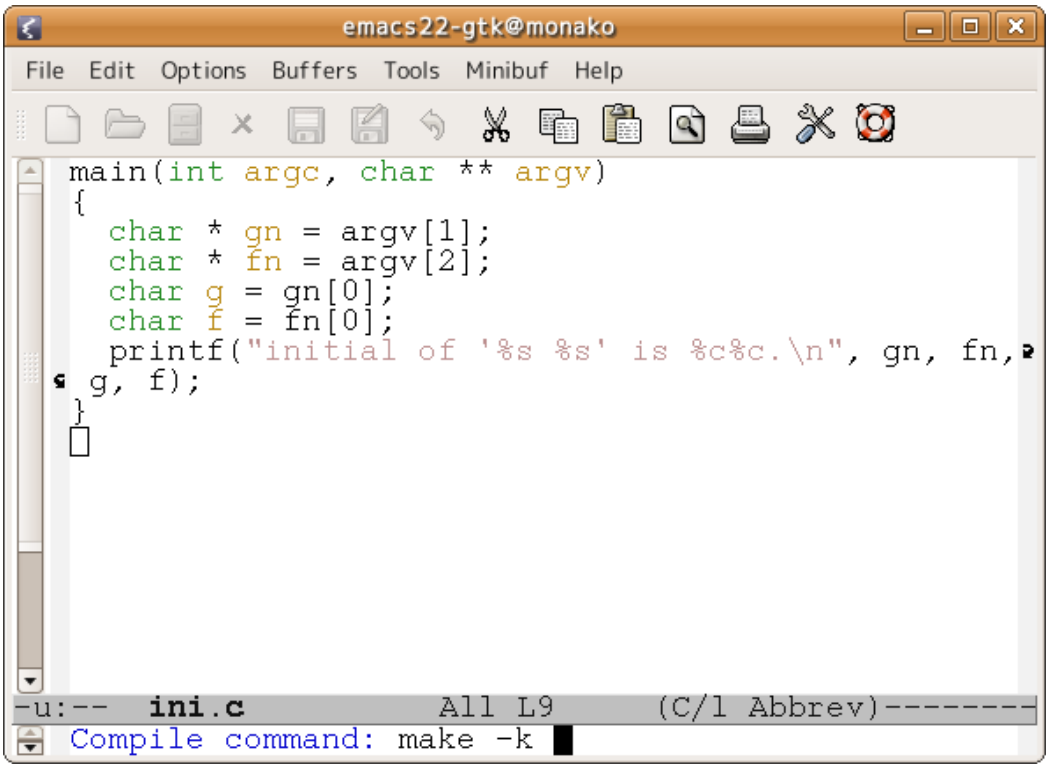

図 G1.1.1 M-x compile 入力後の Emacs 画面

Compile Command: make -k

"make -k" の部分を消して実行したいコマンド (端末に入力する物と同じ) を入力する.

Compile Command: **gcc ini.c**

Emacs の中で別のバッファが開き, 端末で実行した時と同じような出力が現れる (**図 G1.1.2**).

同じ出力を得るのにいちいち M-x compile とやるのは, 一見すると端末で入力するよりもかえって面倒なようだが, ある程度プログラムの行数が増えて来て, コンパイルエラーや警告が出た際に真価が発揮される. コンパイルエラーや警告が起きたら,

C-x ‘ (まず Ctrl キーと x を同時に打ち, その後‘ (= Shift + @) を単独で打つ)

を打つ事でエラーや警告が起きた場所に自動的にカーソルが飛んでくれる. **C-x ‘**を繰り返し入力すると次々へ別のエラーや警告の場所へ飛んでくれる. ‘はバッククオート (逆引用記号). 普通の引用記号 (’) ではないので注意. 普段あまり使わない上に, キーボード上見つけにくい位置にある.

M-x コマンドについて Emacs の操作方法の説明の中で, "M-x ⟨ コマンド名 ⟩" という表記が使われる. M-x とは具体的には, **M-**(**メタキー**と呼ばれる) と, **x** を同時に押す事を意味する. ではそのメタキーとはどのキーの事か (‘**M**’ の事

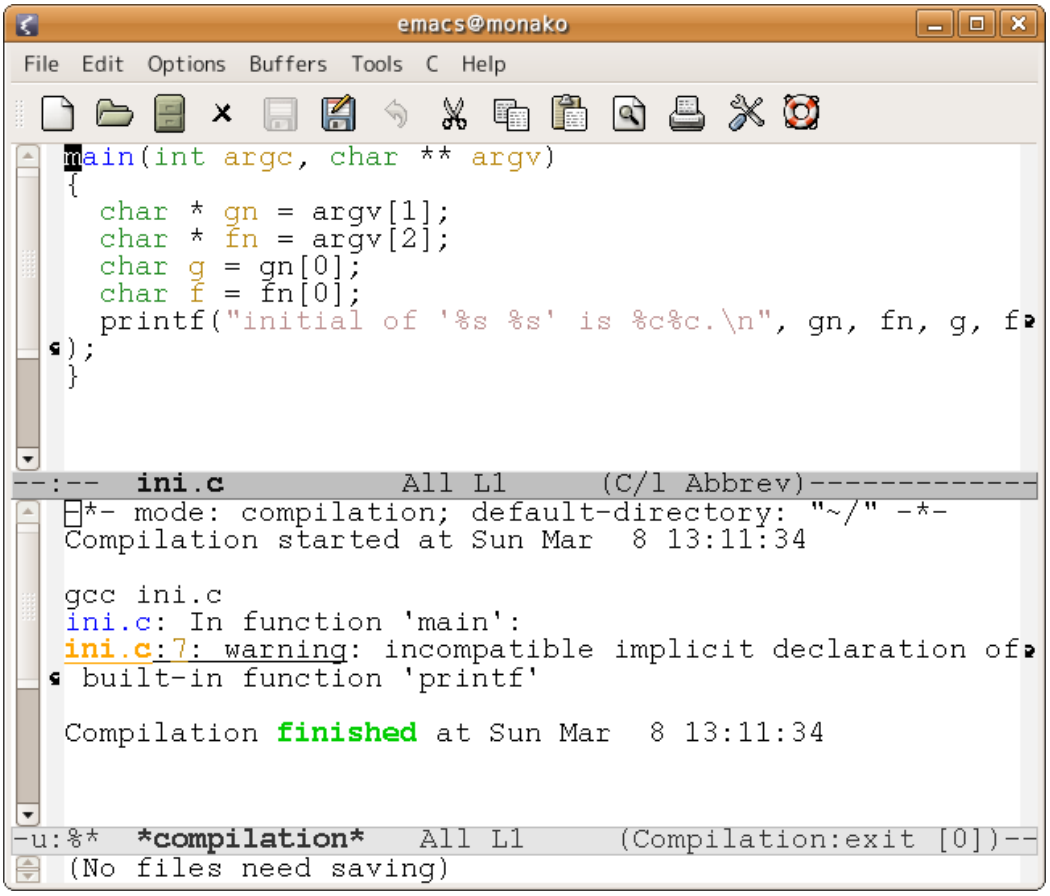

図 G1.1.2 M-x compile で gcc を実行後の Emacs 画面

ではないので注意)? ややこしい事に, キーボードのタイプによって違う事がある上, 何通りも入力の仕方がある. 好みに応じて使い分けて欲しい.

- **Alt** キー (通常スペースキーの左側). x との距離が近いので速いが, 使えない事もある.
- **ESC** キー (通常左上). ほぼ確実に使えるが x との距離が遠い. この場合実は ESC キーと x を同時に打つ必要はない (ESC キーを押した後, x を押せば良い).
- **C-**[キー (Ctrl キーと [キーを同時に押す). どこでも使えて, しかも Ctrl は左手, [は右手なので, 慣れると速い. この場合も **C-**[キーと **x** を同時に打つ必要はない.

準備課題 1.6 上の ini.c を, M-x compile を用いてコンパイルせよ. **C-x** ‘で警告やエラーの場所に自動的にカーソルを飛ばしてみよ. M-x compile でコンパイルする際, -Wall というオプションをつけてみよ (つまり, gcc -Wall -o ini ini.c). 多数の警告 (タイプミスをしていればエラーが出るかもしれない) が出力される. それぞれの警告やエラーの場所にカーソルを飛ばし, その意味を考え, 警告が出ないように修正してみよ.

なお, M-x compile を実行すると, 画面がひとりでに二つに分割される. 元の状態に戻したければ, **C-x 1** を入力すると元に戻る. Emacs で, 画面の分割や画面に表示する内容・ファイルを切り替える方法については, **4.** 節にまとめた.

機会のある時に一度, Emacs のマニュアルを読んで, 画面分割, それを元に戻す, 画面間を行き来する, 画面に表示するファイルを変える, などの操作が出きるようになっておくと良い (多くはメニューからでも行えるので, なれないうちはそれを使うのも良い).

4. Emacs におけるウィンドウ, バッファの扱い

Emacs では, 同時に複数のファイルを開いてそれらを行ったり来たりしながら編集する事が出来る. 画面を分割して半分にファイル A, もう半分にファイル B, というような事が普通にできる. 純粋に複数のファイルを編集するだけなら, Emacs を複数起動してもよいのかもしれないが, この機能は先の M-x compile や, 以下で紹介する M-x gdb (Emacs 内でデバッガを実行する機能) を使ったときに, ひとりでに発動される (勝手に画面が分割される). したがって画面を分割した状態から元にもどしたり, 分割して出来た小画面の間を行き来したり, それぞれの小画面に表示するものを切り替えたり, などが自在にできないと戸惑う事になる.

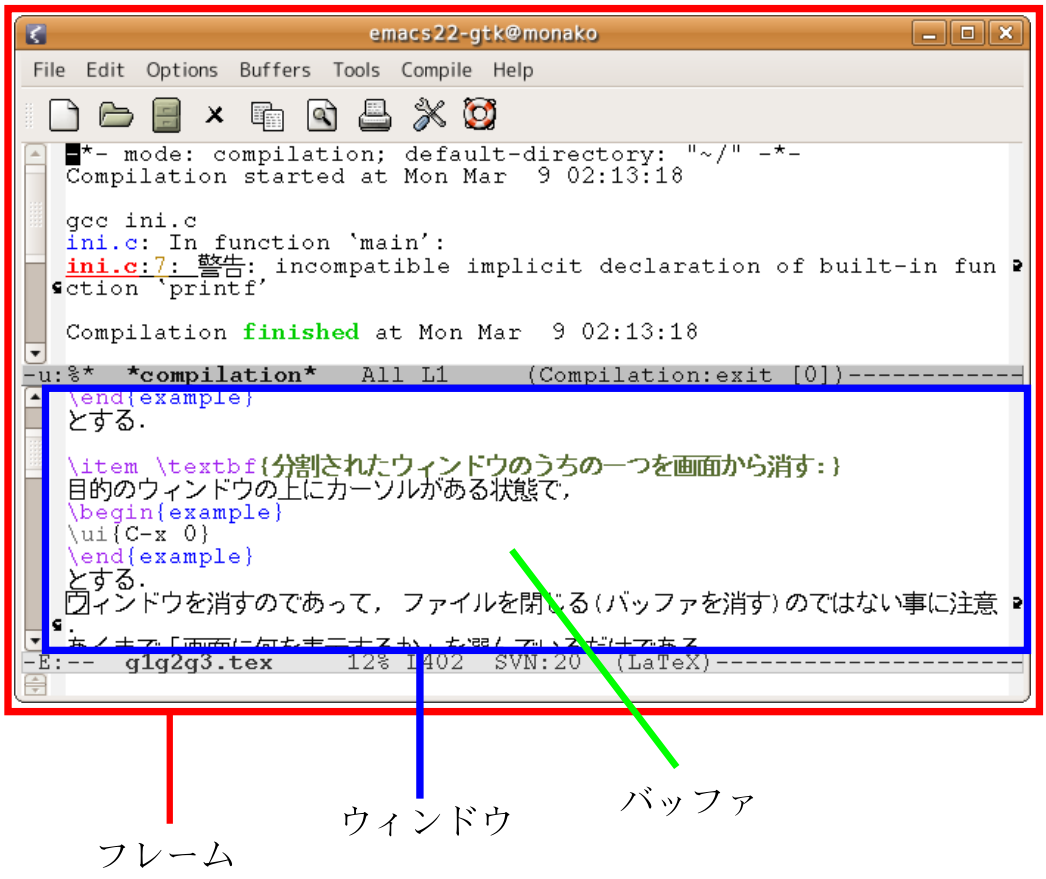

図 G1.1.3 Emacs のフレーム, ウィンドウ, バッファ

ここではそれを混乱なく行えるための最低限の概念を書いておく. 説明を誤解なく行うためにまずは言葉について (**図 G1.1.3** 参照)

フレーム : Emacs を立ち上げると出てくる四角全体を, Emacs では**フレーム**と呼ぶ. 多くのワープロソフトでウィンドウと呼んでいるものにあたる.

ウィンドウ : Emacs では, ファイルの内容や M-x compile におけるコンパイラの出力など, 一かたまりの内容を画面に表示している領域を**ウィンドウ**という. 一般にはフレームの中に 1 つまたは複数のウィンドウが存在している事になる.

バッファ : ファイルの内容や, M-x compile の結果などを保持している「もの」を, Emacs では**バッファ**と呼ぶ. 先のウィンドウと似ているが, **バッファは必ずしも画面に表示されている (ウィンドウが割り当てられている) とは限らない**事に注意が必要である.

つまり, Emacs であるファイルを開けば, そのファイルの内容を保持するバッファは常に (ファイルを閉じるまで) 存在しているが, 常にウィンドウ (画面) に表示されているとは限らない.

慣れないうちは, 自分の編集しているファイルのバッファがウィンドウから消えてしまうと, それを再びウィンドウに表示する方法が分からず, 結果としてそのファイルを編集できなくなったり, 編集する度に Emacs を起動し直したりす

る羽目になり効率が悪い. 慣れれば簡単であるし, 慣れないうちはメニューからほとんどの操作を行う事も出来るので, 以下をマスターしてほしい (**表 G1.1.1** 参照).

(1) **何はなくとも C-g:** 先にも説明したが, Emacs では一連のコマンド操作をやっている最中に訳がからなくなったら, **C-g** で脱出できる (コマンド実行前の状態に戻る). 以下をやっている最中も同様.

(2) **ウィンドウを分割する:** 分割したいウィンドウの上にカーソルがある状態で,

C-x 2 (縦分割は **C-x 3**).

または

Emacs メニュー → **File** → **Split Window**

これを繰り返すといくらでも画面を分割する事が出来る.

複数のファイルを見比べながら編集したい場合に使う. また, M-x compile やのちに説明する M-x gdb を実行するとひとりでにこれが実行される.

(3) **分割されたウィンドウ間を行き来する:** マウスで目的のウィンドウの上をクリックするか, または

C-x o

とする.

(4) **分割されたウィンドウのうちの一つを画面から消す:** 目的のウィンドウの上にカーソルがある状態で,

C-x 0

とする. ウィンドウを消すのであって, ファイルを閉じる (バッファを消す) のではない事に注意. あくまで「画面に何を表示するか」を選んでいるだけである. 画面から消えたバッファは, 「ウィンドウに表示する内容 (バッファ) を切り替える」にある方法で再び表示できる.

(5) **分割されたウィンドウのうち, 一つを除いて全部画面から消す:** 目的のウィンドウの上にカーソルがある状態で,

C-x 1

とする. 画面をすっきりさせたいときに使う.

(6) **ファイルを開く:** 別のファイルを開く (編集する) 度に新しい Emacs を立ち上げる必要はない.

C-x C-f

または,

Emacs メニュー → **File** → **Visit New File**

で現在のウィンドウに新しいファイルが表示される. もともと表示されていたファイル (バッファ) は, 閉じられたのではなく, あくまでウィンドウに表示されなくなっただけの話なので注意. 再び表示したければ, 以下「ウィンドウに表示する内容 (バッファ) を切り替える」を参照. また, 二つとも表示したければ, 適宜ウィンドウを分割して複数のウィンドウを表示してから, 目的のウィンドウ上で, 「ウィンドウに表示する内容 (バッファ) を切り替える」をやる.

(7) **ウィンドウに表示する内容 (バッファ) を切り替える:**

Emacs メニュー → **Buffers** で表示から目的のファイルを選択

慣れて来ていちいちマウスを使うのが面倒になってきたら,

C-x b

表 G1.1.1 Emacs ウィンドウ・バッファ操作

操作内容	キーボード	マウス
操作の中断	**C-g**	
ウィンドウ分割	**C-x 2** (縦分割: **C-x 3**)	**Emacs メニュー** → **File** → **Split Window**
ウィンドウ間移動	**C-x o**	移動先ウィンドウで左クリック
ウィンドウを画面から消す	**C-x 0**	
一つのウィンドウだけを画面に残す	**C-x 1**	
ファイルを開く	**C-x C-f**	**Emacs メニュー** → **File** → **Visit New File**
ファイルを閉じる (バッファを消す)	**C-x k**	
ウィンドウの表示内容 (バッファ) 切り替え	**C-x b**	**Emacs メニュー** → **Buffers**
バッファの一覧	**C-x C-b**	**Emacs メニュー** → **Buffers**

とやると,

Switch to buffer (default ...):

というのが画面下部に出るので, そこでファイル名を入力する. 一覧を表示したければそこで ⟨**tab**⟩. また, 途中まで入力して ⟨**tab**⟩ を押せば補完して (残りを補って) くれる. 一覧を表示するとそこで自分の見覚えのあるファイル名に加えて, M-x compile の結果できたバッファや, Emacs が勝手に作ったバッファも表示されるので一度注意してみよ.

(8) **存在しているバッファの一覧:**

Emacs メニュー → **Buffers**

でメニューにバッファ一覧が表示される. または

C-x C-b

でもよい.

(9) **バッファを消す (ファイルを閉じる):** 目的のバッファをウィンドウに表示して, そのウィンドウ上にカーソルがある状態で,

C-x k

とすると

Kill buffer (default ⟨ そのファイル名 ⟩):

と表示されるのでそこで Enter. 消したいファイルの名前が分かっているのであれば, いきなり **C-x k** で, 上記プロンプトに対してファイル名を入力してもよい (ここでも ⟨tab⟩ を押すと一覧が出てきたり, 途中まで入力して ⟨tab⟩ を押すと補完をしてくれたりする).

Emacs では 10 や 20 のバッファを保持しておくのはよくある事で, あまりこまめにファイルを閉じる必要はない (むしろこまめに保存 **C-x C-s** しておくこと).

5. 情報源: マニュアルについて

本資料でも必須の項目については, 操作方法を含めて説明しているが, 網羅的に説明する事はできない. コマンドの使い方, ソフトの操作方法, C 言語の関数など, 詳細な情報を自分で調べられるようになってほしい.

ここでも検索エンジンは偉大なツールだが, Linux (UNIX) では以下のような決まった操作で, 信頼できる詳細な情報が得られる. 積極的に使いこなすこと.

5.1 man コマンド

$ man ⟨ コマンド名 ⟩ **$ man ⟨ 関数名 ⟩**

などで, コマンドや C 言語の関数の詳細なマニュアルを表示する事ができる.

例:

```
$ man gcc
$ man open
$ man -s 2 open
```

man コマンドは **q** で終了する (less コマンドを起動しているので操作はそれと同じ).

-s 2 というオプションは, マニュアルの第 2 章から検索せよという意味で, man だけでは目当てでないページが表示されてしまう (例えば, C 言語で使う open という関数について知りたいのだが, man open は open というコマンドが表示されてしまう) 時に用いる. 1 章がコマンド, 2 章がオペレーティングシステムが直接提供している関数 (システムコール), 3 章がその他の関数 (ライブラリ関数) である. 実際問題としては, 関数について知りたいのにコマンドが出てきてしまう, という場合に, **-s 2** や**-s 3** を試してみる, という使い方をする事が多い.

同じ内容を Emacs 内に表示させる事もできる.

```
M-x man
Manual entry: gcc
```

のように. これは特に, 関数の使い方を調べて, 必要な#include や, 引数の並びなどをコピーするのに便利である (Emacs のコピー&ペーストをマスターしよう). この場合も目当てでないコマンドが出てしまったら,

```
M-x man
Manual entry: -s 2 open
```

などとする.

5.2 info コマンド

大きなコマンドの本格的なマニュアルは, info コマンドで提供されている事が多い.

```
$ info gcc
$ info emacs
```

など. **q** で終了, スペースでページをめくる. それ以外の操作は Emacs と似ている. ある項目を表示したい時は, **m** を打ってから項目名を入力するか, 行きたい項目の上にカーソルを持って行って **m** を打つ.

```
$ info
```

だけで起動するとすべての info ページの一覧が現れる.

これも Emacs 内で起動でき,

```
M-x info
```

とする.

ぜひ一度, 自宅などで時間を取って, 「マニュアル閲覧の練習」をしてみるとよい.

- man で man コマンドの使い方を眺めてみる
- man で top コマンドの使い方を眺めてみる
- info で info の操作方法を習得する. まずはスペースキーで通して読む. q で終了.
- info で, Emacs の使い方を眺めてみる.

例えば画面の分割や切り替えの方法に習熟すべく, マニュアルの該当セクションを探して読んでみる.

6. デバッガ

デバッガはプログラムを実行しながらその内部を調べる事ができるツールである. 具体的には以下のような事ができる.

(1) プログラムを少しずつ (例えば 1 行ずつ) 実行させる
(2) プログラムを特定の場所まで実行させる
(3) プログラムの実行が停止した場所で, 変数や式の値を表示させる
(4) プログラムの実行が停止した場所で, 関数呼び出しの履歴 (バックトレース) を表示させる

特に, プログラムが segmentation fault などを起こして異常終了した際に, デバッガを用いると, 発生した位置や, そこでの変数や式の値を表示させる事ができる. segmentation fault での終了はさもなければ手がかりを得る事が難しいので, デバッガがもっとも重宝される場面である.

6.1 デバッガの起動と終了

ここでは最低限のデバッガの用い方を説明する. 多くの部分は 4 学期の演習で習っていると思う.

ステップ 1: コンパイル : プログラムをコンパイルする際に, -g オプションをつけてコンパイルする.

> **M-x compile**
> Compile command: **gcc -g -o ini ini.c**

もちろんその他のオプションをつけてもよい. 実際, -Wall を常につけておくのはよい習慣である.

ステップ 2: デバッガを起動 : デバッガを起動するコマンドは, “M-x gdb” というコマンドである.

> **M-x gdb**
> Run gdb (like this): gdb --annoteate=3 **ini**

最後の, “ini” の部分が実際にデバッグしたい実行可能ファイル名で, 必要に応じて修正する.
上記を行うと, Emacs の画面が二つに割れ, gdb のプロンプト (以下の (gdb)) が現れる.

```
Current directory is /home/tau/
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type ”show copying”
and ”show warranty” for details.
This GDB was configured as ”i486-linux-gnu”...
(gdb)
```

この状態ではまだプログラム (ini) は起動していない. 以降 gdb のプロンプトに様々なコマンドを入力する事で, プログラムを終了まで実行したり, 少しだけ実行したり, 停止した場所での状態を調べたりする事ができる.

ステップ 3: プログラムの起動 : ともあれプログラムを起動してみる. run コマンドがプログラムを最初から実行するためのコマンドである.

> (gdb) **run Daisuke Matsuzaka**

で, ini に, 引数 Daisuke Matsuzaka を与えて実行する (つまりこれで, 端末から**./ini Daisuke Matsuzaka** としたのと同じ. コマンド名 ini 自体はここで改めて与える必要はない).

```
(gdb) run Daisuke Matsuzaka
Starting program: /home/tau/ Daisuke Matsuzaka
Initial of 'Daisuke Matsuzaka' is DM.

Program exited with code 046.
(gdb)
```

なお, run の代わりに r だけでもよい.

一般に gdb は曖昧さがなければいくらでもコマンドの名前を省略できる (例えば次に述べる quit の代わりに q だけでもよい, など).

ステップ 4: デバッガを終了する : 終了するには,

```
(gdb) quit
```

とする.

正しいプログラムをデバッガで実行してもあまり面白みはない. 今度は ini を足りない引数で実行し, 先ほど発生していた segmentation fault の発生位置を突き止めよう. もう一度 gdb で ini を実行するが, 今度は引数ひとつで実行する. ステップ 1,2 は先と同じで, ステップ 3 で,

```
(gdb) run Daisuke
```

とする. すると以下のようなメッセージが表示されるとともに, そのエラーが発生しているソースコード上の位置が表示されるであろう.

```
(gdb) run Daisuke
Starting program: /home/tau/ini Daisuke

Program received signal SIGSEGV, Segmentation fault.
0x080483a7 in main (argc=2, argv=0xbf8815d4) at ini.c:5
```

画面を見ると,

```
char f = fn[0];
```

という行でプログラムが停止している ⇒ その行の実行中に segmentation fault が発生している, という事が示されている (**図 G1.1.4**).

Segmentation fault は一般に, 不正なメモリアドレスを参照した際—C 言語の言葉で言えば, 間違ったアドレスを保持しているポインタ p を, p[i], *p, p->f などの式で参照した際— に発生するエラーであり, 上記を見ると (おそらく) fn というポインタが不正なアドレスを保持しており, それを通じて fn[0] という参照を行っているからであろうと想像がつく.

それを確かめるには, fn という変数に何というアドレスが入っているかを見るとよい. それには print コマンドを用いる. print コマンドは gdb 内で式の値を表示するコマンドである.

```
(gdb) print fn
$4 = 0x0
(gdb) print fn[0]
Cannot access memory at address 0x0
(gdb)
```

入っているアドレスは 0x0 (つまり 0 番地. 16 進数表記) であり, これは不正なポインタの代表値のような物である. もちろんアドレスを見てどのアドレスなら不正かを一般的に判断するのは難しいが, 0 を代表として, 非常に小さい数字はほぼ間違いなく不正である. 0x0 が不正である事は, その次の **print fn[0]** を実行した際のエラーメッセージ (Cannot access memory at address 0x0) で確信できる.

今回の場合, ここに 0x0 が入っていた理由は, コマンドライン引数を一つしか与えておらず, argv[0], argv[1] には文字列が入っているが, argv[2] には 0x0 が入っていたためである.

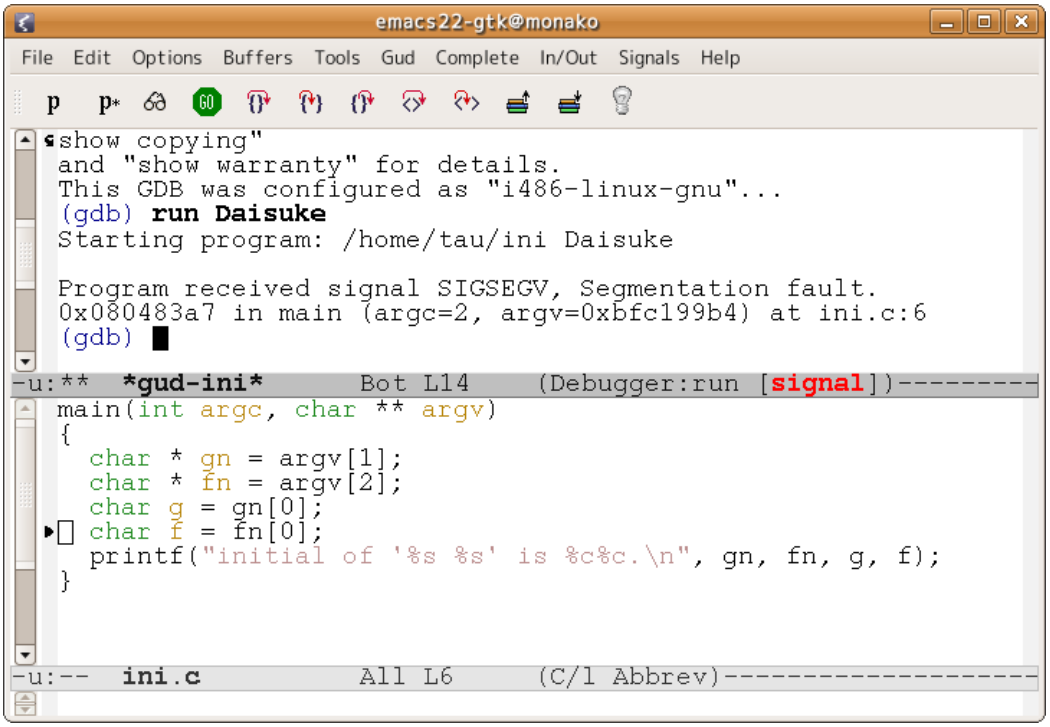

図 G1.1.4　GDB 内で segmentation fault が起きた場所を表示

6.2　関数呼び出し履歴の表示 (bt, up, down)

前節ではプログラムが停止した際, 特に segmentation fault で停止した際にソースコードの位置を示してくれる事を見たが, 最終的に segmentation fault を起こした位置だけでは役に立たず, どのような関数呼び出しを経てそこにたどり着いているのかを知りたい事がある. このための機能がバックトレースを表示する (bt) および, フレーム間の行き来をする (up/down) コマンドである.

説明のために, 先ほどのプログラムを少し修正して以下のようにする. 単に, fn[0] のように文字列の先頭を取得している部分を, get_first_char という関数を作って取得するようにした.

```
char get_first_char(char * s)
{
  return s[0];
}

main(int argc, char ** argv)
{
  char * gn = argv[1];
  char * fn = argv[2];
  char g = get_first_char(gn);
  char f = get_first_char(fn);
  printf("initial of '%s %s' is %c%c.\n", gn, fn, g, f);
}
```

同じように-g をつけてコンパイルし, デバッガで, 足りない引数で実行すると, 当然の事ながら, 3 行目の return s[0]; の行で segmentation fault が起きる.

```
(gdb) r Daisuke
Starting program: /home/tau/ini2 Daisuke
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0804837a in get_first_char (s=0x0) at ini2.c:3
```

ポインタ s の値や, s[0] を表示させる事で, 同じ事が起きていると納得できる.

```
(gdb) print s
$1 = 0x0
(gdb) print s[0]
Cannot access memory at address 0x0
```

ここで, bt (backtrace) コマンドを実行すると, どのような関数呼び出しの結果, そこへたどり着いているのかが表示される.

```
(gdb) bt
#0 0x0804837a in get_first_char (s=0x0) at ini2.c:3
#1 0x080483c2 in main (argc=2, argv=0xbfb59894) at ini2.c:10
```

この場合, main が get_first_char を呼び出しているという至極もっともな結果が表示される.

さて先ほどと違い, s=0x0 は, 関数の引数として渡された物なので, 間違いの「大元の原因」は, その関数を呼び出した地点に遡って初めて分かる. そのために用いるのが up コマンドである. up コマンドを実行すると, Emacs 上で矢印が, main 関数内の, get_first_char が呼び出された地点に移る. 今度はそこで, fn や, その他, main 関数内で定義されている変数を用いた式を評価できるようになる.

```
(gdb) up
#1 0x080483c2 in main (argc=2, argv=0xbfb59894) at ini2.c:10
(gdb) p fn
$2 = 0x0
```

容易に想像できる通り, この状態で down を実行すると注目地点が元に戻る.

6.3 ブレークポイントとステップ実行 (break, next, step, cont, fini)

デバッガを用いてプログラムを所定の位置まで実行させたり, 一行ずつ実行させる機能である. それらすべての基本は, ブレークポイントという機能で, プログラムの途中に印 (ブレークポイント) をつけた上でプログラムを実行する. プログラムの実行がそこに達したら停止する. 停止したらあとは segmentation fault で停止したときと同じで, print や bt, up, down を使ってプログラムの状態を調べる事ができる.

ステップ 1,2 : プログラムをコンパイルし, M-x gdb でデバッガを起動するところまではこれまでと同じ.

ステップ 3 ブレークポイント設定 : run コマンドでプログラムを走らせる前に, 止めたいところにブレークポイントを設定する. ここではプログラムの先頭で止まるように, main 関数に設定する.

```
(gdb) break main
Breakpoint 1 at 0x8048393: file ini2.c, line 7.
```

break は b でもよい.

ステップ 4 プログラムの起動 : 後は通常どおりプログラムを走らせる.

```
(gdb) r
Starting program: /home/tau/ini2

Breakpoint 1, main (argc=1, argv=0xbf8f3634) at ini2.c:7
```

するとプログラムが main 関数の先頭で停止する. Emacs 上でソースファイル上の停止位置に印が表示される.

この状態から以下のような手段でプログラムを「少しずつ」実行できる.

break と continue : もう少し先に別のブレークポイントを設定してそこまで進める. この場合, 進めるために run コマンドではなく, continue を使う (run は常にプログラム最初からの実行).

step : 1 文実行する. その文に関数呼び出しが含まれていたらその関数の中に突入する (つまり呼び出された関数の先頭で停止する).

next : 1 文実行する. step と異なり, あくまで今いる文の実行が終わるところまで進む.

これらを組み合わせてプログラム実行中のどの時点で, どのような状態になっているかを調べて行く事ができる.

なお, ブレークポイントは以下の場所に設定する事ができる.

関数の先頭 :

(gdb) **break** ⟨ **関数名** ⟩

例:

(gdb) **b main**

ソースコード上の指定位置 :

(gdb) **break** ⟨ **ソースファイル名:行番号** ⟩

例:

(gdb) **b ini.c:5**

Emacs のキー操作 : 前項と同じ事を, Emacs でソースコード上のブレークポイントを設定したい行にカーソルを移動して, **C-x SPACE** を打つ事でできる.

6.4 デバッガ: まとめ

コマンドとしては以下辺りを抑えておくとよい. display, finish は本資料では説明していない. gdb の使い方についても是非, info などで一度時間を取ってみてみるとよい.

- q(uit)
- r(un)
- p(rint), disp(lay)
- bt, up, down
- b(reak), n(ext), s(tep), c(ont), fini(sh)

7. C プログラムのデバッグ

今回の課題は C プログラムの間違いを修正する事である. C 言語に関する復習, デバッガの使い方の復習とレベルアップ, より高次元な, 「デバッグの仕方 ($\neq$ デバッガの使い方)」を身につける練習と思って, クイズ気分で取り組んでほしい.

本課題 1.7 課題 HP にアクセスすると, たくさんの「間違った」プログラムの入ったファイルがおいてある. それらに含まれる間違いを修正し, 正しく動かせ. コンパイルする際は最終的には, -Wall オプションをつけて, 警告が出なくなるようにせよ.

手順: ファイル problems.tar.gz をダウンロードしたら, それを以下で展開する.

```
$ tar xvf problems.tar.gz
```

すると, problems の下に多数のサブディレクトリ 00, 01, . . . , ができる. 一つのディレクトリが一問に相当する. 各ディレクトリのファイルをコンパイルして実行し, 正しい動作をさせる事が目標である. コンパイル時にエラーになったらそれを読んで修正する. 無事コンパイルできたら実行する. 実行の仕方 (与えるべきコマンドライン引数など) と, そのプログラムが意図している計算内容は, プログラム先頭にコメントとして書いてあるので, それを読むこと. 引数を与える場合, いろいろな引数で正しく動く事を目標にする.

例 1:

```
$ cd 00
$ gcc p00.c                    # ここでエラーが出たら修正
$ ./a.out                      # ここでエラーが出たり動作がおかしければ修正
```

もちろんコンパイルには M-x compile を使う事を推奨する.

例 2:

```
$ cd 03
$ gcc p03.c
$ ./a.out 3                    # 引数については p03.c のコメントを読む
3^2 = 8                        # 間違い! 修正
```

いろいろなレベルの間違いが含まれている. それらに対し, その意味するところを正しく把握して, 修正するのが課題である. 正しいプログラムにするために必要な修正の量は, 文字数・行数としてはどれもわずかである. また, 個々のプログラムは一から書いてもすぐに書けてしまうような単純な物もある. だからといって与えられたプログラムを無視して, ともかく動くプログラムを書いて, 「はいできました」というのはこの演習の意図するところではない. あくまで, 与えられたプログラムの, 「どこが, なぜ間違っているかを求める」のが目標である. 「自分は医者, プログラムは患者」と言うつもりで, 症状から間違いを「診断」「説明」してほしい. ヒントとして, どのような段階でどのような種類のエラーが待ち受けているのか, そのいくつかを列挙しておく (以下は今後も実際に遭遇する, 典型的なエラーである).

コンパイル・リンク時のエラー : gcc がエラーを出力する.

- 文法エラー
- 型エラー
- 関数宣言の欠如に関する警告
- リンクエラー. プログラム中で使用している関数がどこにも定義されていない, というエラー. これを直すには gcc を起動するコマンドラインに, あるオプションを追加する必要がある.

実行時のエラー : 実行可能ファイルができるが, 実行するとエラーメッセージを表示して途中で終了する.

- segmentation fault
- assertion エラー (“Assertion ‘x == y’ failed.” のようなメッセージ)
- その他のエラー (“fopen: No such file or directory” のようなメッセージ)

動作・出力の間違い : 実行可能ファイルができ, 実行してもあからさまなエラーはでないが, 答えなり, 動作なりがおかしい.

もちろん最後のタイプのエラーが, 実際エラーであるとわかるためには, そもそもそのプログラムが何をする「はずの」プログラムなのかが, どこかで規定されていなくてはならないが, 出力から常識的に判断してほしい (例えばプログラムの出力が, “$10 + 20 = 50$” だったら, それはおかしいという事).

プログラム自身は高度でもなければ, 役に立つ物でもない. あくまで, C 言語の規則をよく知る事, プログラムの実行規則を知る事, 問題究明のための考え方を習得する事, そのための道具 (デバッガなど) を習得する事が目的である. 具体的には次のような概念や道具を「練習する」事を心がけること.

- コンパイル時のエラーは, きちんとエラーメッセージを読んで, 何を言われているのか, なぜそれがエラーになるのか, C 言語の規則を理解すること (当てずっぽうでプログラムを修正しない).

- M-x compile と, **C-x ‘**によってコンパイルエラーの位置へ自動的に飛ぶ, を使いこなすこと
- segmentation fault など異常な終了の仕方をしたら, ソースを当てもなく眺めるのではなく, まずは M-x gdb で, それが発生している場所を突き止めること. 当てずっぽうで直す, の前になぜそれが起きるのかを理解すること. segmentation fault は触ってはいけないメモリ番地を触った時におこる. それが起きる理由のほとんどは, 配列の添字にその要素数以上の値を指定した場合, ポインタ変数に間違った値を入れ (あるいは値を入れ忘れ) てそれを通じてメモリをアクセスした場合, である (*p, p[i], p→f など).
- assertion エラーは発生位置 (ソースファイル名と行番号) がひとりでに表示されるが, やはり M-x gdb で突き止めるのも簡単である.
- その他の実行時エラーについても, プログラムが終了する際に呼ばれる exit, abort などの関数にブレークポイントを仕掛け (break exit) て実行する事で捕捉できる. up コマンドを使って exit/abort を呼び出した地点に戻る事で, 問題の発生場所を突き止められる.
- 一般に, プログラムの入力を変えられる場合, その入力を色々かえて挙動を探る. その際, 「エラーが出るなるべく小さな入力を突き止める」という考え方が重要である. その方が圧倒的にデバッグは楽になるし, それができた時点で原因が推測できてしまう事も多い.
- プログラムの挙動を調べるのにはプログラムがどこを実行しており, その時に変数の値が何であるかを調べるのが基本である. printf をはさんで要所で変数や式の値を表示する, いわゆる “printf デバッグ” は有効である.

8. Linux を便利に使うためのおまけ

8.1 ランチャにアプリケーションを登録する

よく使うアプリケーションを画面左のランチャに登録しておく事ができる.

(1) Dash ホーム + 適当なキーワードで目的のアプリケーションを, 表示させる. 例えばスクリーンショットをよく撮りたいなら, Dash ホーム +screenshot.

(2) その状態で目当てのアプリケーションのアイコンをマウスの左ボタンを押しながら, ランチャまでドラッグする

8.2 ソフトウェアパッケージをインストールする

演習で使う多くのソフトウェアはすでにインストールされているが, 新しいパッケージを入れるのも簡単なので, やり方を覚えておくと良い.

方法 1 GUI :

(1) ランチャから, Ubuntu ソフトウェアセンターを選択

(2) 好きなアプリケーションを選んで「変更の適用」.

これは, 「どんなアプリ (ゲーム?) があるのだろう」というようなときに有用である.

方法 2 apt-get コマンド : インストールしたいソフトの名前 (パッケージ名) が分かっている場合, root ユーザになった後, コマンドラインで, apt-get コマンドでインストールできる. 例えば “iperf” というパッケージを入れると言う事になったら,

```
$ sudo apt-get install iperf
```

でよい.

どのようなパッケージがインストールできるかの検索は,

```
$ apt-cache search 〈 キーワード 〉
```

でできる. ゲームを探したいのなら,

```
$ apt-cache search game
```

また, 「使いたいコマンド名」がそのままパッケージ名になっている事が多い. したがってあるコマンドを使って, 「そんなコマンドはない」といわれたら,

```
$ sudo apt-get install 〈 ないといわれたコマンド名 〉
```

とすれば, 10 秒後には使える状態になっている事が多い. それ以前に, Ubuntu の場合, 有名なコマンドについては, 「そんなコマンドはない」と言われたときにパッケージ名を教えてくれる事もある.

```
$ sl
プログラム 'sl' はまだインストールされていません。 次のように入力する事でインストールできます:
sudo apt-get install sl
bash: sl: command not found
```

GUI を用いた方法で, 有用そう, 面白そうなアプリケーションを探してインストールしてみよ. apt-cache search でも探してみよ.

第2日　ファイルの読み書きと, 音データの入出力

今回はファイルの読み書きを中心に行う.

2 年 4 学期に C 言語の標準ライブラリ (fopen, fprintf など) を用いたファイル入出力をやったと思うが, ここでは UNIX のシステムコールを用いたファイル入出力について紹介する. 関数名としては, open, read, write, close の 4 つを用いる. 両者の違い, なぜこの実験でこちらを用いるのかについても軽く触れる. その過程で, ファイルの中身を「バイトの列」として正しく理解できるように, 別の言葉で言えば「バイナリファイルの読み書きが混乱なくできるように」なってほしい.

次に, ある特別なファイル (/dev/dsp) を読み書きするだけで, 音の入出力が行える事を説明する. したがってプログラミングとしては, **ファイルの入出力ができれば録音・再生が行える** 事になる. 音を表現するデータ形式も非常に扱いやすい物で, 各時刻における振幅をそのまま並べただけの物 (Linear PCM) である.

最後に, 後々波形データを解析したり変換したりするための助けとして, グラフを可視化するツール gnuplot について学び, 録音・再生される音を可視化する.

1. ファイル入出力

1.1 ファイルへの出力とファイルの作成

まずはファイルにデータを書き込む練習をしてみよう. 基本フォームは以下のようになる.

ステップ 1: ファイルを開く :

```
int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

フラグがややこしいが, さしあたってファイルを作るときはこうする, と覚えておけば良い.

- O_WRONLY で, *filename* を書き込みモードで開く事,
- O_CREAT で, *filename* がなければ作る事,
- O_TRUNC で, *filename* が存在していても一度空にする (一から作り直す) 事,
- 最後の 0644 は, もしこれで *filename* が作られた場合, 読み書き権限を誰に与えるかという事を指定しているが, とりあえずこの実験では 0644 以外は使わない.

このパターンはよく使うので, 上と同じ事をする creat というシステムコールもある. それを使って上記を creat(*filename*, 0644) と書いても良い.

ステップ 2: 書きたいデータを配列に準備する : 実際に書きたいデータを作るところだから, もちろんプログラムによって千差万別である. 以下はあくまで一例.

```
unsigned char data[N];
data[0] = ...;
data[1] = ...;
...
```

ステップ 3: データを書く :

```
write(fd, data, N);
```

これで, data[0], data[1], ..., data[N−1] の N バイトが *filename* に書き込まれる. 一般に, write(fd, a, n) は, アドレス a から始まる最大 n バイトを, fd (が表しているファイル) に書き込む, という事である.

ただし, 要求した N バイトすべてが書き込まれるとは限らず, それ未満のデータ (1 バイト以上) だけが書き込まれる事もあるので注意が必要である. 実際に書き込まれたバイト数が返り値になる. 失敗した場合は何が返るか? き

ちんとマニュアルで調べよ (**第 1 章**, **5.** 節参照). もちろん, write は好きなだけ繰り返し呼び出してよい. その場合, write を呼び出した順に, ファイル内にデータが順に格納される.

ステップ 4: ファイルを閉じる :

```
close(fd);
```

これ以上 write でデータを書き込む事はできなくなる.

まず open というシステムコールを呼ぶと, そのファイルに書いてよい証しとして, 整数 (**ファイルディスクリプタ**と呼ばれる) が返される. あとは個々の読み書きや, 使い終えた後ファイルを閉じる際に, ここで返された値を渡す.

open と write の関係は, ディズニーランドのなんとかパスポートと, 乗り物の関係に似ている. 一度券売り場でパスポートを買い (open), それを手にしたら個々の乗り物に乗る (write する) ときはそのパスポートを見せれば (fd を引数に渡せば) よい. ファイル入出力に限らず, このようなパターン—最初の呼び出しで「何か」が返されて, その返された「何か」をその後呼び出す関数に渡す—はいろいろな場面で現れる.

準備課題 2.1 ファイルの第 0 バイト目に 0, 第 1 バイト目に 1, . . . , 第 255 バイト目に 255, が入っているような, 256 バイトのファイル my_data を作る C プログラム mk_data.c を作れ. mk_data.c は, open, write, close を用いてファイル my_data を作成する.

そのプログラムを無事コンパイルするには, いくつかのヘッダファイル (.h で終わるファイル) を#include する指示を書かなくてはならない (#include <????.h>). **どのファイルを#include すればよいのかを, man コマンドで調べて**, 正しくコンパイルせよ (**第 1 章**, **5.** 節参照). 以降も, C 言語で提供されている関数を使うには, **あるヘッダファイルを#include しなければならない, という場合がしばしば現れるが, 一々断らない.**

準備課題 2.2 mk_data.c が完成し, 無事 my_data ができたら, ls コマンド (-l オプション) を用いて, 実際にファイルが 256 バイトである事を確認せよ. また, そのファイルを cat コマンドや Emacs で開いて, どんな風に見えるか, 見てみよ.

準備課題 2.3

- 第 0 バイト目に 228
- 第 1 バイト目に 186
- 第 2 バイト目に 186
- 第 3 バイト目に 229
- 第 4 バイト目に 191
- 第 5 バイト目に 151

という, 6 バイトからなるファイル hitoshi を作るプログラム hitoshi.c を作り, 同じ事をしてみよ.

注意: 例えば最初の問題に対して以下のような結果を期待しているのではない.

```
012345678910111213...253254255
```

同様に 2 個目の問題に対して, 以下のような結果を期待しているのではない.

```
228186186229191151
```

言い換えればここで言う「第 0 バイト目に 0」というのは, あくまで「整数の 0」, あえて 2 進数で書けば 00000000 という 8 ビットの事を言っているのであって, 文字の'0' の事を言っているのではない.

この話がよく分からないと感じたら以下の段落を読んで, ファイルの中身に何が格納されているのかについて理解をしておいてほしい.

すべてのファイルは, バイト列 (ビット列) に過ぎない

(1) ファイルは言うまでもなく**データ**を格納した物である.
(2) そのデータとは, **バイト**がずらずらと並んだ物である.
(3) 1 バイトは 8 bit の事であったから, 1 バイトは 256 通りの値を区別できる.
(4) それを 0 から 255 までの数字だと思えば, 要するにあらゆるファイルの中身は,

「0 . . . 255 までのどれかの数字」がずらずらと並んでいる

物なのである.

もう少し具体的に言えば, オペレーティングシステムがプログラムに対して提供しているファイルの入出力機能が,

- このファイルから X バイト読ませて下さい ⇒ 0, . . . , 255 までの数字が X 個並んだ物が返ってくる
- 0, . . . , 255 までの数字が X 個並んだ物を渡して, このファイルに X バイト書かせてください ⇒ それをファイルに書いてくれる

というだけの物であって, すべてはこれを基本としている, という事である.

言われるまでもない当たり前の話と映るかもしれないが, **これまでファイルの読み書きはテキスト (文字) しかやった事がない, という人は混乱の元となるかもしれないのでよく注意してほしい.**

ともかくわかっておくべき事は, ファイルに格納されている物は (どんなファイルであっても), 「0 . . . 255 までのどれかの数字」がずらずらと並んだ物であり, ファイルの中にそれ以外の物—「文字」や絵—などという物はない. この様に言うと, いやちょっと待て, 先学期習った fprintf 関数で,

```
fprintf(fp, "hello\n");
```

のように書けばファイルに, hello という文字列が書かれ, それはエディタでそのファイルを開けば確認できるではないか, これは「文字」を書いているではないのか? という疑問を持つ人もいると思う. しかし実は, 上記と「まるっきり同じファイル」は, 以下のようにしても作る事ができるのである.

```
unsigned char data[6];
data[0] = 104;
data[1] = 101;
data[2] = 108;
data[3] = 108;
data[4] = 111;
date[5] = 10;
write(fd, data, 6);
```

この様にしてできた二つのファイルはまるっきり同じである. この意味において, すべてのファイルはバイト列に過ぎない, それ以外の物は入っていない, という事なのである.

バイト列に過ぎないデータが, エディタで文字として表示されたり, 画像として表示されたり, 音楽として流れたりするのは, すべて, プログラムが「ある約束 (符号化方式. ファイルフォーマット)」にしたがってバイト列を解釈しているおかげ, という事である.

符号化方式のうち最も単純で, お馴染みの物は, 英数字を符号化した, ASCII 符号という物で, 例えば a という文字に 97, b に 98, c に 99, . . . , を割り当てている. この他「改行」に 10 を割り当てている. だから上述の, hello\n は, 104, 101, 108, 108, 111, 10 という並びになる. 最初の問題で Emacs でファイルを開いたときに, 一部に abcdefg. . . のような見慣れた文字が出現した事と思うが, それはこのような事情による. 他にも, 漢字には漢字のための, 音には音のための, 静止画には静止画のための, 動画には動画のための符号化があるのである.

くどくどと述べてきたが, 要するにファイルから入力してきたデータが 10, 20, 30, 40 というバイト列である事と, 文字列"10 20 30 40"であるという事は大違いで, それをうっかり勘違いしていると, 今後音声などのデータを扱うときに意味不明な事になるから注意せよという事である. ちなみに後者の文字列 ("10 20 30 40") はバイト列としては, 49, 48, 32, 50, 48, 32, 51, 48, 32, 52, 48 である. どんなデータもバイト列だと今一度強調するために断っておく.

1.2 od コマンド

ファイルにどのような「バイト列」が入っているのかをありのまま見せてくれるコマンドがある.

```
$ od -t u1 ファイル名
```

のように使う.

準備課題 2.4 od コマンドを用いて, my_data を表示して, 予期した通りの結果になっている事を確認せよ. hitoshi についても同じ事をしてみよ.

od は, -t コマンド (t は type の意味) で, バイト列の「解釈の方法」を様々に変えて, 同じファイルを表示する事ができる. 上記で用いた`-t u1` は, ファイルをバイトの並びとみなし, かつ個々の 1 バイトを unsigned とみなして表示せよという意味である. つまり, 個々のバイトを 0, . . . , 255 の整数とみなせ, という意味である. これが, `-t d1` なら, 個々のバイトを符号付き整数とみなす— −128, . . . , 127 の整数とみなす—という意味になる. `-t u2` なら 2 バイトずつをまとめて 16 ビットの整数 (0, . . . , 65535) とみなす. もう分かると思うがこれらはすべて, 「同じバイト列を」見ているに過ぎない. そのバイト列を「どう解釈するか」その方法に色々あるという事の例に過ぎない.

od コマンドは, ファイルにデータを書いたつもりなのに結果が思うようにならない, とか, ファイルからデータを入力しているつもりなのに思うようにならない, などのとき, 診断ツールとして有用である.

1.3 コマンドライン引数

ほとんどの実用的なプログラムは, ちょうど od コマンドの例で見たように, 扱うファイル名や, 動作を変えるためのオプションをコマンドライン引数として受け取る. 我々が最初の 2 問で作ったように, どんな場合でも my_data という名前のファイルを作る, などという頑固な事にはなっていないのが普通である.

ここではそのようなプログラムを C で作るときのやり方を説明する. 難しい事はなく, コマンドラインで与えた引数を, 自分のプログラムの中で「受け取る」方法を理解すれば良いだけである.

(1) コマンドラインで与えた文字列は, main 関数の引数として渡される.

(2) main 関数には二つの引数が渡されており, それを受けとるには以下のようにする.

```
main(int argc, char ** argv)
```

変数名はどうでもよく, 型 (int と char **) とその順番が重要である.

(3) 第一引数 (argc) には, コマンドラインで与えた文字列の数 (コマンド名を含む) が入り, argv にはそれらの文字列の配列が入る.

(4) 例えば,

```
$ ./mk_data foo
```

というコマンドラインに対しては,

- argv[0] = "./mk_data"
- argv[1] = "foo"

が格納され, したがって, argc = 2 となる (要するに, argv の要素数を与えているのが argc).

準備課題 2.5 ./mk_data コマンドをもう少しマシな物にするために, データを書き出すべきファイル名をコマンドラインから受け取るようにせよ.

1.4 エラー検査

ファイルを作成する際の基本フォームが

```
int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

であると述べたが, 実はこれでは足りない. **実際にファイルを開けたかどうかを検査しなくてはならない.** 例えば開こうとしたファイルが, 書き込み禁止 (不可) なディレクトリにあるかもしれず, そのような場合はファイルを開く事はできない. 後にファイルを読み込む練習をするが, その際にタイポなどで, 存在しないファイルを読もうとすれば, もちろんファイルを開く事はできない. 後の週でネットワークプログラミングを行うが, ネットワークではその他にも様々なエラーの要因 (設定ミス, 通信相手がいない, ネットワーク自体が不調, など) がある. ライブラリ関数の使い方を間違っている, という事だってある.

教訓は, ライブラリ関数などを呼び出したら, **それが成功したか否かを必ず検査しなくてはならない**, という事である. その調べ方は関数によって違うが, たいがいの場合, 返り値で成功・失敗が区別できるようになっている. 例えば open という関数は, 開くのに失敗すると, −1 を返すと決められている. したがって簡単な基本フォームは以下のようになる.

```
int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd == −1) { /* エラー時の処理 */ }
```

「エラー時の処理」と言っても具体的には何をすれば良いのだろうか? 多くのライブラリ関数は, エラーを返した直後に, perror という関数を呼び出すと, その原因を表示してくれる. したがって完全な基本フォームは以下のようになる.

```
int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd == −1) { perror("open"); exit(1); }
```

exit(1) は, プログラムを終了する関数である. したがってこのプログラムは, ファイルを開くのに失敗すると, その理由を短く述べて終了するようになる.

ここで perror や exit などの新しい関数が出てきた. man を使って#include すべきヘッダファイルを入れる事を忘れずに.

準備課題 2.6 上記のようなエラー処理を施したプログラム mk_data2.c を作り, open の引数に次の入力を与えて起動してみよ. どんなエラーメッセージが出力されるか.

- **/foo**
- **bar/baz** (ここで, bar は存在しないディレクトリとする)

もし, ライブラリ関数を呼び出す度にこれをやるのが面倒という人は,

```
void die(char * s)
{
  perror(s);
  exit(1);
}
```

という関数を作っておき, 失敗したらこれを呼び出すようにすれば多少楽になる.

なぜこれが重要か? この「基本フォーム」を見て, 「なんだ, 結局失敗したらただ終了するだけなのか」と, ガクッとくるとともに, 「失敗は失敗, どうせ失敗するんだったら何が起きても同じ. エラーメッセージなんか表示したって, 慰めにしかならない」と思ったそこのキミ, その考えは即刻改めなくてはならない.

エラー検査の一番の目的は,

エラーが起きたら**そのまま先へ進まない**

という事である. それは, エラーが起きたまま処理を先へ進めると, プログラムは**もっとわけの分からない挙動を示す**からである. 仮に上のプログラムで, 何かの理由で open が失敗したとしても, プログラムの処理自体は先へ進んでしまう. そして, write システムコールまで到達する. もちろんファイルは開かれていないのだから実際にデータがかけるはずはない. そこで write システムコールが失敗した事も検査をせずに先へ進めば, 処理が失敗しているのに気づかないままどんどん先へ進み, プログラムが一見正常に終了するように見えるところまで進む事もありうる. にもかかわらずファイルができていないとか, ファイルが最初から存在していた場合, その中身が書き換わっていない, などの意味不明な状態に陥る.

たとえ話としては, 社保庁が年金記録漏れ問題を長年放置したために, 今や何がどうなっているのかさっぱり分からない状態になっている, というのと同じである. 何事も最初に間違いが起きたときに気づいて行動を起こすべきなのである.

準備課題 2.7 write の挙動と, 返り値を man コマンドで調べてみて「書き込みが成功した」事を確認するにはどのような検査を入れればいいか, 考え, 以降実践せよ.

エラー検査を省略してしまいたくなる気持ちは以下のような物だろう.

- 自分はまだプログラミングが不慣れで, 数行書けばすぐに間違える状態である. そんな時にプログラムの行数を増やしたくない.
- エラーが起きたときにする事はどうせつまらない事なので, あまり考えたくない.

繰り返しになるが, ここで実践せよと言っているのは大げさな, エラーからの回復 (ユーザにファイル名を聞き直すなど) などではなく,

```
if (... 失敗 ...) { perror("なにか一言"); exit(1); }
```

の「1 行」を加えるという単純かつ決まりきった事なので, 上記の心配は当たらない. 大して考える必要もなく, プログラムの行数も 1 行増えるだけ. にもかかわらず, それがプログラムを完成させるまでの時間を短縮するのである. 「急がば回れ」の精神なのだ, と思って, 必ず, **エラーを検査する—というよりも, 「成功を確認せずに先へ進まない」**—事を癖にしてほしい. だからやるべきは, 「エラー検査」というよりも「成功確認」と言うべきである. 石橋を叩いて渡る, ということわざもある.

実を言うとファイル入出力くらいならこれを守らずとも何とかなるのかもしれないが, 通信がからんでくると, 設定間違いや, 通信媒体, 通信相手の不調による, 「不可避なエラー」という物がたくさん出てくる. その時に「誰が悪いのか」をすぐに突き止める手段として, 常に, 「自分は悪くない」という事を確認しながら先へ進む事が必須になってくる.

1.5 読 み 込 み

次にファイルからデータを読み込む練習をしてみよう. 基本フォームは以下のようになる.

ファイルを開く :

```
fd = open(filename, O_RDONLY);
```

O_RDONLY で, ファイルを読み込みのために開く事を指定している. もちろん直後のエラー検査はすること. 今後の説明では一々断らない.

データを読み込むための領域を配列として準備する : 例えば,

```
unsigned char data[N];
```

データを読む :

```
n = read(fd, data, N);
```

これで, data[0], data[1], . . . , data[N−1] に, 読み込まれたデータが格納される.
read(fd, a, n) は, fd (が表しているファイル) から最大 n バイト読み出し, それをアドレス a から始まる場所に書き込む.
ただし write と同様, N バイトぴったり読み出されるとは限らず, それ未満のデータだけが読まれる事もあるので注意が必要である. 実際に読まれたバイト数が返り値になる. 失敗した場合は何が返るか? もちろんマニュアルで調べよ. もちろん, read は好きなだけ繰り返し呼び出してよい. その場合, read を呼び出した順に, ファイル内のデータが順に読み出される.

ファイルを閉じる :

```
close(fd);
```

これ以上 read でデータを読み込む事はできなくなる.

注: ファイルの「終わり」まで読むには?

```
n = read(fd, data, N);
```

は, **最大** N バイトのデータをファイルから読み込もうとする. この時, 実際に読まれたデータのバイト数 (1 以上) が, 返り値として返される. エラーの場合は −1 が返される. **ファイルの終わりまで読みきって, もうデータがないというときは 0 が返される.** これが, 「エラーもなく, ファイルが最後まで読めた」という事の証となる. 結局, 「ファイルの終わりまで読む」基本フォームは以下の通り.

```
while (1) {
  n = read(fd, data, N);
  if (n == −1) { perror("read"); exit(1); }
  if (n == 0) break;                    # ファイルの終わり
  . . . data に何か処理をする . . .
}
```

もちろん何バイト一度に読むか (上記の N) は自由である.

ファイルの終わりを通常その名の通り, END OF FILE と呼び, EOF と書く. ファイルを読もうとして, EOF が検出された (例えば, read が 0 を返した) 場合, 比喩的な表現として, 「EOF を読んだ」などという事がある. もちろん実際には, データを読んだわけではない. 「EOF が返された」などという事もある. 実際には 0 が返っているのに. これらは比喩的な表現です.

本課題 2.8 コマンドラインで指定されたファイルを open で開き, 何バイト目が何というバイトだったかを, 2 カラムで (第一カラムがバイト数 (最初のバイトを 0 バイト目とする), 第二カラムがその値) 表示するプログラム read_data.c を書け. 例えばこのプログラムを作って, 先ほど作った my_data を読ませれば当然の事ながら,

```
0 0
1 1
2 2
...
254 254
255 255
```

のように表示されるべきである. それを確認せよ.

これはまさしく, od コマンド (を, 単機能にして出力形式を変えた物) を作っている事になる.

注: 読み書き同時 open ファイルを書き込みと読み込み両方で開く必要がある場合は, O_RDONLY, O_WRONLY の代わりに, O_RDWR を用いる.

```
int fd = open(filename, O_RDWR | O_CREAT | O_TRUNC, 0644);
```

トピック: fopen/fprintf と, open/read/write の関係 2 年 4 学期に, ファイル入出力を行うための関数として, fopen, fprintf, fscanf などの関数を習った事だろう. 例えば以下のように使う.

```
FILE * fp = fopen(filename, "wb");
if (fp == NULL) { perror("fopen"); exit(1); }
fprintf(fp, "hello %s san!\n", argv[1]);
```

この機能は C 言語の「ストリーム入出力ライブラリ」と呼ばれている.

いわばファイル入出力に最低でも二通りの流儀 (open/read/write を用いる流儀と, fopen/fprintf などを用いる流儀) がある事になる. 両者の違いとしてまず最初に気づく, 関数名以外の違いは, open が整数を返すのに対して, fopen は FILE *という構造体へのポインタという物を返す事である. したがって fopen で開いた結果の fp を, write/read などのシステムコールに (そのまま) 渡す事はできないし, 逆も同様である.

以前に open の返り値がディズニーランドのパスポートだと述べたが, このたとえ話を (無理やり) 続けると, fopen の返り値は, ディズニーランドまでの電車の切符, ディズニーランドパスポート, アンバサダーホテル宿泊券, ホテルと会場の往復バスなどがすべてセットになった, クーポン券のような物である. 実は open が「最も基本」的な機能であり, fopen も実際に open を使って最終的にはファイルを開いている. そして実際, FILE というデータ中にはこっそりとファイルディスクリプタ (open の返り値) が含まれている (クーポン券の中の一枚に, ディズニーランドのパスポートが含まれているのと同じ). 同様に, fprintf などの機能も, 適宜 write システムコールを用いる事で実現されている (乗り物に乗るときはクーポン券ではなく, 結局ディズニーランドのパスポートを見せる). fopen/fprintf などのライブラリは, open/read/write の周りに, ファイル入出力をより便利にするためのおまけを色々くっつけた物と思えば良い.

- 例えば read/write システムコールは, 配列とその長さ (バイト数) を指定した読み書きしかできないプリミティブな物だが, fprintf は上記のように文字列の中に, 別の文字列 (%s) や整数の値を文字列化した物 (%d) を埋め込む機能を持っている.
- 読み込みに関しては, fgets(data, N, fp) のような, ファイルから 1 行読み込む (改行文字が現れるまで読み込む) などの, よく使う物が用意されている.
- read/write により近い物として, fread/fwrite という関数がある. 引数などもよく似ている.

また, fopen とその周辺の関数たちは, C 言語の標準ライブラリとして規定されている関数たちで, UNIX であろうと Windows であろうと使える. 一方 open/read/write は UNIX のシステムコールであって, 基本的には UNIX 固有の物である, という違いもある.

こう聞くとじゃあなぜわざわざこの実験では, プリミティブなシステムコールの方をわざわざ使わせるのか? と疑問に思うかもしれない. その理由はいくつかある.

- fopen/fprintf/fwrite/fread なども結局はシステムコールを使って実現されている. そのため後者の方が動作が単刀直入でわかりやすいという事がある. 例えば前者にはバッファリングという機能があり, いくつかの書き込み要求をメモリ上に貯めておき, 一回の write システムコールで書きにいく, というような事をする. したがって書き込みを行った瞬間に出力が行われるとは限らず, 「音が思ったタイミングで鳴らない」などの問題を究明する時の未知数が一つ増えてしまう.
- 今回の実験で音の入出力をするに当たっては, fprintf が提供するような文字列を便利に入出力する機能は不要で, システムコールを用いたからといって話が難しくなる事はあまりない.

2. 標準入出力とリダイレクト

ファイルを読み書きしたければ, open システムコールでファイルを開き, 返されたファイルディスクリプタを read/write システムコールに渡す, というのが基本だが, UNIX では, open しなくても「最初から開かれている」ファイルディスクリプタが存在する. それが**「標準入出力」**という物で, 以下の 3 つである.

0 : **標準入力**とよばれ, read システムコールを用いて読み込む事ができる. これを読み込むと, 通常はキーボードからの入力を読み込む事になる. 例えば以下はキーボードから 10 バイト読み込む.

```
read(0, data, 10);
```

1 : **標準出力**とよばれ, write システムコールを用いて書き込む事ができる. これに書き込むと, 通常は端末へ出力する事になる. 例えば以下は端末にデータを 10 バイト書き込む.

```
write(1, data, 10);
```

2 : **標準エラー出力**とよばれ, write システムコールを用いて書き込む事ができる. これに書き込むと, 通常は端末へ出力する事になる. 例えば以下は端末にデータを 10 バイト書き込む.

```
write(2, data, 10);
```

なぜ標準出力と, 標準エラー出力の二つが用意されているのかは後述.

それぞれ, 同じ機能の, ストリーム入出力ライブラリ版とでも言うべき物が存在しており, それぞれ, stdin, stdout, stderr と表記する (#include <stdio.h> する必要がある). だから,

```
read(0, data, 10);
```

は

```
fread(data, 1, 10, stdin);
```

と, ほぼ同じ意味.

```
write(1, data, 10);
write(2, data, 10);
```

はそれぞれ,

```
fwrite(data, 1, 10, stdout);
fwrite(data, 1, 10, stderr);
```

とほぼ同じ意味になる. そしてよく使う fprintf も,

```
fprintf(stdout, "hello %s san\n", name);
fprintf(stderr, "hello %s san\n", name);
```

と書けばそれぞれ標準出力, 標準エラー出力に書き込む事になる. そして前者が普段よく用いている printf に他ならない. つまり上記の 1 行目は,

```
printf("hello %s san\n", name);
```

と同じ事である.

そして UNIX においては, コマンドを起動する際にファイルのリダイレクト (redirect) という表記を用いると, それらの入出力先を変える (リダイレクトする) 事ができる.

```
./a.out > filename
```

とすると, 標準出力を *filename* にした上でプログラムが起動される. したがってプログラムが open や fopen を一切呼び出さなくても,

```
write(1, data, 10);
fprintf(stdout, "hello %s san\n", name);
printf("hello %s san\n", name);
```

などは, *filename* への書き込みと同じ効果を持つ. 要するに単純なファイル入出力しか行わない (扱うファイルが 1 つだけで, 起動時に決まるような物) プログラムであれば, open などを用いて明示的にファイルを開く必要がなくなるのである.

> は, 標準出力のみ, *filename* に送り込む. 標準エラー出力を別のファイルに送り込みたければ, 2> という表記を用いる. つまり,

```
./a.out > filename 2> filename2
```

とする. ただし多くのプログラムはプログラムの通常の出力を標準出力に書き, エラーメッセージなどは標準エラー出力に書く, という慣習を守る. そのような慣習に沿ったプログラムは, 標準出力だけを必要に応じてリダイレクトし, エラーは常に端末に表示する (つまりリダイレクトしない) という使い方をするのが便利である.

3. デバイスファイルの入出力を用いた音声入出力

Linux 環境 (正確には, Open Sound Software というインタフェースがインストールされている環境) では, 音データを入出力するための, 驚くほど単純なインタフェースが用意されている. それは, "/dev/dsp" という名前のファイルを通常のファイルと同じインタフェースで読み書きすれば, それが音の入出力になっている, という物である. 背景説明を後回しにしてまずは以下をやってみよ.

まずは音データの入力.

準備課題 2.9 以下のいろいろなコマンドで, /dev/dsp を読んでみよ. ファイルが終了する事はないので, 適当なところで, Ctrl-C で強制終了させる.

```
$ od -t u1 /dev/dsp
```

```
$ ./read_data /dev/dsp
```

./read_data は, 問題 **2.8** で作ったプログラムである.

おそらく, 127, 128, 129 付近の値がずらずらと並ぶと思われる.

コンピュータ内蔵のマイクが有効になっていれば, ここで内蔵マイクに向かって叫べば, 表示されるデータに変化が現れるだろう. 内蔵マイクではなく, マイク端子が有効になっていれば, マイク端子に外付けマイクを接続してそのマイクに向かって叫べば, 表示されるデータに変化が現れるだろう. どちらかの方法で出力が変化する事を確認してみよ. もしかするとボリュームやミュートの設定等でどちらをやっても変化しない可能性もあるが, それらの設定方法は後で説明する事にして今は先へ進む.

とりあえずうまく行ったなら, 以下のようにして/dev/dsp の中身を「そのまま」ファイルに保存すれば, 「録音器」が完成する.

```
$ cat /dev/dsp > data          # 数秒待って, 適当なところで Ctrl-C で終了させる
```

cat は, 「ファイルの中身を見る物」と思っているとこれ自体不思議に見えるかもしれないが, cat とは要するに, 「ファイルの中身をそのまま標準出力に出すプログラム」である. 中では以下のような事が起こっていると想像すれば良い (説明を簡略化するため, エラー検査などは省略している).

```
int main(int argc, char * argv) {
  int fd = open(argv[1], O_RDONLY);
  unsigned char data[1000];
  while (1) {
    int n = read(fd, data, 1000);
    if (n == −1) { perror("read"); exit(1); }
    if (n == 0) break;
    write(1, data, n);
  }
  return 0;
}
```

だから上のコマンドラインは, /dev/dsp から読んできた物をそのまま, data に保存する, という動作になる.

次は音データの出力. 想像できるように, /dev/dsp という名前のファイルに書けば良い.

準備課題 2.10 以下のコマンドで, /dev/dsp に書いてみよ.

```
$ cat data > /dev/dsp
```

ここで, data は先ほど作った物である. 先の実験でもし, マイクが有効に機能していそうであれば, これで先ほど自分が叫んだ音が, 再生されるはずである. ただしこの場合もボリューム設定などが関係してくるので, きちんと設定方法を理解するまではうまく行かなくても深追いしなくてよい.

/dev/dsp が何者かを理解する　ともかく/dev/dsp というファイルを読み書きするだけで, 音が入出力できてしまった. これをともかく納得してしまえば本実験を進めるには充分なのだが, 一方でこんな物を見せられると「ファイルというのはディスクに保存されたデータの事だと思っていたのに, 読み込むとその時の音が出てくるなんて...」という人も

いると思う (念のため, /dev/dsp を読んだときに出てくるのは, 過去に/dev/dsp に書かれた値とは全く関係がない. あくまでその時コンピュータに入力されている音である).

コンピュータに音を入出力するには, ハードウェアレベルではコンピュータに装着されているサウンドカードと CPU の間で, データのやりとりをする事になる. その詳細はサウンドカードごとに異なるが, オペレーティングシステムとしてはそれらに対して, あまりサウンドカードの機種に依存しない, かつ簡易なプログラミングインタフェース (API) を提供したい.

UNIX において, サウンドカードを扱う API として策定されている物の一つが, Open Sound System (OSS) と呼ばれる API で, /dev/dsp というファイル名を通して, 音の入出力を行えるようにする, という「決まり」もそこで仕様として策定されている.

そもそもデバイスにはサウンドカード, ハードディスクコントローラ, シリアル入出力, USB デバイスなど, いろいろな種類がある. UNIX では, それら様々なデバイスの種類毎に, それと通信するための新しい関数群 (例えば, open_sound_card, read_sound_card, etc.) を発明せずに,

(1) すべてのデバイスに「共通の」インタフェース (open, read, write, close, etc.) を設ける.
(2) open, read, write, close, etc. の操作にどう「反応するか」はデバイスごとに異なってよい (それを決めているのは, **デバイスドライバ**と呼ばれるプログラム).

という慣習を作っている. そして, それらを「普通の」ファイルの読み書きのインタフェースとも共通化してしまった.

最後の, 「デバイス」を読み書きするインタフェースと「普通のファイル」を読み書きするインタフェースを同じにする必然性はないと感じるかもしれない. しかし, 共通化する事で普通のファイルもデバイスも全く同じように読み書きできるようになる. 後に出てくる, ソケット (ネットワークとのやりとりをする API) も, UNIX においては, 実はファイルと共通の API で読み書きする事もできる.

よく考えてみると, そもそも「普通のファイルの挙動 (書いたデータが読まれる)」という物自体, ハードディスクや USB メモリなどの「2 次記憶装置」と呼ばれる—それ自体とても複雑な—「デバイス」とのやりとりを上手にやる事によってどうにか実現されている非常に高級な機能である. 普段ファイルをあまりに何気なく使っているので忘れてしまいがちだが, コンピュータを作る上で, 普通のファイルの挙動が, その他のデバイスとのやりとりより簡単に, 自然に実装できるというわけではないのである. だから, /dev/dsp が「普通のファイルと挙動が違う」と言って悩むのはあべこべで, 「いろいろなデバイスが, 同じインタフェースを持つが, だがそれぞれ異なる挙動を示す」という世界がまず先にある (それが自然).

なお, Linux 環境では OSS というインタフェースは, やや古いものになりつつあるようで, 現在は, PulseAudio というインタフェースが基本のインタフェースとなっている (2011 年現在). 従って残念ながら, 自分で Linux 環境を構築した場合に, (/dev/dsp を始めとした)OSS のデバイスファイルが, 最初からインストールされていない場合がこれからは増えるだろう. 演習用に構築されている環境でも PulseAudio がインストールされているが, それを下敷きとして OSS をエミュレートするレイヤが設定されている. そのような環境を自分で構築する方法は, 実験 Web ページに掲載する予定である. この実験では, デバイスファイルを通常のファイル入出力 API (open, read, write など) で読み書きするだけで音の入出力が行え, 従って cat など, 通常はいわゆるデータファイルの読み書きにしか使わないソフトウェアが音の入出力にそのまま使えるという学習上の便利さを重視して, OSS を用いている.

4. 音入出力の設定

音の録音や再生がうまく行っていないと思ったら, サウンドカードの設定に問題がある可能性がある. 録音, 内蔵マイク, 外付けマイクが入力として有効になっていない, 録音レベルが低すぎる, ミュートが設定されている, などの問題がある. 再生でも, 内蔵スピーカ, 外付けスピーカが有効になっていない, 再生レベル (ボリューム) が低すぎる, ミュートが設定されている, などの問題がありうる.

入力源 (マイク端子か内蔵マイクか), 出力先を選んだり, 再生録音レベルを調節するためのアプリケーションとして, 「音量調節ツール」がある. パネルにスピーカの絵をしたアイコンがあればそれである. なければ, **第 1 章**, **8.1** 節で説明した要領で追加せよ. その上で,

(1) パネル上のスピーカの絵をしたアイコンを右クリック → 音量調節ツールを開く
(2) 編集 → 設定や, ファイル → デバイスの変更, を開いて色々探る (**図 G1.2.1**).

具体的にどれを選んだら良いのかは機種によっても異なるようで, あまり一般的な事は言えない. 実験で貸し出している機種でどれを選んだら良いかは実験 HP に記述する.

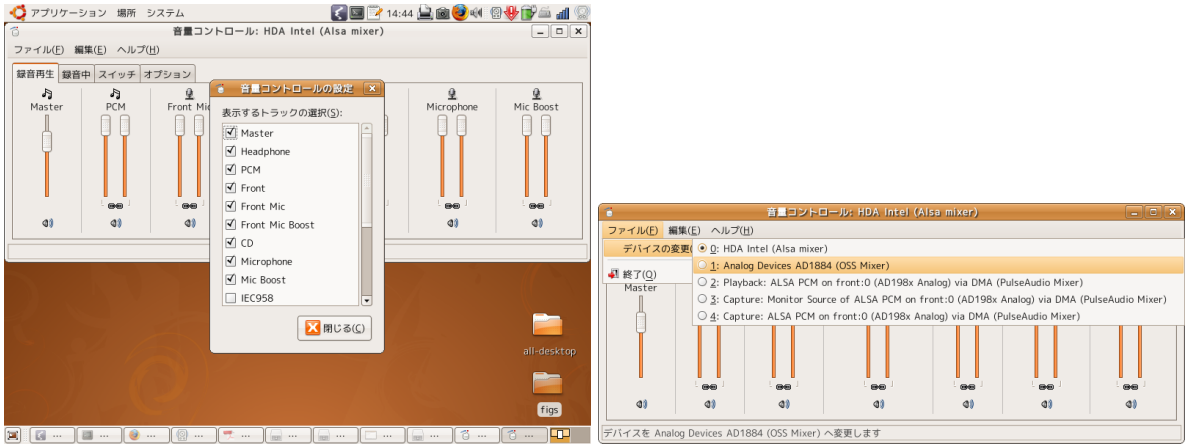

図 G1.2.1　音量調節ツール

5.　/dev/dsp のデータ形式

実際に/dev/dsp を読んだときに読まれるデータや, /dev/dsp に音を鳴らしたいと思ったときに書き込むべきデータの形式は非常に単純な物で, 音の波形 ($y = f(t)$) を, 一定間隔で標本化 (sampling) した値がずらずらと並んでいるだけの物—Linear Pulse Code Modulation (Linear PCM)—である (**図 G1.2.2**).

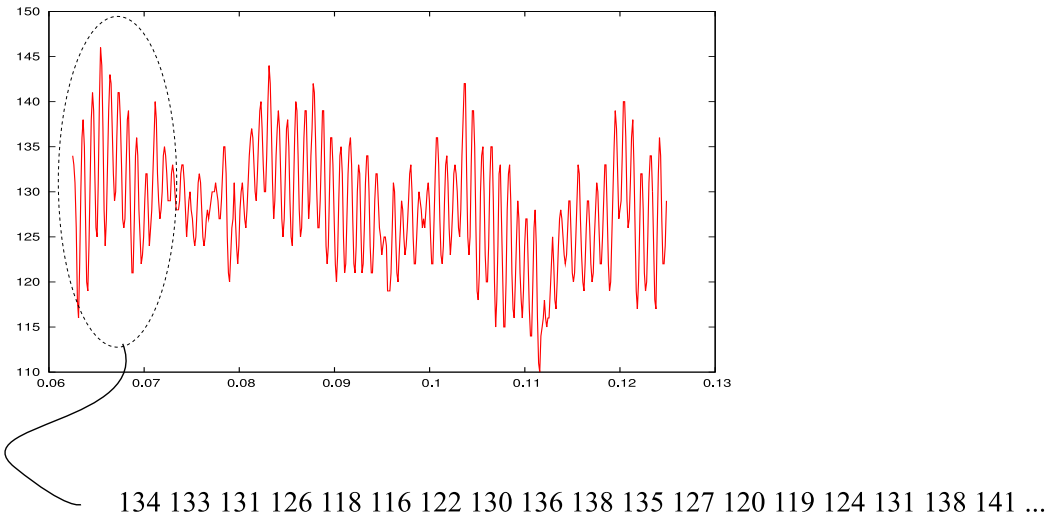

図 G1.2.2　/dev/dsp のデータ形式 (Linear PCM)

個々の標本が何 bit の値として表されるか (標本化分解能, sampling resolution), 1 秒間に幾つの標本をとるか (標本化周波数, sampling rate), 幾つのチャネルをとるか (モノラルかステレオか), は設定によって変える事ができ, 可能な組み合わせや既定の設定もサウンドカードによって異なる. 実験環境での既定値は以下のようになっている.

- 標本化分解能: 8bit (256 種類の値)
- 標本化周波数: 8kHz (8000 標本/秒)
- チャネル数: 1 (モノラル)

したがって読み出されるデータのレートは 1 バイト/標本 ×8000 標本/秒 ×1 = 8000 バイト/秒となる.

1 つの標本は 1 バイトで 256 種類の値をとり得るが, これを符号なし整数, つまり 0 から 255 までの値として読み, 128 を引くとそれが各時点での振幅となる. だから, 入力がほとんどないときは, 128 前後の値が読み出される事になる.

準備課題 2.11 cat とリダイレクトを使って/dev/dsp を 5 秒ほど読み出し, ファイルに保存せよ. そのファイルのサイズを見て, 上記のデータレートと大体あっている事を確認せよ. 逆に適当な大きさのファイルを cat コマンドで/dev/dsp に出力してみて, 出力し終えるのにかかる時間がデータレートと大体あっている事を確認せよ.

6. gnuplot

/dev/dsp から読み出されるデータを波形として可視化してみよう. gnuplot は数式で表されたグラフや実験データを, 可視化 (グラフ表示) する汎用的なツールである. 奥が深いツールだがここではこの実験で必要な最低限の機能を説明する.

gnuplot に表示させたいデータは, x 軸の値を第一カラム目に, y 軸の値を第二カラム目に書いた行を, ずらずらと並べただけの物である. 例えば,

```
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
```

のように. 値はもちろん小数点付きの数字であっても構わない.

そのようなデータがファイル (a.txt とする) として準備できたら, 以下のようにする.

```
$ gnuplot
        (バナーが表示される)
gnuplot> plot "a.txt"
gnuplot>
```

これで窓が現れ, グラフが表示される (**図 G1.2.3**). 特に設定をしないと, 各データが点として表示される. これを線で結んで表示するなど色々な設定が可能だがこの実験では不要なので省略する.

gnuplot をまず立ち上げてから, plot コマンドを入力する代わりに, 以下のような内容のファイル (a.gpl とする)

```
plot "a.txt"
pause -1
```

を用意して,

```
$ gnuplot a.gpl
```

としてもよい (pause −1 は, 改行が入力されるまで窓を表示し続けるという指示で, これを書かないと, 窓が一瞬表示されてすぐに終了してしまう).

/dev/dsp から読み出されたデータを表示するにはどうしたら良いか? 課題 **2.9** をクリアしていれば一瞬である.

本課題 2.12 課題 **2.9** において read_data を使って得た/dev/dsp の出力を, gnuplot で表示してみよ. なんとなくそ

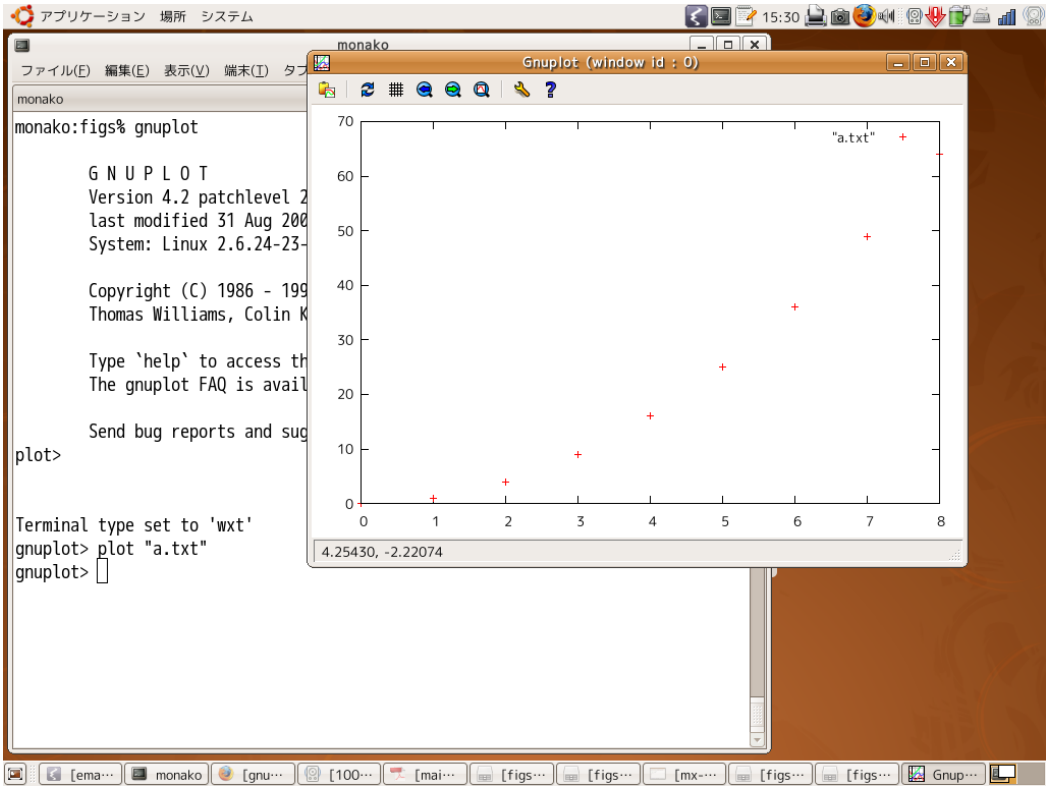

図 G1.2.3 gnuplot

れらしい波形が表示されれば成功である．次に, read_data を少し変更して, 横軸が時間 (秒単位), 縦軸は振幅 0 の時が正しく $y = 0$ として表示されるようにして (読み出されたデータから 128 を引いて) みよ．

/dev/dsp のデータ形式が Linear PCM である事を実感するもう一つの方法は, 人工的な波を出力して, 音を生成してみる事である．

本課題 2.13 いろいろな周波数の正弦波:

$$y = A \sin(2\pi f t)$$

を/dev/dsp に出力してどのような音が聞こえるかを試してみよ．A (振幅), f (周波数) をコマンドラインから受け取って変更できるようにせよ．文字列を浮動小数点に変換するには, atof という関数を使う (#include を忘れずに). あまり高い周波数を出して耳を痛めないように．$f = 440$Hz 付近, 小さめの A からスタートしてみよ．

第3日

全時間実習とする．遅れている人はこの時間で追いつく．余裕のある人は，次回を予習する，発展課題の構想を練って議論する，などの時間として使う．

第4日　デジタル信号処理

1. ね　ら　い

1.1　標本化定理と AD 変換

時間的にも振幅的にも連続値を有するアナログ信号は，時間軸上での離散化（標本化），振幅軸上での離散化（量子化）を通してデジタル信号へと変換される。有名なシャノンの標本化定理は，情報の欠落無く AD 変換（Analog-Digital 変換）を行なうための「標本化」に関する必要十分条件を教えてくれる。ここでは，故意に不適切に標本化された（エイリアスが混入した）信号を生成し，標本化定理を実験的に検証する。

1.2　標本化信号に対するフーリエ変換プログラミングとそれを用いた信号解析

時系列信号に対する最も広く知られた信号解析手法（与えられた時系列信号を基本波の重み付け和として表現する。即ち，基本波へと分解する手法）である，フーリエ変換を復習し，標本化信号に対するフーリエ変換である DFT（Discrete Fourier Transform）を C 言語で実装する。作成したプログラムを使用し，前節で用いたエイリアスが混入したサウンドデータに対してスペクトル解析を試みる。

1.3　畳み込み演算のプログラミングと FIR フィルタ　〜簡易サウンドエフェクタの試作〜

フィルタリングの基礎となるインパルス応答＆畳み込み演算について復習し，この演算を C 言語で実装する。また，インパルス応答が有限長のデジタルフィルタを FIR（Finite Impulse Response）フィルタと言うが，このフィルタを使って，音響ホールの反響音（エコー）効果をシミュレートする。自身の声を音響ホールでの歌声に変換する。

2. 解　　説

2.1　アナログ→デジタル→アナログ変換

地表を伝わる地震波，目に飛び込む可視光線，耳に飛び込む音響信号，鼻を刺激する化学物質など様々な情報は，時系列信号として存在することが多い。通常自然界に存在する時系列信号は任意時刻において値を持ち，また，観測信号の振幅は連続値を持つ。即ち，アナログ信号である。アナログ信号に対して標本化，量子化を経ることでデジタル信号は生成される。信号のデジタル化により，アナログの状態では困難であった処理が可能となった。デジタル信号処理である。

2 年後期の「信号解析基礎」を学んだ学生は，標本化信号を元のアナログ信号に戻すための標本化に関する必要十分条件を学んだはずだ。アナログ信号が存在する周波数帯域が $f \leq F_a$ であった場合，標本化周波数 F_s を $2F_a$ 以上に設定することが必要である。例えば 5kHz までの帯域に存在するアナログ信号を標本化するには，10kHz 以上の標本化周波数を選べばよい。こうすれば，標本化信号から元のアナログ信号を完全に復元できる[1]。

2.2　不適切な標本化とエイリアス

$2F_a$ 以下の標本化周波数を用いた場合，どのような現象が起こるのだろうか？例えば，オーケストラの演奏を，そのまま 8kHz で標本化すると，何が起こるのだろうか？数式的な詳細は「信号解析基礎」のノート及び参考図書に譲るが，標本化という操作は，本来の信号スペクトル $X(\omega)$ に対して，それが周波数軸上で繰り返し現れるような人工的な成分を生み出す（**図 G1.4.1** 参照。標本化周波数の代わりに標本化角周波数 ω_s が使われている）[2]。結局，標本化周波数が不適切に低い場合，**図 G1.4.2** にあるように，本来のスペクトルと，標本化によって人工的に作られたスペクトルとが重なってしまう。これがエイリアスである。オーケストラの演奏を，そのまま 8kHz で標本化すると，エイリアスが容易に起きる。実験ではこのエイリアスをダウンサンプリングを通して人工的に生成する。

2.3　アップサンプリングとダウンサンプリング

与えられたデジタル・サウンドデータを，本来のサンプリング周波数とは異なる周波数で再生することを考えてみる。例えば，48kHz で AD したデータを，8kHz で DA する（ダウンサンプリング）にはどうすれば良いのか？8kHz で AD

[1] 音楽 CD は 44.1kHz の標本化，16bit の量子化によるデジタル信号（整数値列）である。これは「地球上の全ての音は 22.05kHz 以下の帯域にしか存在していない」ことを主張しているものでは，当然，ない。この仕様は，そもそも「人間の耳が聞き取れる帯域（可聴領域）が凡そ 20kHz まで」という生理学的事実を考慮して決められた仕様である。但し，中にはそれ以上を聞き分ける方々もいる。

[2] デルタ関数 $\delta(t)$ を使った標本化信号の表現，及び，そのフーリエ変換については数式の展開を確認しておくとよいだろう。

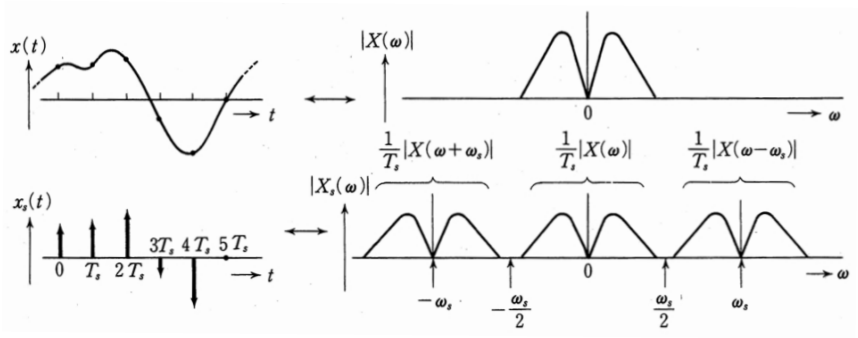

図 G1.4.1　サンプリングされた波形の周波数スペクトル（上：アナログ信号 $x(t)$，下：標本化信号 $x_s(t)$）

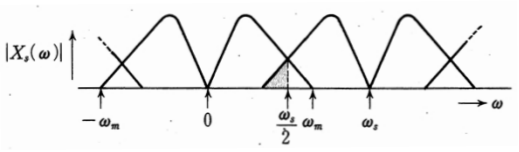

図 G1.4.2　折り返し歪み（エイリアス）の発生

したデータを 48kHz で DA する（アップサンプリング）にはどうしたらよいのか?，といった問題である。

図 G1.4.3 に示すように，48kHz で AD したデータを間引けば（6 サンプル毎に一つ取り上げる），標本化周期的には 8kHz になる。一方，8kHz で AD したデータに対して，各サンプルの間に 5 つゼロを挿入すれば，標本化周期的には 48kHz となり，かつ，8kHz でのデジタルイメージと同一になる。これらを，そのまま 8kHz（あるいは 48kHz）データとして DA すると，どのような音が聞こえるだろうか？期待した音とは異なるはずである。デジタル化（標本化）のお作法を知らないと，このような誤った信号を生成することがあるので注意が必要である。

2.4　離散フーリエ変換（Discrete Fourier Transform, DFT）

可視光線，音響信号などの時系列信号は一種類の情報だけを運ぶ訳では無い。例えば音声には，何を喋ったのか，誰が喋ったのか，どのように喋ったのか，など様々な情報が含まれている。つまり，複数の独立した情報が一つの時系列信号に混入していることになる。どの情報が，時系列信号のどの側面に対応しているのか（符号化されているのか），を知ろうとする場合，その信号を「分析」し「分解」する必要が生じる。このような場合に常套手段として用いられているのが，フーリエ変換に基づく周波数解析，である。以下，フーリエ変換を復習する。

任意の周期的なアナログ時系列信号 $x(t)$（但し周期を T とする）に対するフーリエ級数展開は以下のようになる。

$$x(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}(a_n \cos n\omega_0 t + b_n \sin n\omega_0 t) \quad (\omega_0 = \frac{2\pi}{T})$$

これを，オイラーの公式（$e^{j\theta} = \cos\theta + j\sin\theta$）を使って書き改めると，以下のように，複素フーリエ級数展開となる。

$$x(t) = \sum_{n=-\infty}^{\infty} c_n e^{jn\omega_0 t} \quad (\omega_0 = \frac{2\pi}{T}), \qquad c_n = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} x(t)e^{-jn\omega_0 t}dt$$

ここで，1) フーリエ級数展開では $x(t)$ は実関数であるが，複素フーリエ級数展開では $x(t)$ は複素関数としての扱いを

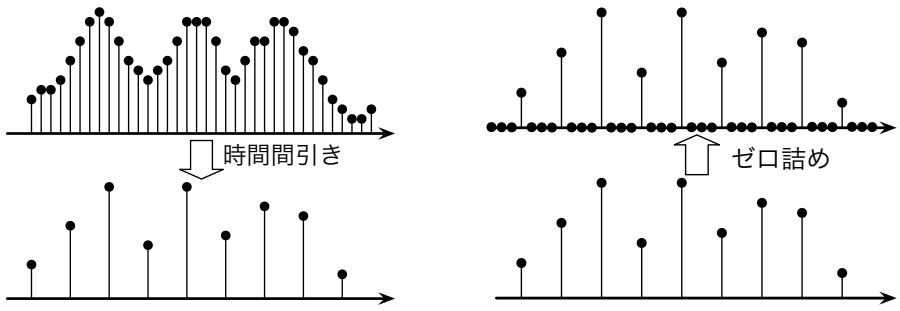

図 G1.4.3　時間間引きのみによる不適切なダウンサンプリングとゼロ詰めのみによる不適切なアップサンプリング

受ける点，及び，2) 後者では周波数が負の領域にまで拡張されている点に注意して欲しい。また，複素フーリエ級数展開は，基本波を $e^{jn\omega_0 t}$ として，任意の $x(t)$ を，重み付き基本波の足し合わせに分解する操作である。

さて，非周期信号に対しては，上記の周期信号に対する展開系を $T \to \infty$ へと拡張することで応用することが可能である。任意の複素アナログ信号 $x(t)$ に対するフーリエ逆変換／フーリエ変換は下記となる。

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)e^{j\omega t}d\omega, \qquad X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt$$

ここでも同様，角周波数 ω の基本波 $e^{j\omega t}$ に対する重みが $X(\omega)$ である。和記号（$\sum$）が積分記号（$\int$）になり，正規化係数が付与されただけである。以上はアナログ信号 $x(t)$ を，基本波とその重みに分解する数学的ツールであるが，これのデジタル（標本化）信号版が，離散フーリエ変換（DFT）である。標本化周期 T_s，サンプル数 N の標本化信号 $x_s(n)$（即ち $x_s(n) = x(nT_s)$，但し $n = 0, ..., N-1$）に対して，逆 DFT，DFT は下記のように定義される。

$$x_s(n) = \frac{1}{N}\sum_{k=0}^{N-1} X_s(k)e^{j\frac{2\pi}{N}kn}, \qquad X_s(k) = \sum_{n=0}^{N-1} x_s(n)e^{-j\frac{2\pi}{N}kn} \quad (k = 0, ..., N-1)$$

ここでも同様，基本波を $e^{j\frac{2pi}{N}kn}$ として（n が時間，k が周波数に対応），重みを付けて足し合わせているだけである。なお，分析実時間長は NT_s であるため，k は $\frac{k}{NT_s}$[Hz] の周波数に対応する。逆 DFT 及び DFT をオイラーの公式を使って三角関数表記に戻すと下記のようになる。$x_s(n)$ の実部，虚部を $x_s^r(n)$，$x_s^i(n)$ として，

$$x_s^r(n) = \frac{1}{N}\sum_{k=0}^{N-1} X_s^r(k)\cos(2\pi nk/N) - X_s^i(k)\sin(2\pi nk/N)$$
$$x_s^i(n) = \frac{1}{N}\sum_{k=0}^{N-1} X_s^r(k)\sin(2\pi nk/N) + X_s^i(k)\cos(2\pi nk/N)$$

であり，DFT は，

$$X_s^r(k) = \sum_{n=0}^{N-1} x_s^r(n)\cos(2\pi nk/N) + x_s^i(n)\sin(2\pi nk/N)$$
$$X_s^i(k) = \sum_{n=0}^{N-1} -x_s^r(n)\sin(2\pi nk/N) + x_s^i(n)\cos(2\pi nk/N)$$

となる。ここまで来ると，C 言語でプログラミングできるだろう。任意の複素系列信号 $x_s^r[], x_s^i[]$ を入力とし，それを複素系列信号 $X_s^r[], X_s^i[]$ に変換する関数を書けば，それが C 言語で実装した DFT となる。

ある区間の信号を DFT し，振幅スペクトル（$\log|X_s(k)| = \log\sqrt{X_s^r{}^2(k) + X_s^i{}^2(k)}$，$k$ は周波数に相当）や，位相スペクトル（$\angle X_s(k) = \arctan\left(\frac{X_s^i(k)}{X_s^r(k)}\right)$）を取り出すことを周波数解析と言う。言い換えれば，各周波数の振幅の値や位相の値を使って，様々な情報が表現される（振幅や位相に情報が埋め込まれる）ことが多い，ということである[3]。再掲するが $k = 0, ..., N-1$ は周波数で言えば $0, ..., \frac{N-1}{N}\frac{1}{T_s}(= \frac{N-1}{N}F_s)$ に相当する。実験で確認して欲しいが，DFT を行なうと振幅スペクトルや位相スペクトルはナイキスト周波数 $F_s/2$ を軸にした対称形になる（**図 G1.4.1** 参照）。

2.5 畳み込み演算と FIR（Finite Impulse Response）フィルタ

屋外である歌を誰かに歌ってもらう。ホールで同じ歌を同じように歌ってもらう。全く同じように歌ってもらっても，声の響き方，聞こえ方は違ってくる。歌い手の口を出た時は同じように空気が震動していたはずなのに，耳に届いている空気の震動パターンは，屋外とホールでは大きく異なってくる。目の前にいる人の声を聞く。今度は携帯を通して聞く。当然，相手は同じように喋っていても聞こえて来る声（空気の震動パターン）は直接聞くのと携帯を通した場合とでは異なってくる。このような信号の変容は，その信号にフィルタがかかることで起こる現象である。ここでは，デジタル信号処理におけるフィルタリングの基礎である，FIR フィルタについて解説する[4]。

単位インパルス信号を

$$\delta(n) = \begin{cases} 1 & (n = 0) \\ 0 & (n \neq 0) \end{cases}$$

[3] ちなみに，人間の耳は（近似的に言って）音響信号に対する，天然の「対数振幅スペクトル抽出器」として機能している。
[4] 以下では，添字の s は省いている。

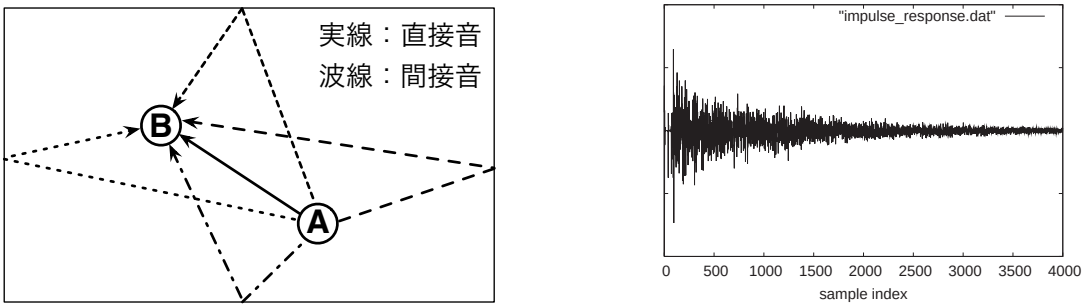

図 G1.4.4　反射波によるエコー（間接音）の発生と，音響ホールのインパルス応答

とすると，任意のデジタル（標本化）信号 $x(n)(n=0,1,2,...)$ は下記のように表現される。

$$x(n)=\sum_{k=0}^{\infty}x(k)\delta(n-k)\quad(n=0,1,2,...)$$

つまり，単位インパルスを基本波として，その重み付け和として $x(n)$ を表現しているだけである。ここで，単位インパルス $\delta(n)$ をフィルタ H に通した時に，その出力（応答）が $h(n)(n=0,1,2,...)$ となったとする。インパルスに対する出力（応答），という意味で，この $h(n)$ を単位インパルス応答と呼ぶ。フィルタ入力を $i(n)$，出力を $o(n)$ と書けば，

$$i(n)=\delta(n)\rightarrow\boxed{\text{フィルタ }H}\rightarrow o(n)=h(n)\quad(n=0,1,2,...)$$

となる。さて $x(n)$ をこのフィルタに通した場合の応答はどのようになるだろうか？時刻 k だけ移動した単位インパルス $\delta(n-k)$ に対する応答は，当然，$h(n-k)$ となる。よって，

$$i(n)=x(n)=\sum_{k=0}^{\infty}x(k)\delta(n-k)\rightarrow\boxed{\text{フィルタ }H}\rightarrow o(n)=\sum_{k=0}^{\infty}x(k)h(n-k)=\sum_{k=0}^{\infty}i(k)h(n-k)\quad(n=0,1,2,...)$$

となるのは自明だろう。これがデジタル信号における「畳み込み」演算である。畳み込み演算は対称性があるため，

$$o(n)=\sum_{k=0}^{\infty}i(k)h(n-k)=\sum_{k=0}^{\infty}h(k)i(n-k)\quad(n=0,1,2,...)$$

が成立する。結局，フィルタの単位インパルス応答（$h(n)$）が既知となれば，如何なる信号を入力したとしても，その出力はプログラミングできることになる。さて，$h(n)$ が有限項で打ち切りとなったとする。即ち，

$$h(n)=0\quad(n>n_0)$$

である。この場合のフィルタをインパルス応答が有限という意味で，Finite Impulse Response (FIR) フィルタと呼ぶ。

2.6　反射波によるエコー効果の FIR フィルタを用いたモデル化と実装

音響ホールのステージにおいて，単位インパルスを発生させることを考えてみる。例えば，手を口の前でパンとたたいてみる（**図 G1.4.4** 左中の A）。するとどうなるか。図に示すように，直接耳（図中の B）に届く経路もあるし（直接音，実線），様々な壁に反射して時間差を伴って耳に届く経路もある（間接音，破線）。当然，直接音は減衰が小さいし，間接音は減衰が大きい。また，間接音の経路は無数にある。その結果，パンというインパルス生成に対して，耳に届いた信号が**図 G1.4.4** 右となったとする。このインパルス応答 $h(n)$ が与えられれば，そのホールにおいて，自分の口で発生させた音がどのように聞こえるかは，簡単にプログラミングで確認できることになる。例えばカーネギーホールで歌うと自分の声はどのように聞こえるか，は，$h(n)$ さえあればプログラミングできる，ということになる。

この反射波によるエコー効果に対する簡単なモデルを考える。今，インパルス応答が**図 G1.4.5** のようなものであったとする。この場合，直接音によるインパルスと，数個の間接音によるインパルスのみを考える。直接音から次の反射音までの遅延を初期遅延（d_p），その後の遅延を高次遅延（d_s）と呼ぶことにする。また，直接音の振幅を $a(0)$，反射波による間接音の振幅を $a(k)(k>0)$ とする。ここで以下の関係式を仮定する。

$$a(k)=\begin{cases}1 & (k=0)\\ a_p & (k=1)\\ a_s a(k-1) & (k>1)\end{cases}$$

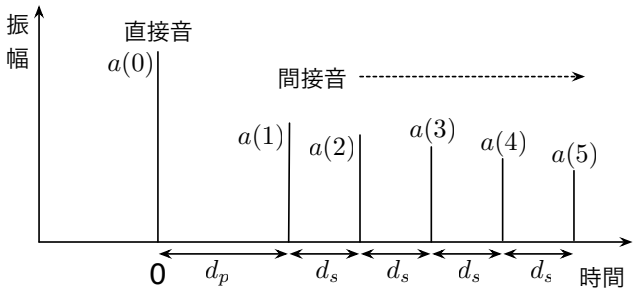

表 大小の音響ホール用のパラメータ値

パラメータ	小ホール	大ホール
a_p	0.25	0.25
a_s	0.4	0.4
d_p	50 [ms]	100 [ms]
d_s	25 [ms]	50 [ms]
K	10	10

図 G1.4.5 直接音及び間接音で構成されるインパルス応答のモデル

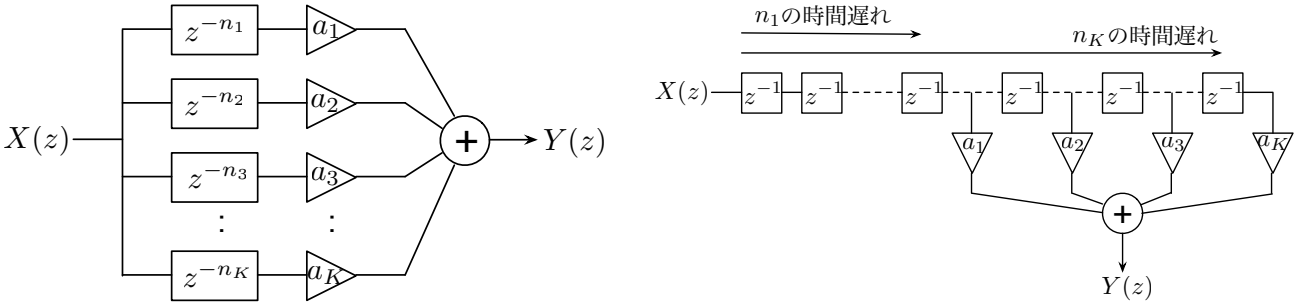

図 G1.4.6 **図 G1.4.4** に対する 2 通りのフィルタ構成

$$d(k) = \begin{cases} 0 & (k=0) \\ d_p & (k=1) \\ d_s + d(k-1) & (k>1) \end{cases}$$

a_p は初期反射音の減衰率，a_s は高次反射音の減衰率である。このようなモデルを仮定すれば，入力 $i(n)$ に対する出力 $o(n)$ は次のように表現できる。FIR フィルタの一種である。

$$o(n) = \sum_{k=0}^{K} a(k) i(n - d(k))$$

このようなモデルを仮定すれば，幾つかのパラメータ値を操作することで，様々なエコー効果を模擬することができる。例えば，表に示す値を使うことで，大小の音響ホールを模擬することができる。

なお，**図 G1.4.4** に示した反射波による音響効果は，基本的には，ある単一の音源からの音波が異なる時間遅れで（かつ，異なる係数倍されて）観測者に届くことが原理となっている。これをそのままデジタルフィルタ的に解釈すれば[5]，**図 G1.4.6** 左となる。これを**図 G1.4.6** 右のように考えれば，典型的な FIR フィルタであることが分かるだろう。

3. 実験課題と検討事項

本課題についても web ページが用意されている。各種データやサンプルプログラムがダウンロードできる。また，理解の助けとなるようなリンクを紹介しているので必要に応じて参照すると良い。なお，目安として，2 年後期に「信号解析基礎」の単位を取得した者は全課題に臨んで欲しい。それ以外の者は課題 4 までは遂行して欲しい。

本課題 4.1 同一音源に対して異なる標本化周波数で標本化したサウンドデータの聴取

標本化周波数 F_s が与えられ，これを用いて任意の音源を AD 変換する場合，エイリアスが生じないように，事前に $F_s/2$[Hz] 以上の情報を削除する（Low Pass Filtering, LPF）必要がある。数種類の音源に対して，適切な LPF を施した上で，48kHz, 16kHz, 8kHz の標本化，16bit の量子化を施して得られたデジタル・サウンドデータを実験 web に掲載している。これを聴取して，標本化周波数の違いによる聞こえ方の違いについて，音源間で比較せよ。デジタ

[5] 乗算，加算，時間遅れの 3 つの演算で信号の処理を考えれば，ということである

ル・サウンドデータ再生プログラムは web にある。

検討事項　音源によって「標本化周波数の違いによる聞こえの差」に違いがあるのは何故か，を考えよ。

本課題 4.2 不適切に標本化された（不適切にアップ／ダウンサンプリングされた）サウンドデータの聴取

図 G1.4.3 に示すように，48kHz で（正しく）標本化されたデータに対して 6 サンプル毎に一つの値を抽出して作成した，疑似 8kHz 標本化データを web に掲載している（不適切なダウンサンプリング)。48kHz データが 4kHz 以上の帯域に情報を持つ場合，当然，エイリアスが生じる。適切に 8kHz で標本化されたデータは課題 1 で聴取したが，不適切な 8kHz データについても聴取し，エイリアスの聞こえ方について音源間で比較せよ。

また，8kHz で（正しく）標本化されたデータに対してゼロを 5 つずつ挿入することで作成した，疑似 48kHz 標本化データも掲載している（不適切なアップサンプリング)。これについても聴取し，比較せよ。正しくアップサンプリングしたデータ（48kHz 標本化データであるが，情報は 4kHz 以下にのみ存在する）も掲載している[6]。

検討事項　課題 1 を踏まえ，エイリアスの目立ち方に音源間で差がある理由を考えよ。また，不適切なアップサンプリングによって作られた 48kHz データは，どのようなスペクトル形状を持つことになるのか，を考えよ。

本課題 4.3 C 言語における DFT の実装

任意の複素系列信号 $x_s^r[], x_s^i[]$ を入力とし，それを複素系列信号 $X_s^r[], X_s^i[]$ に変換する関数をプログラミングせよ。またその関数を用いて，任意の複素系列信号 $x_s^r[], x_s^i[]$ に対して，その対数振幅スペクトルを計算するコマンドを作成せよ。コマンドの出力をファイル化し，GNUPLOT を使ってグラフ化せよ。なお，以降の実験では，区間長 N が異なるスペクトルを比較するため，対数振幅スペクトルとしては $\log_{10}\left[\frac{1}{N}\{{X_s^r}^2(k)+{X_s^i}^2(k)\}\right]$ を使え。

穴埋めソースファイルは，実験 web に掲載している。

本課題 4.4 DFT によるスペクトル解析[7]

課題 1，課題 2 で聴取したサウンドデータを分析し，対数振幅スペクトル特性を求めよ。8kHz, 16kHz, 48kHz のデータに対して，区間長は $N = 256, 512, 1536$ とせよ。実時間長で言えば，どれも 32[msec] である。なお，各音源に対して，どの時点からの N サンプルを分析対象するのかについては実験 web を参照せよ。

検討事項　課題 1, 課題 2 で行なった予想と DFT による分析結果を比較し，自らの予想を検証せよ[8]。エイリアスが目立つ音源，不適切なアップサンプリングによる影響は，分析結果のどこに現れているのか考察せよ。

選択課題 4.5 実際のインパルス応答を使った音響効果シミュレーション

数種類の異なるインパルス応答を実験 web に掲載している（データフォーマットなどは web 参照)。これを用いて，各種音源を演出してみよ。なお，本課題のために実際に無響室（反響音が全く無い部屋）で収録されたサウンドデータも掲載している。これについても実験してみよ。

[6]なお「不適切なアップサンプリング」と「適切なアップサンプリング」の聞こえの差はエイリアスに因るものではない。注意するように。「不適切なダウンサンプリング」と「適切なダウンサンプリング」の聞こえの差はエイリアスに因るものである。

[7]必ず，課題 1，課題 2 の予測を行なってから実行すること!!!

[8]課題 1, 課題 2 の予想が不適切であっても，課題 4 での検証の結果，正しい考察に到達していれば，減点はしない。自らの思考の誤りを実験を通して見つけ，それを正す能力も今後要求される能力の一つである。

穴埋めソースファイルは，実験 web に掲載している。

検討事項　エコー効果は FIR フィルタであることを説明した。作成した各種エコーフィルターの対数振幅スペクトル特性を求め，GNUPLOT で図示せよ[9]。

選択課題 4.6 FIR を使った，簡易サウンドエフェクタの試作

図 G1.4.5 に従って簡易サウンドエフェクタを試作してみよ。なお，穴埋めソースファイルは，無い。

4. 実験方法と注意事項

本テキストと実験 web をよく参照しながら実験を進めるように。各種コマンドの使い方，データファイルのフォーマットなどの情報も実験 web に掲載している。なお，配布 PC の内蔵スピーカでは，音の違いがよく分からない可能性があるので，外付けスピーカを繋げると良いだろう。但し，むやみにボリュームを上げて回りの学生に迷惑をかけないように。また，ヘッドセットマイクは乱暴に扱うと，破損する（折れる）ことがあるので注意するように。

参考図書

- 「プログラミング言語 C」B. W. カーニハン，D. M. リッチー著，石田晴久訳，共立出版 (1989)
- 「信号解析入門」越川常治著，近代科学社 (1992)
- 「ディジタル・サウンド処理入門」青木直史，CQ 出版社 (2006)
- 「C 言語ではじめる音のプログラミング」青木直史，オーム社 (2008)

[9] フィルターを通した後の楽器音・音声の振幅スペクトルを求めよ，と言っているのではない。フィルターの対数振幅スペクトル特性を尋ねている。フィルターの周波数特性とインパルス応答の関係が分からない学生は「信号解析基礎」のノートを再度チェックすると良いだろう。

G2.　情報: 第2部

第5日　インターネット基礎

1. はじめに

今回からいよいよインターネット電話 (or 会議システム) を作るにあたっての, もう一つの重要な要素である, ネットワークに関する課題を始める.

今回は, ネットワークの「プログラミング」について学ぶ前の準備として, インターネットについて理解しておくべき概念を学び, 既存のコマンドを通してネットワークを使ってみる事で, それを実感する. 今回学ぶ内容は, ネットワークを用いるソフトウェアを設定したり, ネットワークがつながらない時のトラブルシュートのために必要な知識や概念でもある.

準備課題 5.1 演習用 wireless network ZENKIJIKKEN に接続し, インターネットのページが閲覧できる事を確かめよ. 例えば演習ホームページ http://g1g2g3.logos.ic.i.u-tokyo.ac.jp/にアクセスしてみよ.

2. インターネットの実体

2.1 IP

コンピュータを日常利用するに当たっては,

インターネットへ接続されている ≈ Web ページが閲覧できてメールができる

という事かもしれない. 本実験を通して「インターネットへ接続されている」という状態をもう少し精密に, 技術的に理解できるようになって欲しい.

「インターネット」という物の技術的な実体は, Internet Protocol (IP) という通信プロトコルを基礎としている. IP は, **IP アドレス**という名前を宛先として, その宛先さえ指定すれば世界中のどこへでもパケットを届けてくれるという仕組みである. そのために IP では様々な宛先への経路 (ルート) を教え合う仕組み, IP を低位層のネットワーク (Local Area Network. 有線 LAN や無線 LAN) と結びつける仕組みを規定している.

IP アドレスの例は, 133.11.238.2 のような, 0-255 までの数字が 4 つ並んだ物である. ビット数で言えば, 8 ビット $\times 4 =$ **32 ビット** である. これは, IP ver. 4 (IPv4) と呼ばれるプロトコルで用いる IP アドレスで, IP ver. 6 (IPv6) の場合は, FEDC:BA98:7654:3210:FEDC:BA98:7654:3210 のように, 16 進数 4 桁 (16 ビット)$\times 8 =$ **128 ビット** のアドレスが用いられる. 本実験で用いる IP アドレスは IP ver. 4 の物である. 当然の事ながらあるホストで, **IP を用いた通信ができるための条件その 1 は, IP アドレスが割り当てられる事**である.

UNIX では, ホストに設定されている IP アドレスは, **ifconfig** コマンドを使って調べる事ができる (Windows では, ipconfig).

準備課題 5.2 ifconfig コマンドを用いて自分のアドレスを調べてみよ.

```
$ ifconfig
```

なお, 有線, 無線含めて多数のネットワークインタフェースに関する出力がなされるが, 実験室では無線 LAN を用いているだろうから, このうち見るべきは無線 LAN の物である. それは wlan0 という名前のインタフェースになっていることだろう.

```
$ ifconfig wlan0
```

とすれば wlan0 の設定内容だけを見る事ができる.

「インターネットに接続されている」という状態は, 最も原始的な意味では「IP パケットを送受信できる」という事である. UNIX で, あるアドレスに IP パケットを送信し, その返事を受け取るというコマンドが, **ping** である. Windows でも ping コマンドを用いる.

準備課題 5.3 ifconfig コマンドを用いてお互いのマシンの IP アドレスを調べてそれを教え合い, ping コマンドを用いてお互いの疎通確認をしてみよ. 現在どのマシンも名乗っていない IP アドレスへ向けて ping コマンドでパケットを投げても, 当然返事は帰ってこない. 適当なアドレスを指定して ping コマンドを実行してみて, その時の挙動も確認してみよ.

```
$ ping 〈友達のマシンの IP アドレス〉
$ ping www.yahoo.co.jp
$ ping 〈でたらめな IP アドレス〉
```

もちろん, 自分に IP アドレスが割り当てられていないときに ping コマンドを発行しても, 成功しない. これもあえて試して確認しておくと良い.

エラーのときにどんな挙動になるかを知っておく事は, 「急がば回れ」で, 後々自分のプログラムの間違いを診断する際に重要である.

IP で世界中のマシンとパケットのやりとりが出来るために, 中心的な役割を果たしているのが, パケットの宛先 IP アドレスに応じて, そのパケットを適切な方向へ送り込む (転送する) **ルータ**と呼ばれる機材である. ルータでない機材 (ほとんどのホスト) は,

- 自分の「近隣」の宛先へは, LAN を用いてパケットを直接届け,
- それ以外のパケットはすべて決められたルータ (**default gateway**) に投げつける,

という単純な事しかしない. どの IP アドレスを「近隣」とみなすかは, 「サブネット」という範囲で設定されており, 実体としてはある IP アドレスの区間 (当然その区間には自分の IP アドレスが含まれる) である. 表記としては,

```
133.11.238.0/25
```

のように,

```
address/n
```

という表記を用いる. *address* は IP アドレス, n は (原理的には)0...31 までの整数である. これは「*address* と上位 n ビットが一致するすべてのアドレスの集合」を意味している. 例えば 133.11.238.0/25 は, 133.11.238.0, 133.11.238.1, ..., 133.11.238.127 を意味している.

例えば 133.11.238.10 という IP アドレスで, サブネットとして 133.11.238.0/25 を設定しているホストがある IP アドレスに IP パケットを送信するとき,

- そのアドレスが, 上記の範囲にあれば, そこへは定位層のネットワーク (LAN) の機能を用いて (どうにかこうにか) パケットを届ける
- そうでなければ, default gateway に, (やはり定位層のネットワークの機能を用いて), パケットを届けて, あとは適切な転送先に送ってもらう

という場合分けをする．この場合分けを, 宛先に応じてもっと細かく設定しているのがルータである．適当に書いた「どうにかこうにか」の部分はそのうち授業で学ぶ事だろう．要するにサブネット内は「どうにかこうにか」LAN の機能を用いて通信し, サブネットを越えるためにルーティングを行ってサブネット間を渡り歩くのが IP 通信で, だから Internet (Inter Network) と呼ばれる．

ここまでをまとめると, あるホストで IP 通信ができるための条件その 2 は, **そのホストが属するサブネットの情報が設定されている事**, である．そして条件その 3 は, **default gateway が正しく設定されている事**, である．ただ, これは自分と同一のサブネット内の相手としか通信しないのであればなくてもよい事になる．サブネットの情報も ifconfig コマンドで調べる事ができる．実際に表示されるのは **subnet mask (サブネットマスク)** と呼ばれる情報である．あるサブネット *address/n* に対するサブネットマスクとは, 32 bit 中上位の n ビットがすべて 1, 下位 $(32-n)$ ビットが 0 であるようなビット列を IP アドレス風に表記した物である．例えば 133.11.238.0/25 というサブネットのサブネットマスクは, 上位 25 ビットが 1 であるようなビット列を IP アドレス風に表記した, 255.255.255.128 である．要するに, 133.11.238.0/25 というサブネットは,

a & 255.255.255.128 = 133.11.238.0

となるような IP アドレス a の集合, ということである．

Default gateway は, route コマンドを用いて調べることができる．このコマンドは一般に, どの IP アドレス宛のパケットは, 次にどこへ届けるかと言う情報 (ルーティングテーブル; 経路表) を表示するもので, 通常のホストであれば, 同一サブネット内とそれ以外の欄が表示される．そして後者に対して default gateway へパケットを投げる, という情報が表示される．

$ route -n
カーネル IP 経路テーブル

受信先サイト	ゲートウェイ	ネットマスク	フラグ	Metric	Ref	使用数	インタフェース
133.11.238.0	0.0.0.0	255.255.255.128	U	2	0	0	wlan0
0.0.0.0	133.11.238.1	0.0.0.0	UG	0	0	0	wlan0

2.2 グローバル IP アドレスとプライベート IP アドレス

ひとたびホスト (PC など) に IP アドレス, default gateway, サブネットマスクが設定され, default gateway までは LAN で通信できる, という状態が確保されてしまえば, あとは世界中のどんなマシンに対しても IP パケットを届ける事が出来る．もちろんそれには default gateway が正しく設定されている (宛先 IP アドレスに応じて適切な経路 = 次のルータを選んでくれる), というのが前提であるが, その仕組みについて話すと長くなる (そのうち授業で出てくる) し, 幸い個々のホストがその設定に関与するわけではないのでここでは深入りしない．逆に世界中のホストから default gateway まで, あるサブネット行きの IP パケットが届くという状態になっていればあとはそのサブネット中の個々のホストに対する IP パケットをその gateway が LAN を使って届けてくれる．こうして個々のホストは, 世界中のホストと通信が出来ることになる．

ただしいくつか例外があり, 実験室環境もその「例外」に相当しているのでそれを一応説明しておく．

フィルタリング ：セキュリティの方針により, ルータが一部の IP パケットを転送しない, ということがある．例えば,

- 特定の IP アドレスに向けたパケットしか転送しない
- 特定の IP アドレスに向けたパケットは, 特定の IP アドレスから送られてきたパケットしか転送しない

など．後述する「ポート」を用いてさらに細かく設定されることもある．

プライベート IP アドレス ：全世界的な慣習として, ルータがそれらに向けた転送をしないことが決められているサブネットが存在する．それらのサブネットに属するアドレスを**プライベート IP アドレス**と呼ぶ．それ以外の IP アドレスは**グローバル IP アドレス**と呼ぶ．

プライベート IP アドレスとは具体的には以下である．

192.168.0.0/16	192.168.0.0	...	192.168.255.255
172.16.0.0/12	172.16.0.0	...	172.31.255.255
10.0.0.0/8	10.0.0.0	...	10.255.255.255

これらの IP アドレスに対して, 異なるサブネットからルータを経由して IP パケットが転送される, ということは慣習上行われない.

逆に言うと, 本来世界中で一意に割り当てられるはずの IP アドレスも, プライベート IP アドレスに関しては例外で, LAN が異なれば同じプライベート IP アドレスを複数のマシンが名乗っても混乱は生じない. プライベート IP アドレスの名前の由来でもある. そこでプライベート IP アドレス (サブネット) は, LAN を手軽に構築する手段として多用されている. 実験室の無線 LAN につないだ場合も, プライベート IP アドレスが割り当てられる. 家庭でプロバイダと契約して, ブロードバンドルータなどに PC をつないだ際に割り当てられるアドレスも大概プライベート IP アドレスである.

プライベート IP アドレスしか持たない PC でも, 世界中のマシンへ向けてパケットを送ることは可能である. 一方, 世界中のマシンからそのプライベート IP アドレスへ向けてパケットを送っても, それがそのマシンに届くことはない. ではなぜ, そのようなマシンでも普通にホームページが見られるのか? ホームページを見たいというリクエストが Web サーバに届くところまではよいが, 返事 (ホームページの内容) をどうやって受けとるのか?

そこには分かりにくいトリックが働いている. 自分から世界中のマシンへ向けて IP パケットを送る際に経由するルータ (通称, NAT ルータと呼ばれる. 家庭で使うブロードバンドルータも大概がこれである) が, IP パケットの送信元 IP アドレスをこっそり, PC の IP アドレスからルータの物に変更している. リクエストを受け取った Web サーバはそれがルータからの物であると思ったまま, ルータに返事を返す. そしてルータに届いた IP パケットをルータが PC に転送する.

2.3 UDP と TCP

IP での通信ができるようになれば, 原理的には, 世界中のどのホストとでもパケットのやりとりができる. しかしながら実際のアプリケーションを作るに当たっては IP だけではまだ不十分な点が多い. そのために, UDP と TCP というプロトコルが IP 上に構築されている.

IP が不十分な点の第一は, IP アドレスは個々のホストにつき一つ (複数設定する事もできるが, その場合でもせいぜい数個) しか持たせられないということである. そのため, 複数のアプリケーションが一つのホストで同時に起動されて通信しようと思うと, それらの仕分けをする必要が生ずる. つまりそれら複数のアプリケーションに異なる論理的な「宛先」を割り当ててやらないといけない. 例え話としては, 「東京都文京区本郷 7-3-1」というアドレスに, ビルの名前や学科の名前をつけて, それを実際に受け取る人を指定できる必要が有る. このために**ポート番号**という数字を用い, IP アドレスとポートの組を通信の宛先名, とできるようにしたのが **UDP (User Datagram Protocol)** である. ポート番号は 16 ビット, つまり一つの IP アドレスで, 65,536 個の UDP 通信の宛先を論理的に持つ事ができる.

IP が不十分な点の第二は, 通信の到達保証 (信頼性) がないという点である. つまり, 送信したパケットが必ず宛先に到着するという保証はないし, 到着したか否かを送信者に知らせる仕組みもない. これは様々な場面でプロトコルやネットワーク機器の設計を単純にする. 例えばネットワーク機器は高負荷時に何の制御や通知もせずに, パケットを破棄する事ができる. そもそも世界中と通信する事を目標に設計されたプロトコルだから, すべてのルータが健康に動作しているなどという前提でプロトコルを設計する事はできないので, これはもっともな事である.

一方で, すべてのアプリケーションを「送信したパケットが黙って破棄されるかもしれない」という前提で記述しなくてはいけないのでは, プログラムは恐ろしく複雑になってしまう. そこで, IP の上に信頼性のある通信を提供しているのが, **TCP (Transfer Control Protocol)** である. インターネット上のアプリケーション—Web, メール, ファイル転送など—は, 多くが TCP を用いており, インターネット技術のコア中のコアと言ってよいプロトコルである. そのためインターネットの事を代名詞的に, **TCP/IP** と呼ぶ場面も多い. ただし, 「無理な物は無理」—例えば通信相手自身が途中でいなくなってしまったり, LAN への接続が長時間切れたらエラーになる—という事は当然である. 一時的なルータの高負荷や, 短時間の切断に対する耐性を提供するのが TCP である.

2.4 DNS

最後に, 宛先の指定に IP アドレスとポート番号を用いると述べたが, IP アドレスは人間が用いる名前としては不便である. そこで通常, ホスト名は, IP アドレスではなく, **DNS 名**というシンボリックな (アルファベットを用いた) 名前で指定する. DNS 名の例は, www.yahoo.co.jp や www.cnn.com のような, web ページを閲覧しているときにも, 用いている物である. DNS 名は実際の IP 通信に先立って, IP アドレスに変換される必要があり, この変換を行う仕組みが **DNS (Domain Name System)** である. DNS は技術的には DNS 名からそれに関連する情報 (IP アドレスなど) を引き出すための巨大なデータベースである.

DNS の変換処理は, 多数の計算機 (**DNS サーバ**) に分散して実現されている. したがってこの問い合わせ自体に IP (UDP) が使われている. 個々の PC は, 手近な DNS サーバひとつを指定して, すべての問い合わせをそこ (**Primary DNS サーバ**) へ投げつける. DNS サーバは, DNS サーバ間で協調し, 自分で処理できない問い合わせを適切に転送して, どんな問い合わせに対しても結論を出す. そこであるホストで, シンボリックなホスト名を用いた通信 (≈ 一般ユーザが快適に, 「いわゆるインターネット」) ができるための条件その 4 は, primary DNS サーバが設定されている事である. ただ, 純粋な技術的な言葉使いにしたがえば, これは「IP 通信ができる」ための要件ではない.

UNIX 上で, DNS を用いて DNS 名を IP アドレスに変換するコマンドはいくつかある. nslookup, host, dig などがあるが, 本実験では host コマンドを使う. Windows には, nslookup コマンドがある.

準備課題 5.4 host コマンドで www.yahoo.co.jp や, 自分のよく使うホスト (Web サーバなど) の IP アドレスを調べよ.

```
$ host www.yahoo.co.jp
```

それで IP アドレスが分かったら, http://www.yahoo.co.jp/ の代わりにそのアドレスを用いて, ブラウザでページをアクセスしてみよ. そのまま yahoo を見続けてはいけない.

また, シンボリックなホスト名を IP アドレスに変換する仕組みは, DNS 以外にも, 設定ファイル (/etc/hosts) へ直接記述する, NIS, LDAP などの仕組みがあり, 実際の運用ではそれらを組み合わせる事もあるので, 少々ややこしい. この実験では, DNS 以外は無視してよい.

2.5 ここまでのまとめ. インターネットの要件

個々のホストに以下の情報を与える (設定を施す) と, インターネットで通信をするための要素技術が使えるようになる.

IP アドレス:
サブネット: どの宛先アドレスの範囲に, LAN を用いて直接 (gateway を経由せずに) パケットを送信するか
default gateway: サブネット外の IP アドレスへ向けたパケットを送りつける宛先. そこから先の転送を行ってくれる (はず).
DNS サーバ: シンボリックなホスト名 (DNS 名) を IP アドレスへ変換してくれるサーバ

さて, 上記の情報は手動で設定する事もできるが, ラップトップなど, 多くのパーソナルユーザ向けのマシンでは **DHCP (Dynamic Host Configuration Protocol)** というプロトコルで自動的に設定される. 今時の PC は, 何も設定をしなければ DHCP を用いてこれらの情報を自動設定するようになっているため, PC を物理的にネットワークに接続すれば気づかないうちにインターネットへつながっている事も多い. その裏ではこれらの情報が自動的に注入されているのである. ifconfig で表示されるのもこうして注入された情報に他ならない.

コンピュータの世界には (世の中すべてで?) 大した意味のない 3 文字略語があふれているが, 表 **G2.2.1** に載せた物は間違いなく重要な概念である.

表 G2.2.1 インターネット関係の最重要 2/3/4 文字略語

略語	概要	実装の (主な) 所在
IP	IP アドレスを宛先として, 世界中のネットワークに IP パケットを届ける パケットは途中で破棄されるかもしれない	ルータ
UDP	通信の宛て先に IP アドレスとポート番号を用いる. 一ホストで複数の通信が可能になる	ホスト
TCP	パケットの到達性と到達順序を保証する	ホスト
DNS	IP アドレスの代わりにホスト名 (www.yahoo.jp など) を用いた通信 を可能にする. 正体は, ホスト名 →IP アドレスの変換	DNS サーバ
DHCP	ホストをインターネットに接続するための情報を配信する	DHCP サーバ

3. nc コマンドで TCP/IP 通信を行う

前節で述べた通り, インターネットを用いたアプリケーションは, TCP もしくは UDP というプロトコルを用いて,「IP アドレスとポート番号の組」を宛先として指定しながら通信を行う.

もちろんそれは普段, インターネットを用いているあらゆるアプリケーションの中で起きている事である. 例えば Internet Explorer や Firefox などのウェブブラウザ, Windows メールや電信八号などのメールソフトなど, あらゆるソフトが内部で TCP や UDP を用いた通信を行っている. 本実験で作るアプリケーションも例外ではない. そのために UNIX でも Windows でも,「ソケット」と呼ばれるプロセス間通信のためのインタフェース (Application Programming Interface; API) が用いられる. その説明は次回に回す事とし, 今回は TCP や UDP を用いた通信を行うプリミティブなコマンドを通して, その概念を実感する事を目指す. もちろんそれらのコマンドもソケットインタフェースを用いて通信を行っている.

netcat というプログラム (nc コマンド) は,

(1) どこかの IP アドレス + ポートとの接続を確立する
(2) その接続相手から受け取ったデータを標準出力に垂れ流す
(3) 標準入力から受け取ったデータをその接続相手に垂れ流す
(4) 標準入力を読み切る (EOF に到達する) か, もしくは接続相手との接続が切れたら終了する

という機能だけを持つ単純なプログラムである. 「接続を確立する」というのは, 話し相手を決めて, その人が実際にいる事を確認する, という事で, 電話をかけて呼び出し音が鳴るまでのプロセスと思えば良い. 電話で話をするには, 「どちらかが電話をかけて, どちらかが受け取る」という風になっていないといけない. ソケット通信の場合も同様で, 電話をかける側をクライアント, 受け取る方を「サーバ」と言う慣習になっている.

サーバ: 接続を確立する相手を指定せずに,「誰か」が接続をしてくるのを待つ. ポート番号を自ら指定する事もできるし, オペレーティングシステムに割り当ててもらう事もできるが, どちらにしても一つのポート上で,「接続待ち」状態になる.

クライアント: サーバが接続待ちをしている IP アドレスとポート番号を指定して, 接続をする. この両者が揃うと, 両者の間で実際のデータの通信を始める事ができる.

nc コマンドをサーバとして用いるには, -l オプションを指定し,「待ち受け」に入るポート番号を指定する. ポート番号は, 0〜65,535 のどれかを指定するが, 1,023 以下は管理者権限がないと用いる事ができない上, 一般に下の方の番号は予約されていたり, 既存のアプリケーションが用いている可能性が有るため, ここでは用いない. したがって, 大きい数 (例えば 10,000 以上) を指定する物と覚えておく. なお, nc コマンドは, 特に指定しなければ (UDP ではなく)TCP を用いて通信する.

例:

```
$ nc -l 50000
```

nc コマンドをクライアントとして用いるには, 接続したいサーバの IP アドレスとポート番号を指定する. 例:

```
$ nc ⟨IP アドレス⟩ 50000
```

もちろん **IP アドレス**の部分には, ifconfig コマンドで調べた, サーバプロセスが走っているホストの IP アドレスを用いる. これは自分のマシンの物でも, 友達のマシンの物でも, (IP アドレスさえ分かっていれば) 海の向こうのマシンの物でも良い.

また, デバッグの際は同一のマシン内で両方のプロセスを立ち上げるのも便利である. そのような場合, 127.0.0.1 というアドレスは「自分」を意味する物として常に使えるので便利である.

それら二つのプロセス (A, B とする) が揃って起動すると, A の標準入力へ入力したデータは B へ送られ, それが B の標準出力へ出力される. その逆 (B の標準入力 $\rightarrow$ A の標準出力) も同様である.

準備課題 5.5 nc コマンドを用いて 2 つのマシンの間を接続し, 簡単な文字列を送りあってみよ.

準備課題 5.6 わざと間違えたポート番号 (サーバが接続待ちでないようなポート番号) へ接続し, その際の挙動を見ておくこと.

準備課題 5.7 同じポート番号で二つのサーバを立ち上げたらどうなるか? これも後のために一度は見ておくこと.

準備課題 5.8 man コマンドで nc の使い方を調べ (**man nc**) よ. サーバやクライアントが, 接続に成功した際にメッセージを表示してくれるようにするためのオプションが有る. それを調べて使ってみよ. また, 既定では TCP を用いるが, UDP を用いるためのオプションも調べて見よ.

nc コマンドの基本は, 標準入力 $\rightarrow$ ネットワーク, ネットワーク $\rightarrow$ 標準出力, というデータの転送を行う事であったから, 当然, 標準入出力のリダイレクトを上手に用いれば, ファイルの内容を送受信する事も可能である.

本課題 5.9 nc コマンドを用いて, 適当なファイルの内容を別のマシンへコピーしてみよ. UDP でも転送してみよ.

転送する「ファイル」として, /dev/dsp を用いれば, 一方のマシンに入力した音をもう一方のマシンへ転送する事ができる. 転送した側でそれをリダイレクトして/dev/dsp に書けば, 片方のマシンに吹き込んだ音をその場で別のマシン上で再生できてしまう.

本課題 5.10 nc コマンドを用いて, 片方のマシンに入力した音声を別のマシンへ転送し, そのマシンでその音を出力してみよ. 安定して送れるか? TCP, UDP それぞれ用いて違いがあるかどうか観察してみよ.

これで「一方通行」の音声転送が出きてしまった. 後はこれを逆方向にも行えば, 会話ができる. これでインターネット電話の最低限中の最低限の機能はできてしまった. もちろんそれは nc コマンドの機能を使っているからで, 我々はいわば nc コマンドの中身も理解して (C プログラムとして) 作るのが目標であるし, それをクリアしてなお余力があれば, 効率的な転送方法, よりよい符号化, 多者間通話などの高度な課題へ進んでいく事ができる. これらは nc をこねくり回

すだけではできない.

4. nc でアプリケーションプロトコルを理解する

nc はネットワークと標準入出力を直接結ぶアプリケーションで, 使い方を工夫すると, アプリケーションがネットワークにどのようなデータを流しているのかを調べる事ができる. それにより, 普段使っているインターネットを用いたアプリケーションが, 意外と簡単な仕組み (プロトコル) でできているという事を知るだろう. ここでは試しに, ウェブブラウザとウェブサーバの間で流れているデータを調べてみる.

例えばウェブブラウザを立ち上げて, 適当なウェブページを閲覧しているとする. 各ウェブページには URL (Uniform Resource Locator) という名前 (例: http://www.ee.t.u-tokyo.ac.jp/j/banner/life.html) がついており, 現在表示されているページの URL はウェブブラウザ上部のアドレスバーに表示される.

URL はプロトコル, ホスト名 (+ ポート番号), パス名 (+ その他の部分) からなっており, 上記の例では,

- プロトコル = http
- ホスト名 (+ ポート番号) = www.ee.t.u-tokyo.ac.jp (ポート番号は省略されており, その場合 80 となる)
- パス名 (+その他) = /j/banner/life.html

である. ウェブブラウザがこの URL を受け取ったときに行う動作は,

(1) www.ee.t.u-tokyo.ac.jp というホスト上の 80 番というポート上で, サーバプロセスが待ち受け状態にある事を仮定し, そこへ接続する. もちろんそれに先立って, DNS を用いて IP アドレスを求める.
(2) 接続したら, 「/j/banner/life.html というページをよこせ」というリクエストを送信する.
(3) 送信したら, サーバから返事を受け取り, それを表示する. 無事ページの内容が取得できる場合もあれば, そんなページはない, などのエラーが返される事もある.

ステップ 2 で, 正確にどのような文字列を送れば良いのか, ステップ 3 で, 正確にどのような文字列が返されるのかは, HTTP (Hyper Text Transfer Protocol) というプロトコルで規定されているのだが, ここではその内容その物よりも, その「調べ方」が主題である.

ステップ 1: まず, nc コマンドを用いて, 取り合えず適当なポートで待ち受けにはいるだけの, web サーバをでっち上げる.

```
$ nc -l 50000
```

ステップ 2: ウェブブラウザ (例えば Firefox) を立ち上げ, このサーバから, /j/banner/life.html というページを取得するよう, リクエストする. 例えば上記で立ち上げたサーバの IP アドレスが 192.168.1.100 だったとすると, 以下をアドレスバーに入力する.

```
http://192.168.1.100:50000/j/banner/life.html
```

当初の URL のホスト名部分 (www.ee.t.u-tokyo.ac.jp) を, 192.168.1.100:50000 に置き換えた物になっている事および, :50000 で, ポート番号が指定されている事に注意. これを省略すると, プロトコルごとに定まった既定のポート番号 (http の場合 80 番) が用いられる.

すると, ブラウザが上記で立ち上げたサーバに接続し, リクエストを送る. そのリクエストは当然ながら, nc なんちゃってサーバプロセスの標準出力として, 観察する事ができる. 詳細はブラウザによっても異なるが, 以下のような感じの文字列が表示される事だろう.

```
$ nc -l 50000
GET /j/banner/life.html HTTP/1.1
Host: 191.168.1.100:50000
User-Agent: Mozilla/5.0 (X11; U; Linux i686; ja; rv:1.9.0.6) \
    Gecko/2009020911 Ubuntu/8.04 (hardy) Firefox/3.0.6
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

```
Accept-Language: ja,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: Shift_JIS,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

ここでは詳細は重要でないが, 最初の行で, 欲しいページの名前が, GET というコマンドとともに指定されている事に注意して欲しい.

ステップ 3: ここで本来のサーバであればすぐに返事を返すが, この偽サーバは何もしないので, ブラウザは返事待ち状態になる. サーバプロセスを強制終了する, でたらめな文字列を返す, しばらく放置する, など適当な事をやって, ブラウザがどのように反応するかを見てみよ.

ステップ 4: では, まともなウェブサーバはどのような返事を返すのかを見てみよう. そのため今度は nc にクライアント (ブラウザ) の役をやらせる. まず上記の実験をもう一度行って, 今度はクライアントからの文字列をリダイレクトして, 適当なファイル名 (例: request) で保存しておく.

```
$ nc -l 50000 > request
```

request には, 上で見たような文字列が保存されるはずである (確認せよ). そして今度は本物のウェブサーバに向けて, nc コマンドを用いて接続し, そのファイルの内容を送りつけ, 結果をまた別のファイル名にリダイレクトして保存する. もちろん宛先ポート番号も本来の物 (80) を用いる.

```
$ nc -q -1 www.ee.t.u-tokyo.ac.jp 80 < request > response
```

-q -1 は, 「標準入力を読み切っても終了するな」という意味のオプションで, これをつけないと nc コマンドが request を読みきったところで即座に終了してしまう (結果, 何も表示されない). 数秒も待てば response は得られているはずなので Ctrl-C で終了する.

ステップ 5: response を開いてみれば, サーバからどのような文字列が返されるのかが分かる. 実はこのままでは, ページがない (Not Found) という結果が返されているはずである. その理由は, request の 2 行目

```
Host: 192.168.1.100:50000
```

にある. これは, request を送る宛先のホスト名が書かれているのだが, これを受け取ったサーバの了解 (「自分は www.ee.t.u-tokyo.ac.jp という名前の Web サーバである」) と食い違うために生ずる. この行を手動で,

```
Host: www.ee.t.u-tokyo.ac.jp
```

と修正した上で同じ実験を行えば, なんとなく見覚えのある (?) 文字列が返ってくるはずである.

文字列にはプロトコルのヘッダと, ページの本体がかかれているのだが, 最初の空行までがヘッダである. そこを捨てて,

```
$ firefox response
```

とでもしてみれば, 見覚えのあるページが出てくる事だろう.

5. iperf でネットワークのバンド幅を測定する

iperf コマンドは, nc コマンドと使い方が似ている. ただし目的はネットワークの性能 (バンド幅) の測定であり, サーバを立ち上げ, クライアントを立ち上げると勝手に 10 秒ほどデータを流して, ネットワークのバンド幅 (転送速度) を表示してくれる.

本課題 5.11 iperf コマンドを用いて, 友達のマシンの間でバンド幅を測定せよ. たくさんの人 (またはたくさんのプロセス間) で同時に測定するとどうなるか. その状況で, nc で音声を流してみるとどうなるか?

第6日　ソケットプログラミング(クライアント)

1. 概　　要

いよいよネットワークを用いた通信を自前で行うプログラムの作成に入る．それには UNIX でも Windows でも, 標準的に提供されている「ソケット」というプログラミングインタフェース (API) を用いる．

今回はソケットのクライアントを作るための API について説明する．前回やったように, ソケットのクライアントは, あるポート上で「待ち受け」状態になっているサーバがいるという前提で, その IP アドレスとポートに向かって接続する (電話をかける)．接続後はデータの送受信 (会話) ができる, という物である．

前述したとおり IP の上に構築されたプロトコルに UDP と TCP があり, 後者が信頼性 (到達保証) を提供する．とりあえず最初はそちらが使いやすいだろうということで, 以下の説明では TCP を用いると仮定して具体的な説明をする．

TCP を使って実際に通信するための手順は以下の通り．括弧内が実際に呼び出す関数名である．

ステップ 1: ソケットを作る (socket)
ステップ 2: 接続する (connect)
ステップ 3: データの送信 (send または write), 受信 (recv または read) を行う
ステップ 4: 通信終了後, ソケットを閉じる (close)

ファイル入出力との類推で言えば, socket は open と似ている．send, recv はそれぞれ write, read と似ている．実際, send の代わりに write, recv の代わりに read を用いる事もできる．close もファイルを閉じるときと実際に同じ API を用いる．つまり UNIX においてはソケットはファイルディスクリプタの一種なのである．余分なステップは connect という事になるが, これは別のプロセスに接続するという, ファイル入出力では必要なかった物だから, 余分なステップになるのもうなづける．

あえてもう少し実際のプログラム風に書けば以下のような手順になる．

```
unsigned char data[N];
int s = socket*();
connect*(s, "192.168.1.100", 50000);
read(s, data, N);                    # data に N バイトまでのデータを受け取る
data[0] = ...;
data[1] = ...;
  ...;
write(s, data, N);                   # data から N バイトまでのデータを送る
close(s);
```

もちろん write, read は繰り返し用いても, どのような順番で用いても良い．

なお，UDP を用いる場合，大雑把に言えば上記の connect というステップが省略され，send/recv のたびに sendto/recvfrom という API 関数を用いる． sendto/recvfrom は send/recv に加えて宛先の IP アドレス, ポートを指定する引数を持つ (つまり, TCP では通信相手を前もって一つに限定しているのに対し, UDP では送る度に自由に宛先を変えることができるということである)．

残念ながら実際のソケット API は引数などがもう少し多い上に, 名前もややこしい (上で関数名に * をつけているのは, 実際の API とは引数などが異なるという事を明示するため)．あくまでそれぞれの API に渡している物は, 上のような情報だという事を見失わないように, 表面上のややこしさはある程度受け入れてもらうしかないのだが, なぜこんな事になっているのかという事情説明を少ししておく．「ソケット」はプロセス間で通信を行うための汎用的なインタフェースとして設計されている．IP 通信だけを対象としているのではない．例えば同一ホスト内の複数プロセス間で通信を行

うための, **UNIX ドメインソケット**という物がある. IP 通信にしても IPv4 と IPv6 がある. ソケット API は, それらすべてをほぼ同一のインタフェースで使えるように設計されている.

- このため一々, ソケット API の引数に「私はどの体系 (IPv4, IPv6, UNIX ドメイン etc.) で通信がしたいです」という事を明示的に指定する必要がある ⇒ 余分な引数が増える.
- TCP, UDP というような, IP にのみ通用する固有名詞も API の表面にあからさまに現れる事はない. だから, TCP を使って通信する時に, socket(TCP) とでも書ければまだいい物を, socket(. . . , SOCK_STREAM) などという, 間接的な物の言い方になる. ⇒ 引数の意味が分かりにくい.
- 通信手段が異なればアドレスの表現も違うので, ソケットの API には, IP アドレスを表すようなあからさまな引数は直接現れない. 例えば IP 通信では, 通信の宛先は, IP アドレスとポートで指定するが, UNIX ドメイン通信では, ファイルのパス名 (“/tmp/foo” などの名前) で指定する. だから connect の引数を, connect(”192.168.1.100”, 50000) とするわけには行かない. そこですべての通信手段のアドレスを包含したようなデータ型 (sockaddr) が, API の表面には現れ, それに無理やり, IP アドレスを渡すような仕組みが用いられる. ⇒ 引数の渡し方が分かりにくく, C 言語に習熟していない人には難解に感じられる.
- 同じ理由で, man ページで socket の API を見ても, IP 通信をするときの固有のやり方までは書いていない. 例えば man connect を見ても, IP アドレス 192.168.1.100 のポート 50000 につなぐ方法が書いていない.

以上の心の準備を元に, 実際の API を解説する.

2. API 説 明

2.1 #include ファイル

IP 通信をするにあたって必要なヘッダファイルがある. 悪いことにこれらは以下で説明する API 関数のマニュアルページには出ていない. 理由は上記で述べたとおりで, ソケットはあくまで「汎用的な」プロセス間通信であるため, 個々の通信プロトコルに依存した情報は載せ切れないため (実際本当に載せきれないかどうかは怪しい. ドキュメントの保守性とある種の美的感覚により載せていない, というのが本当のところか) である.

結論を言うと, 以下の API の man ページに指定されているヘッダファイル**に加え**, IP では以下を#include しておけばよいだろう.

```
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
```

なおこれらの情報源は,

```
$ man 7 ip
$ man 7 tcp
$ man 7 udp
```

である. その他適宜, TCP/IP プログラミングに関する書籍を参照するとよい.

2.2 socket

ソケットを作る.

```
int s = socket(PF_INET, SOCK_STREAM, 0);
```

ファイルの open に相当する物だと思えばよい. 返り値はファイルディスクリプタである (しつこくここでも, man socket で, 成功の確認方法と#include すべきヘッダファイルを調べよ, と言っておく). 以降で使う関数に, ここで返された値を渡す. これも open と, read/write/close との関係に似ている. 引数の,

- PF_INET は, IPv4 という通信体系 (ドメイン) を用いる事を指定しているちなみに IPv6 は PF_INET6, UNIX ドメインは, PF_UNIX.

- SOCK_STREAM は, TCP を用いた通信を指定している. ちなみに UDP は, SOCK_DGRAM.

イメージとしては, 通信相手との間に接続を確立するのは, 両者の間に糸電話を引っ張って結ぶような物だが, ソケットはその端っこ, つまり糸電話の紙コップ部分である.

2.3 connect

指定した IP アドレスとポート番号に接続する.

connect を呼び出すために, IP アドレスとポート番号をどうにかして渡す必要がある. connect(s, "192.168.1.100", 50000) とでも書ければ簡単なのだが, 前述した通り, IP 以外の事も考えて API が設計されているため, 渡し方が回りくどい.

以下は, IP アドレス 192.168.1.100 のポート 50000 に接続するための基本フォームである.

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;                          # これは IPv4 のアドレスです
addr.sin_addr.s_addr = inet_addr("192.168.1.100");      # IP アドレスは... です
addr.sin_port = htons(50000);                       # ポートは... です
int ret = connect(s, (struct sockaddr *)&addr, sizeof(addr));      # 遂に connect
```

やっている事は, 最終的に connect に addr という構造体のアドレス (&addr) を, そのサイズ (sizeof(addr)) とともに渡している. addr のデータ型は, sockaddr_in という物で, これは IPv4 アドレスを表している. それに先立つ 3 行で, その構造体の要素を埋めている.

- sin_family という要素には, このアドレスがどのような通信体系のアドレスであるかを格納する. addr.sin_family = AF_INET; はそれが, IPv4 のアドレスである事を示している. これは, connect が受け取った引数を見て, 確かにこの人は IP 通信のためのアドレスを渡してきている, という事を理解するのに用いられる.
- sin_addr.s_addr という要素に宛先 IP アドレスを格納する. そのために, "192.168.1.100" のような, 文字列による IP アドレスの表記を, 32 bit の形式に変換する関数 inet_addr を呼んでいる.
- sin_port という要素には, ポート番号を格納する. addr.sin_port = 50000 とそのまま代入すれば良さそうな物だが, 細かい事情でそうは行かず, htons という関数を呼んで, 変換してやる必要がある. 事情は後に説明する.

しつこく言うが上のコードを使う場合に, connect が成功した事を確認しないで先へ進む事はくれぐれもないように. 非常に間違いをしやすいところである. また上記を使うには色々#include が必要なのでそれもお忘れなく.

注: inet_addr vs. inet_aton マニュアルによると inet_addr は使わずに, inet_aton をつかえとある. だから,

```
addr.sin_addr.s_addr = inet_addr("192.168.1.100");
```

は,

```
inet_aton("192.168.1.100", &addr.sin_addr);
```

と書く事が推奨されている. 理由は inet_addr にはエラーを検査する方法がないから. inet_aton は返り値でそれが区別でき, IP アドレスとして不正な文字列を渡すと, 0 が返る. ここでは説明のため, 見た目のわかりやすい inet_addr を使って説明した. さんざんエラー検査をしておけといっているので, もちろん本当は inet_aton が推奨である.

2.4 send/recv (または write/read)

無事接続が成功したらデータの送受信ができる. send ≈ write, recv ≈ read と思ってよく, 実際後者を使っても良い. 両者の違いは, send/recv は, write/read よりもひとつ引数が多く, その引数でいくつかオプションを指定する事ができる事である. が, この実験においてはさしあたり使う必要はない (ここで send/recv に言及するのは主に, そちらの方が Windows 環境などを含めて, 「普通」とみなされているから, というだけの理由).

```
n = send(s, data, N, 0);
または
n = write(s, data, N);
```

```
n = recv(s, data, N, 0);
または
n = read(s, data, N);
```

動作については read, write と同じだから説明するまでもないだろう.

2.5 close/shutdown

```
close(s);
```

open したファイルを閉じるのと全く同じ API を用いる. これ以降は, このソケットを用いる事はできない.

また, read/recv は, 相手が close を呼び出し, かつ相手がそれ以前に送ったデータをすべて受け取った後, 0 を返す (ファイルを最後まで読み終わった後と同じ挙動をする). そこでこの場合も, ソケットから「EOF を読んだ」「EOF を受け取った」などという表現をする事がある. 実際にはファイルを読んでいるわけではないのだが, 慣習としてこのような言い方をする.

close をすると, 以降データの送信 (write)・受信 (read) ともにできなくなるが, 「自分はもうデータを送り終わった」という事を相手に教えつつ, まだやってくるデータを受けとる事だけはしたい, という状況がよくある. この様な時, 自分はもうデータを送り終わった事を通知するために close を呼んでしまうと, もうデータを受け取る事ができず, こまる事になる. そのような時に用いるのが shutdown という API である.

```
shutdown(s, SHUT_WR);
```

を呼ぶと, 相手は (こちらが send したすべてのデータを読み終わった後)EOF を受け取る. しかし, こちらは依然として read は可能である.

本課題 6.1 以下のような動作をする C プログラム client_recv.c をソケット API を用いて作れ.

- コマンドラインで指定した IP アドレスとポートに接続する
- 接続したサーバがデータを送ってくるという前提で, データを EOF に到達するまで読んで読んだデータを標準出力に書く

できたら, nc コマンドでサーバを立て, client_recv でそれにつないでデータを送受信してみよ. リダイレクションを利用して, nc コマンドに大きめのファイルを送らせ, 受け取った方もファイルにそれを保存せよ. 両者は全く同じ内容となるはずである. その事を確かめよ. それには, nc コマンドを用いて受け取ったデータを元のマシンに送り返して, diff コマンドを用いて比較するとよいかもしれない. また, ファイルサイズが一致している事, md5sum というコマンドで表示されるハッシュ値が一致している事を見る事もして見ると良い.

```
$ md5sum filename
1a24874fc4ea61095d0cf16da5ee8516 filename
```

本課題 6.2 数秒の音データを返して接続を切るサーバをこちらで用意する. そこに接続して, 出力を/dev/dsp にリダイレクトする事で, 音をながしてみよ.

```
$ ./client_recv 133.11.238.11 50000 > /dev/dsp
```

データを受けとるだけでなく, 送ってから受け取るプログラムも書いてみよう.

本課題 6.3 以下のような動作をする C プログラム client_send_recv.c をソケット API を用いて作れ.

- コマンドラインで指定した IP アドレスとポートに接続する
- 標準入力が EOF を返すまで, 標準入力からデータを読んではそれを接続先に送る事を繰り返す
- データを送り終わったら shutdown(s, SHUT_WR) で, 送信データの終わりを接続先に通知する
- その後, 接続先からデータを EOF まで受け取る

要するにデータを送るだけ送って, あとは受け取るというクライアントである.

受け取ったデータをそのまま送り返すだけのサーバをこちらで用意する. 本プログラムをそこに接続して送ったのと同じデータが返ってくる事を確かめよ.

```
$ ./client_send_recv 133.11.238.11 50001 < orig_file > output_file
```

これで, orig_file と output_file は同一のファイルとなるはずである. diff コマンドで比較してみよ.

```
$ diff orig_file output_file
```

課題 **6.1-6.3** と似た事を, UDP を用いてやってみるのも後々電話を作る際の選択肢を広げるために有用である. 余力があればやってみよ.

選択課題 6.4 以下のような動作をする C プログラム client_recv_udp.c をソケット API を用いて作れ.

- UDP ソケットを用い, コマンドラインで指定した IP アドレスとポートに, 1 バイトのデータを送る (中身はなんでもよい)
- その後, そのソケットにデータが送られてくる. それを受け取り, 受け取ったデータを標準出力に書く. ただし,
  - 1 回の recvfrom の呼び出しで, 1000 バイト受け取ろうとする事
  - 無事 1000 バイト受け取ってそれらがすべて 1 であった場合, それはデータの終わりを示す (End of Data; EOD と呼ぶ) ものとし, これ以上データを読まない. EOD も標準出力には書かない.

選択課題 6.5 課題 **6.2** の UDP 版サーバもこちらで用意する. そこに接続して, 出力を/dev/dsp にリダイレクトする事で, 音をながしてみよ.

```
$ ./client_recv_udp 133.11.238.11 50000 > /dev/dsp
```

同じポート番号 50000 を用いているが, TCP のポートと UDP のポートは (番号が同じでも) 違うものである. 課題 **6.2** とこの課題とでは異なるサーバと通信している.

同様に, データを送ってから受け取るプログラムも UDP 版を書いてみよう.

選択課題 6.6 以下のような動作をする C プログラム client_send_recv_udp.c をソケット API を用いて作れ.

- 標準入力から読み込んだデータを, コマンドラインで指定した IP アドレスとポートに, UDP ソケットを用いて送る.
- 一度に送るのは 1000 バイトまでとする. データの送信は最大で 50 回までとする.
- データを送り終わったら, EOD(1 が 1000 バイト連続したもの) を送る.
- その後そのソケットからデータを課題 **6.4** と同じ方法・約束にしたがって受け取り, 標準出力に出す.

受け取ったデータをそのまま送り返すだけの UDP サーバをこちらで用意する. 本プログラムをそこに接続して送ったのと同じデータが返ってくるかどうかを確かめよ.

```
$ ./client_send_recv_udp 133.11.238.11 50001 < orig_file > output_file
```

orig_file と output_file は, 多くの場合同一のファイルとなるであろう (diff コマンドで比較してみよ). しかし, UDP はデータの到達を保証せず, かつ複数回に渡って送られたデータの順番も保存しないため, 様々な事が起こりうる.

- クライアントからサーバへ送った EOD がサーバへ届かない
- サーバからクライアントへ送った EOD がクライアントへ届かない
- EOD 以外のデータが届かない
- EOD 以外のデータが EOD に追い抜かされる

そのような場合に備えて対処をすることはここでは要求しないが, それらが起きたときにプログラムがどのような挙動になるかを考え, 可能な対処方法を考えてみよ

トピック: なぜ connect その他のインタフェースはこんなに汚い? 若干本題とそれるが, C 言語の汚いところとよりストレスなく付き合えるようになるための雑学.

connect の引数の渡し方はどう見てもきれいな物ではない.

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = inet_addr("192.168.1.100");
addr.sin_port = htons(50000);
int ret = connect(s, (struct sockaddr *)&addr, sizeof(addr));
```

大元の理由は何度も述べている通り, ソケットが IP 以外の通信にも使えるように (アドレスの表現形式などが異なっていても使えるように) 設計されているからである. なぜそれを目標に設計するとかくも汚い事になってしまうのかを説明しておく.

C 言語をはじめとする多くの言語では, 関数の呼び出し形式 (引数の数やその型) は基本的には関数ごとに一意でなくてはならないという制限がある. 例えばある時は double 型の引数を 3 つ, ある時は文字列型の引数を 7 つ受けとる, などという関数 (可変引数関数) を書くという事は, 基本的にはない (厳密には嘘. printf などがまさにその例外になっているのだが, そのような関数を書くのは面倒な上, どんなデータ型に対しても使えるわけではない. ここでこれ以上の深入りはしない).

そこで, connect という一つの関数に, どうしたらある場合は IPv4 アドレスを, ある場合には IPv6 アドレスを, またある場合には UNIX ドメインソケットのアドレスを渡すのか, という事が問題となる. そのために C 言語で常套手段として使われるのが, **構造体 (struct) に必要なデータを格納し, そのポインタを渡す, という手法**である.

復習になるが, 必要な情報をすべて一つの変数にまとめるために構造体がある. 例えば, 3 次元空間内の点を表す構造体として,

```
struct point { double x; double y; double z; };
```

という, 3 つの double を一つにまとめた構造体を定義しておけば,

```
struct point p;
```

で, p は, p.x, p.y, p.z という 3 つの double の情報を保持できる変数となる. 関数に渡すときも, p 一つを渡せば, 3 つの double を渡したのと同じ効果がある. connect の API では, 通信体系ごとにそのアドレスを表す構造体が定義されている. IPv4 であれば sockaddr_in, IPv6 であれば sockaddr_in6, UNIX ドメイン通信であれば, sockaddr_un のように.

あとはこのように定義されたアドレス構造体の変数を connect に渡せば良さそう, という事になる.

```
struct sockaddr_in ipv4_addr;
struct sockaddr_in6 ipv6_addr;
struct sockaddr_un unix_addr;
...
connect*(s1, ipv4_addr, ...);
connect*(s2, ipv6_addr, ...);
connect*(s3, unix_addr, ...);
```

のように. しかし残念ながら一つの関数 (connect) の一つの引数 (第 2 引数) が, ある時は sockaddr_in 構造体, ある時は sockaddr_in6 構造体, またある時は sockaddr_un 構造体を受けとるなどという器用な事は (C 言語では) できない.

しかしそれが, 構造体ではなく, **構造体へのポインタ**ならば許される. だから, ある時は sockaddr_in 構造体へのポインタ, ある時は sockaddr_in6 構造体へのポインタ, またある時は sockaddr_un 構造体へのポインタを受けとる引数, という物は作る事ができる. それが connect の 2 番目の引数である. つまり,

```
struct sockaddr_in ipv4_addr;
struct sockaddr_in6 ipv6_addr;
struct sockaddr_un unix_addr;
...
connect*(s1, &ipv4_addr, ...);
connect*(s2, &ipv6_addr, ...);
connect*(s3, &unix_addr, ...);
```

は機能する. 形式上, その引数の型は, sockaddr 構造体へのポインタ, という事になっているため, 渡す際にキャストをする

```
struct sockaddr_in ipv4_addr;
struct sockaddr_in6 ipv6_addr;
struct sockaddr_un unix_addr;
...
connect*(s1, (struct sockaddr *)&ipv4_addr, ...);
connect*(s2, (struct sockaddr *)&ipv6_addr, ...);
connect*(s3, (struct sockaddr *)&unix_addr, ...);
```

事で, 型を connect が想定している物と (形式的に) 一致させる. また, このような事をするときは, 受け取った方でどの種類のデータを実は受け取ったのか, と言う事が分かるように, 構造体の決まった位置に, 種類を表すタグをつけておくのが普通である. それが, sin_family という要素である.

なぜ構造体その物だとダメなのに, ポインタなら良いのか? つまらない答えはそれが決まりだから, という物だが, なぜそのような決まりになっているのかには, C 言語処理系の仕組みをある程度知ると納得できる理由がある.

それは, ポインタとは所詮, アドレスの事に過ぎないので, 実は何という構造体へのポインタであろうとそのサイズが同じだからである. 例えば 32 bit のマシンではすべてのポインタは 32 bit の整数一個に過ぎない. したがって, sockaddr_in 構造体へのポインタも, sockaddr_in6 構造体へのポインタも, sockaddr_un 構造体へのポインタも, すべて同じ仕組みで渡される. だからある関数へ引数を渡すのに, ある時は sockaddr_in 構造体へのポインタ, ある時は sockaddr_in6 構造体へのポインタ, またある時は sockaddr_un 構造体へのポインタが渡されたとしても, 各引数が渡される場所がずれるような事はなく, 受け取る方に難しさは発生しない.

一方これが, 構造体その物を渡す (ポインタを渡すのと何が違うのか分からないかもしれないが, ここでは構造体の中身をすべて渡す事, と思ってくれれば良い) となると, 構造体は中身が違えばサイズも異なるため, 一つの引数のサイズが場合によって違う, という事になる. そのような引数を渡されて, 受け取る側がそれを常に正しく受け取る事は, 逆立ちしてもできないとは言わないが, ややこしい (かつ遅い) 処理が必要になる. 例えば引数を単にメモリ上に隙間なく並べて渡すような方式では, 他の引数の格納場所などが狂ってきて困った事になる. そこでこれらを間違いなく渡そうと思うと, どの引数がどの場所にあるかなどの情報も, 合わせて渡すなどの処理が, 関数呼び出しの度に必要になる. 「どの引数がどの場所にあるかなどの情報を渡す」というのは, どの引数がどのアドレスにあるか, という事だから, ポインタを渡すというのとほとんど同じ事である. そんな事をこっそりやるくらいだったら, 必要に応じてプログラムを書く方にやらせて, 普段から遅くなるような事はしない, というのが C 言語の基本スタンスである.

以上の話から, connect を使うための雛形が,

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = ...;
addr.sin_port = ...;
int ret = connect(s, (struct sockaddr *)&addr, sizeof(addr));
```

のようになる事までは納得できたのではないかと思う. connect が第 3 引数として sizeof(addr)—第 2 引数で渡された構造体の大きさ— を受け取っているのは, 絶対必要という分けではないかもしれないが, 例えばアドレスの体系によって

は必要なサイズが可変という事もありうるため, この様になっている物と納得しよう (IPv4 の場合, IP アドレス 32 bit, ポート番号 16 bit と決まっており, サイズは自ずと分かる). また, 受け取った側で第 2 引数をコピーする, などの処理を簡単に行うためにも, 渡されていた方が便利であろう.

sin_addr.s_addr については, “192.168.1.100” というアドレスを 13 文字の文字列として表現するのではなく, 8 bit ×4 の 32 bit で表すという方法を採用しているので, inet_addr という関数でその変換を行っている. sin_port については, なぜ整数をそのまま代入するだけではダメで, htons などという関数を呼んでいるのか? htons は, **h**ost-**to**-**n**etwork-**s**hort の略で, short (16 bit) 整数の, 1 バイト目と 2 バイト目の並び方 (どちらが MSB 側でどちらが LSB 側か) について, それを「ネットワークバイト順序」なる物にする関数である.

一般に CPU では, 8 bit を一つの整数とみなした演算, 16 bit を一つの整数とみなした演算, 32 bit を一つの整数とみなした演算, ... などが命令として提供されている. それに対応して, メモリのある番地 (X) から 8 bit を読み込む命令, メモリの連続した二つの番地 (X, X+1) から 16 bit を読み込む命令, メモリの連続した四つの番地 (X, X+1, X+2, X+3) から 32 bit を読み込む命令, などが提供されている. ある CPU の「バイト順序」というのは, 例えば X, X+1 の二つの番地から 16 bit を読み込む命令を発行した時に, どちらのアドレス (X or X+1) から出てきた 8 bit が, 読み込まれた 16 bit の MSB (Most Significant Byte; 16 bit 中の上位 16 bit) になるか, という事を規定している物である. これは CPU によって異なっており, Intel x86 系は Little Endian といって, X+1 番地の方が MSB になる. 他の多くのマシンでは, Big Endian といって, X 番地の方が MSB になる. ネットワークバイト順序も Big Endian である. だから, htons の正体は, Little Endian の CPU 上では 16 bit の MSB と LSB を入れ替える (Big Endian ならなにもしない), という事である.

通信を実現するパケットの中には, 送信元アドレス, ポートや受信宛て先アドレス, ポートなどが含まれている. その表現 (ここで問題となるのはポート番号) がホストごとに異なっているのは困るから, 通信をする際には全世界のマシンで共通の順序を決めておきたい. それがネットワークバイト順序である. もちろんそんな物は connect 関数の中でやってあげて, 何も渡す方がそれに統一してやらなくてもいい, という気もするが, sockaddr_in という構造体の表現は, (bit 列のレベルで)CPU のバイト順序によらない物としたかった, という事ではないかと思われる.

第7日

全時間実習とする．遅れている人はこの時間で追いつく．余裕のある人は，次回を予習する，発展課題の構想を練って議論する，などの時間として使う．

第8日　ソケットプログラミング (サーバ)

1. 概　　要

今回はソケットのサーバ側の API を説明する．これで, クライアント・サーバとも自前のプログラムで通信ができる事になる．ソケットのサーバは, あるポート上で「待ち受け」状態に入り, クライアントからの接続 (connect) を受け付ける．「待ち受け」関数から返ると, その時点でクライアントとの接続が確立しており, データの送受信をする事ができる．接続が確立するまでの手順がクライアントに比べるとさらにややこしいが, 一旦接続が確立した後のやり方はクライアントとサーバで全く同じである．

ステップ 1: ソケットを作る (socket)
ステップ 2: どのポートで待ち受けるか決める (bind)
ステップ 3: 待ち受け可能宣言 (listen)
ステップ 4: クライアントが connect するまで待つ (accept)
ステップ 5: データの送信 (send または write), 受信 (recv または read) を行う
ステップ 6: 通信終了後, ソケットを閉じる (close)

もう少し実際のプログラム風に書けば以下のような手順になる．

```
unsigned char data[N];
int ss = socket*();
bind*(ss, 50000);                    # 待ち受け番号は 50000 に設定
listen*(ss);                  # 待ち受け可能宣言
s = accept*(ss);                    # 接続が来るまで待機
close(ss);                  # これ以上接続を受け付けない場合
read(s, data, N);
data[0] = ...;
data[1] = ...;
  ...;
write(s, data, N);
close(s);
```

もちろん write, read は繰り返し用いても, どのような順番で用いても良い．close(ss) は, これ以上接続を受け付けない—つまりこれ以降 accept を呼ぶつもりがない—位置で呼び出す．

2. API 説 明

bind, listen, accept という 3 つのステップ以外はクライアントと共通である．以下では, ss が socket から返された値を格納しているとする (ss = socket(...))．

2.1 bind

bind は, 「ソケットに, ポート番号などの名前を割り当てる」システムコールである．本実験の文脈に合った形で言うと, 「将来どのポートで待ち受けるかを指定する」システムコールである．ここでも, ソケットが汎用的な (IP 通信に限らない) インタフェースとして設計されているため, その渡しかたが回りくどい (bind(ss, 50000) では済まない)．

以下が基本フォームである．

```
struct sockaddr_in addr;              # 最終的に bind に渡すアドレス情報
addr.sin_family = AF_INET;                # このアドレスは IPv4 アドレスです
```

```
addr.sin_port = htons(50000);                    # ポート...で待ち受けしたいです
addr.sin_addr.s_addr = INADDR_ANY;               # どの IP アドレスでも待ち受けしたいです
bind(ss, (struct sockaddr *)&addr, sizeof(addr));
```

全体としてやっている事は, 自分がどのポートと, どの IP アドレスで待ち受けをするつもりなのかを, addr という構造体変数 (のアドレス) を渡す事で, bind システムコールに教える事である. この渡し方自身は connect と似ているので今度は長い説明の必要はないだろう. それに先立ち addr という構造体の要素を埋めているのがその上の 3 行である.

「どの IP アドレスで待ち受けをするつもりなのか」について, そんなもの自分の IP アドレスに決まっているだろうという疑問がわくだろう. そして実際, 上記ではここを INADDR_ANY と指定して, 「どこでもいい (自分の持つすべてのアドレス)」と言っている. 複数の IP アドレスを持っていてそのうちの一部の IP アドレスに対する接続のみ受け付ける, という場合にはここにその IP アドレスを指定する事になる.

bind でよく発生するエラー: "Address already in use" (しつこい!) bind に限らずすべてのシステムコールの成功確認をする事. そして, bind を用いていると非常によく目にするエラーがこれである.

これは, 「要求したポート番号がすでに使われている」という事である. 自分の車のナンバープレートを, 横浜 500 あ 11-11 にしようと思ったら, すでに登録済みだった, というのと同じである. そして, 自分で適当なポート番号を決めてプログラムを書いていると, そのプログラムを間違えて立ち上げっぱなしでもう一個立ち上げようとしたときなどにおきがちである.

基本的には, そのソケットを close すればそのポート番号は再利用可能になる. そしてそのソケットを使っているプログラムが終了すれば自動的にソケットは close されるので, プログラム終了によってそのポート番号は再利用可能になると考えてよい. ただし, close してから実際に再利用できるようになるまでに少し時間がかかる. それは, 一度そのポートを使ったからには, そのポートをめがけて外からパケットが飛んでくる可能性があるからである.

2.2 listen

listen は, 「接続受付開始宣言」である. これをサーバが行った時から, クライアントの接続要求 (connect システムコール) は成功するようになる. また, その後サーバは実際に接続要求がくるのを待つ (accept システムコールを発行する) 事ができるようになる. 基本フォームは以下の通り.

```
listen(ss, 10);
```

第 2 引数 (ここではいい加減に 10 としてある) は, 複数の connect 要求がほぼ同時に殺到した時に, どのくらいの要求を落とさずに処理するか, という事を制御するパラメータである. この値を越える connect 要求がほぼ同時に殺到すると, そのうちのいくつかはえらく時間がかかったり, 最悪の場合タイムアウトして失敗したりするようになる. この実験で作るプログラムでは, 10 とでもしておけば十分だろう.

2.3 accept

accept は, 遂にクライアントからの接続を待つ操作である. listen との違いが分かりにくいかもしれないが, 例えて言えば listen は, 電話線をジャックに差し込む操作, accept は電話の前で電話が鳴るのをひたすら待つ操作である.

accept の結果, クライアントと通信をするための**新たなソケット**が返り値として返される. accept に渡したソケットで通信をするのでは**ない**事に注意. 実際 accept はこれ以降も新しい接続を受け付ける事ができる (し, そうあってほしい) ので, これは自然な API と言えるだろう. また accept の結果, サーバに何というアドレスのクライアントが接続をしてきたのかも返される. これは必要がなければ無視して構わない. 基本フォームは以下の通り.

```
struct sockaddr_in client_addr;
socklen_t len = sizeof(struct sockaddr_in);
int s = accept(ss, (struct sockaddr *)&client_addr, &len);
```

client_addr は, 情報を accept に渡すためのパラメータではなく, accept から, 接続してきたクライアントに関する情報を受けとるためのパラメータである. accept から返った時に, client_addr の要素 sin_addr.s_addr, sin_port が埋められ

ている, という仕組みである. それに伴って, 第 3 引数も渡した構造体のサイズその物ではなく, 返された構造体のサイズを受け取る変数 (のアドレス) となっている (ただし渡す際に, 第 2 引数で渡した構造体のサイズを入れておく必要があるので注意. これは, 何バイトの入れ物を用意して待っているかをシステムに教えるために必要である). また, 返り値は (成功していれば) **新しいソケット**で, 接続してきたクライアントとの送受信 (send/recv) をするのはこのソケットを使って行う.

本課題 8.1 以下のような動作をする C プログラム, serv_send.c を書け.

```
$ ./serv_send ⟨ ポート番号 ⟩ ⟨ ファイル名 ⟩
```

として起動すると, ポート番号で接続待ちに入る. 接続してきたクライアントに, ⟨ ファイル名 ⟩ の中身を送りつける.

つまり, 課題 **6.2** で client_recv の相手をしたサーバの動作である. client_recv と接続してみて動作を確認せよ. 当然ながら, ⟨ ファイル名 ⟩ として, /dev/dsp を選び, クライアント側で/dev/dsp へ出力すれば, サーバから入力した音をクライアントで鳴らす事ができる.

3. ネットワークの遅延測定

以前に iperf でネットワークの転送速度 (バンド幅) を測定したが, 今度は遅延を測ってみよう. 遅延というのは, ある 1 バイトが送信元を離れてから, 目的地へ到着するまでの時間である. ただし, 片道の時間を測るのは難しいので, 往復時間 (Round Trip Time; RTT) を測る.

それに対し, 性能の指標としてよく使われるのは転送速度 (1 秒あたり何バイト送れるか) である. 両者は似ているが異なる指標である. 道に例えると, 転送速度は道の太さ (何車線あるか) に相当する. 遅延は目的地までの距離である.

両者が異なるという事を納得するたとえ話をすると, 中東から石油を運ぶのに, 飛行機ではなくなぜ遅いタンカーを使うのか? それは言うまでもなくタンカーの方が一度に積める量が多いからである. もちろんある日中東を出発した石油一滴が日本に届くまでの時間 (遅延) は, タンカーの方が圧倒的に遅いのだが, それは石油を毎日別のタンカーで運びつづける事でカバーできる. つまり, 一日一隻のタンカーを出せば, 日本には一日一タンカー分の量 (バンド幅) で石油が届く事になる. 飛行機で同じだけのバンド幅を出すのは難しい事だろう. だから, 遅延が大きくてもバンド幅が大きいネットワークや, その逆というのが存在する.

この話からも分かる通り, 転送速度は大きなデータを送るのにかかる時間を支配する. 遅延は小さなデータを送るのにかかる時間を支配する. 逆に測定もそのようにすれば良いという事になるだろう.

本課題 8.2 以下のような動作をするプログラム pingpong_s.c と pingpong_c.c を書け.

- `$ ./pingpong_s SZ N`

  は, クライアントからの接続を待つ. 接続してきたら 1 バイトのメッセージを送信する. 以降は, SZ バイトのメッセージを受け取ったらすぐさま 1 バイトのメッセージを送信する, を N 回繰り返す.
- `$ ./pingpong_c SZ N`

  は, サーバへ接続し, 1 バイトのメッセージを受け取る. 以降は, SZ バイトのメッセージを送り, すぐさま 1 バイトのメッセージを受信する, を N 回繰り返す. この N 回にかかる時間を測定し, 1 往復あたりの平均時間 $\bar{t}$ を計算して表示する. それとともに, 転送バンド幅 (SZ / $\bar{t}$) も表示する.

両者を組み合わせれば, ネットワーク性能が測定できるだろう. メッセージが最小のときの往復時間はどれほどか (N を増やして測定する事). いろいろな SZ に対して転送速度を測り, gnuplot で可視化せよ. SZ は 1 バイトから 10MB

程度まで, 等比級数的に刻みを入れてデータを取る事. SZ を十分大きくしたときの転送速度は iperf と同じくらい出ているか?

サーバの擬似コードは以下のような感じ.

```
...
s = accept(ss, ...);
1 バイト送る;
for (i = 0; i < N; i++) {
  SZ バイト受け取る;
  1 バイト送る;
}
```

これはクライアント

```
...
connect(s, ...);
1 バイト受け取る;
t0 = 現在時刻;
for (i = 0; i < N; i++) {
  SZ バイト送る;
  1 バイト受け取る;
}
t1 = 現在時刻;
1 往復あたりの時間と転送速度を表示;
```

上記で現れた「現在時刻」を得る方法について. gettimeofday というシステムコールを用いる. 例によって man で調べる事. 以下のような関数を作ってしまうと良い.

```
/* 現在時刻 (単位: 秒) を浮動小数点数で返す */
double current_time()
{
  struct timeval tp;
  gettimeofday(&tp, NULL);
  return tp.tv_sec + tp.tv_usec * 1.0E-6;
}
```

4. 基本的なインターネット電話の完成

本課題 8.3 2 つのホスト間で会話ができる, 基本的なインターネット電話機能を実現せよ. つまり, 片方のホストのマイクに向かって喋るともう片方のスピーカに音が鳴る. それを飽きるまで続けていられるような物である. サーバとクライアントがどのような手順を守ってデータを送り合えば良いかを考えて, それぞれの動作を実現せよ.

ここまでを必須課題とする. 後は余力に応じてこの電話に機能, 性能, 設計その他の発展を施してもらいたい. 以下は, 候補として考えてもらいたいところである.

- 無音状態でネットワークをほとんど消費しないような, ネットにやさしい通信方法

- より一般に, 音声を送るための, 効率の良い符号化 (音データの表現. 圧縮)
- 発生から音が届くまでの遅延を小さくする
- 音質を上げる. それとデータサイズの両立
- 多者間通話を可能にする (一人が喋った声は残り全員に届く)
- 多者間通話で, 通話が始まった後新しい参加者を受け付けられるようにする

以降の内容は, これまで話した内容を単純に組み合わせるだけでは無理で, 発展的な内容や, 本テキストで説明していない API を自習する必要がある物もある.

適宜演習 HP 上で内容を補足しながら進める.

G3.　情報: 第3部

第9日

全時間実習とする．遅れている人はこの時間で追いつく．余裕のある人は発展課題に取り組む時間として使う．

第10日

全時間実習とする．遅れている人はこの時間で追いつく．余裕のある人は発展課題に取り組む時間として使う．

第11日

発表の時間. この日までに課題を動かし, その成果をスライドに書いて発表にまとめる事を目指す. 動いたソフトのデモンストレーションを行うこと. スライドを作るには, OpenOffice.org, Microsoft PowerPoint, TEX などのソフトを用いる. 詳細ルールは実験時間内および演習 HP 上で発表する.

第12日

発表の時間. この日までに課題を動かし, その成果をスライドに書いて発表にまとめる事を目指す. 動いたソフトのデモンストレーションを行うこと. スライドを作るには, OpenOffice.org, Microsoft PowerPoint, TEX などのソフトを用いる. 詳細ルールは実験時間内および演習 HP 上で発表する.