平成 30 年 11 月 23 日

京都工芸繊維大学コンピュータ部

Lime 58

はじめに

コンピュータ部副部長の馬杉です。諸事情により代理として馬杉が始めの挨拶を務めさせていただきます。この度はお忙しい中本学の学園祭にお越しいただき、また我々コンピュータ部にご興味、ご関心を持っていただきましてありがとうございます。そしてこのLime58 号を手にとって下さったことを、部を代表いたしまして厚く御礼申し上げます。この Lime は部員の活動の一部を記したものです。日々の活動の成果や部員個人の趣味全開なものまで内容は多岐にわたります。これを通じて、コンピュータ部のことを知っていただき、少しでも、我々の活動に興味を持っていただければ幸いです。そして、今年のLime 作成にあたり、編集および表紙の作成をしてくれた兼光君、忙しい中記事を書き上げた部員各位、並びに、OB・OG の方々を含め、日々我々の活動を支えてくださっているすべての方々への感謝をもって始めの挨拶とさせていただきます。

平成 30 年 11 月吉日

京都工芸繊維大学コンピュータ部副部長　馬杉 康平

目次

1 Pythonでワンライナーを書こう

機械課程 1 回生 山上 健

1.1 はじめに

初めまして。突然ですが、ワンライナーという言葉をご存知ですか？多くのプログラミング言語は複数行で記述されるのが普通ですが、あえて 1 行で書いてみようという縛りプレイのことです。今回使用する言語は Python です。Python は改行が重要な役割を持っているのでワンライナーには向いていなさそうですが、それがそうでもないということを伝えていきたいと思います。

記載する Python のソースコードはバージョン 3.6.5 で動作します。また、すべてのソースコードは決して悪意のあるものではありません。

1.2 ルール

- 改行は使わない (当たり前)
- 文 (代入文、**if 文**、**break** 等) は使わない
- **if 文**、**def 文**等の複合文をコロンで繋げない
- セミコロンは使用禁止
- 組み込み関数 **exec** や **eval** は原則不可

Python では式と文は厳格に区別されていて、文は必ず改行を伴います。コロン・セミコロンについては、使ってしまうと 1 行化のハードルがかなり下がってしまうのであえて使いません。**exec** 関数は使うか否かは個人の自由ですが、やはり難易度が下がってしまうので今回は使いません。

1.3 まず読んでみよう

下のソースコード 1.1 を見てください。実際に 1 行で書くとこんな感じになります。

```
1 (lambda a:a.extend(n for n in range(2,int(input('input: ')))if
      all(k*k>n and next(iter([]))or n%k for k in a))or print('
      output:',*a))([])
```

ソースコード 1.1 oneliner.py

初見では何をするプログラムかわからないかと思います。今は雰囲気だけ掴んでもらえれば結構です。紙面の都合上 1 行で表示されていないと思います。

実際に実行すると下のようになります。

```
>>> (lambda a:a.extend(n for n in range(2,int(input('input: ')))
    if all(k*k>n and next(iter([]))or n%k for k in a))or print('
    output:',*a))([])
input: 57
output: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53
```

正解は、入力した整数未満の素数を列挙するプログラムです。それでは上記のサンプルコードを読み解きながら、ワンライナーのポイントを説明していこうと思います。

1.4 解説

前章では 1 行のサンプルコードを載せましたが、1 行のままでは解説しづらいのでいくつか改行と空白を施した解説用のソースコード 1.2 を用います。

```
1 (
2     lambda a:a.extend(
3         n for n in range(2, int(input('input: ')))
4         if all(k*k > n and next(iter([])) or n%k for k in a)
5     ) or print('output:', *a)
6 )([])
```

ソースコード 1.2 oneliner2.py

1.4.1 代入

ワンライナーを書くときは、原型となる普通のソースコードを用意してから徐々に行数を減らしていくとやりやすいです。今回は解説用コード 1.2 を原型に戻しながら解説します。

下のソースコードは、解説用コード 1.2 の 1 つ手前のコードです。

```
1 a = []
2 a.extend(
3     n for n in range(2, int(input('input: ')))
4     if all(k*k > n and next(iter([])) or n%k for k in a)
5 ) or print('output:', *a)
```

ソースコード 1.3 oneliner3.py

1 行目で素数を格納するための空リストをグローバル変数に代入しています。しかし代入文は使ってはいけないルールなので、解説用コード 1.2 では **lambda 式**により無名関数を作り、ローカル変数として扱える引数に空リストを 6 行目で渡しています。

1.4.2 三項演算子

三項演算子とは下の **if** と **else** のことです。

```
>>> 'TRUE' if True else 'FALSE'
'TRUE'
>>> 'TRUE' if False else 'FALSE'
'FALSE'
```

実は **and** と **or** でも似たようなことができます。

```
>>> True and 'TRUE' or 'FALSE'
'TRUE'
>>> False and 'TRUE' or 'FALSE'
'FALSE'
```

解説用コードの 4 行目ではリスト中の素数 k の平方が n を超えるまで **or** に分岐し、n が k で割り切れるか評価します。ちなみに、**and** に続く部分はループの中断を表しています (詳しくは後述します)。

この **and** と **or** は、三項演算子の **if・else** よりも頻出のテクニックです。解説用コードの 5 行目の単体の **or** を見てください。**if・else** はセットでなければ三項演算子となり

ませんが、**or** と **and** は単体でも真偽を評価できます。下の例を見てください。

```
>>> True and 'Hello World'
'Hello World'
>>> True or 'Hello World'
True
>>> False or 'Hello World'
'Hello World'
>>> False and 'Hello World'
False
>>> None or 'Hello World'
'Hello World'
```

初めて見る方には少し難しいと感じられるかもしれません。**and** については、左側が偽ならば、右側が真でも偽でも偽を返すことに変わらないので、右側を評価せず左側をそのまま返します。逆に左側が真ならば、右側の真偽を確認する必要があるので、右側を返します。**or** についても同様です。なぜ上の文字列が真と判定されているのかはここでは述べませんので、気になる方は公式文献 [1] を参照してください。

さて、三項演算子の使い方をマスターしたらソースコードをまた 1 つ戻したものを見てみましょう。

```
1 a = []
2 a.extend(
3     n for n in range(2, int(input('input: ')))
4     if all(next(iter([])) if k*k > n else n%k for k in a)
5 )
6 print('output:', *a)
```

ソースコード 1.4 oneliner4.py

解説用コードの 5 行目の **or** の左側の **extend** は **None** を返し、偽として評価されて右側の **print** に続きます。**None** を返すメソッド単体の式文を繋げる場合は **or** を使いましょう。

1.4.3 内包表記

Python の内包表記は非常に強力で、ワンライナーに限らずいろんな場面で使われています。内包表記にはいくつか種類がありますが、今回は**ジェネレータ式**を使っています。前述した通り頻繁に使われる表現なので、使用法についての説明は今回は割愛します。詳細は公式文献 [2] を参照してください。

今回は素数をリストに追加するだけでなく、その追加された素数を新しい整数の素数判定に使っています。リストの **extend** メソッドは反復可能オブジェクトを引数に取るので、リストから追加済みの素数を取り出しつつ、それらを使って得られた新しい素数を返していくジェネレータを渡しています。

先程のソースコード 1.4 の 2 行目の **extend** メソッドに渡したジェネレータ式 (2 行目最後尾の丸括弧と 5 行目先頭の丸括弧で囲まれた部分) と、4 行目の **all** に渡したジェネレータ式を展開すると下のようになります。

```
1 a = []
2 for n in range(2, int(input('input: '))):
3     def g():
4         for k in a:
5             yield next(iter([])) if k*k > n else n%k
6     if all(g()):
7         a.append(n)
8 print('output:', *a)
```

ソースコード 1.5 oneliner5.py

外側のジェネレータは **for 文**をむき出しにして置き換えています。内側のジェネレータについては、後述する関数 **all** の引数として使いたいので**ジェネレータ関数**を定義して **yield 文**を記述しています。逆に、ジェネレータ関数を 1 行化する場合は **def 文**や **yield 文**は使えないのでジェネレータ式を使うことになります。

ちなみに、引数としてジェネレータ式を渡す場合はジェネレータ式の丸括弧は省略でき、引数を囲む丸括弧だけで十分です。

1.4.4 組み込み関数の活用

次は解説用コード 1.2 の 4 行目の組み込み関数 **all** の部分について説明します。これは「素数かどうか判定したい整数 n を、リストにある n の平方根以下の素数で小さい順に割っていき、そのすべてで割り切れなければ真を返す」素数判定器です。割り切れてしまった場合、素数ではないことが確定するのでループを抜ける仕組みになっています。

それでは、**all** を使う前のソースコードを見てみましょう。

```
1  a = []
2  for n in range(2, int(input('input: '))):
3      def g():
4          for k in a:
5                  yield next(iter([])) if k*k > n else n%k
6      for i in g():
7          if not i:
8              break
9      else:
10         a.append(n)
11 print('output:', *a)
```

ソースコード 1.6 oneliner6.py

6 行目に **break 文**、4 行目・7 行目に **for / else 文**が出現してしまいました。この形をワンライナーで書くときに組み込み関数 **all** が重宝します。

あまり見慣れない関数だと思うので公式文献 [3] の説明を引用します。

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

先程のソースコード 1.6 と見比べてみてください。形がかなり似ていると思いませんか？ **for 文**中の **return 文**は到達すると値を返すのでループが中断されます。よってループを中断する **break** に当てはまります。一方で、外側の **return** は最後までループし終わった後に実行されます。これは **for** を抜けた後に分岐する **else** に該当します。このようなフローをワンライナーで表現する場合は、ここで挙げた **all** や **any** 等の組み込み関数を活用しましょう。

1.4.5 その他小技

さて、三項演算子の説明のときに「後述する」と記述した、これです。

```
next(iter([]))
```

ここまでいくつかソースコードを載せましたが、ここだけ変わっていないことにモヤモヤしている人もいるかと思います。説明の前に、この部分を戻したソースコード 1.7 を下に示します。これが原型のソースコードになります。

```
a = []
for n in range(2, int(input('input: '))):
    def g():
        for k in a:
            if k * k > n:
                raise StopIteration
            else:
                yield n % k
    for i in g():
        if not i:
            break
    else:
        a.append(n)
print('output:', *a)
```

ソースコード 1.7 oneliner7.py

変化した部分がわかりましたか？この章のはじめに載せたコードを実行すると同等であることがわかると思いますが、実は意図的に例外を送出しています。通常、**try 文**や**with 文**を使わずに **raise 文**等で例外を発生させるとプログラムが強制終了します。しかし、ジェネレータ関数内でのこの **StopIteration 例外**の記述は決して珍しい表現ではなく、プログラムが強制終了しません。

まず、なぜ例外を送出しているのかを説明します。このプログラムの素数判定器は、「素数判定したい数 n を素数で割っていく」試し割りです。効率化を考えて、2 以上 n の平方根以下の素数を使えば十分です。よって「割る数 k の平方が n を超えるまでループさせ、途中で割れたら素数ではないとして偽を返して中断し、割れなければ真としてループを中断する」という 2 種類の中断があれば実現できそうです。前者は **all** の構造と一致しますが、後者は「ループはまだできるがジェネレータはここで終える」という意味を持たなければなりません。**all** が真のまま内部のループを中断させるために、ループを中断できる例外を送出しています。

では、なぜ例外を送出してもプログラムが正常に動作するのでしょうか？ **for** ループが反復可能オブジェクトを参照するとき、終端に達すると反復可能オブジェクトは **StopIteration** を送出します。この例外を **for** は終端の合図として受け取りループを終了する仕組みになっています。空のリストを **for** に渡してもエラーにならないのはこの仕組みによるものです。質問サイトのある投稿 [4] を参考にさせていただきました。

今回は **raise 文**が使えないので、**StopIteration** を送出することができる組み込み関数 **next** と、その引数として空のリストのイテレータオブジェクトを利用することで 1 行に収めています。

ちなみに、この小技を使わずとも標準ライブラリ **itertools** の **takewhile** 関数を利用することで同じことができます。**itertools** はワンライナーではよく使うライブラリなので、気になる方は公式文献 [5] や、私がワンライナーを書き始めた頃に参考にした記事 [6] を参照してください。

1.5 おわりに

今回は自分の知識を多くの方と共有したいという思いで執筆しました。自作のソースコードを解説しながら進める方針にした結果、詳しい部分まで説明することができなかったと感じます。この拙文を読んでワンライナーに興味を持っていただければ嬉しい限りです。Python に限らずプログラミング言語 1 つあればすぐに始めることができ、頭の体操としてもおすすめなので是非挑戦してみてください。

参考文献

[1] 真理値判定 - Python 3.6.5 ドキュメント https://docs.python.jp/3/library/stdtypes.html#truth

[2] ジェネレータ式とリスト内包表記 - Python 3.6.5 ドキュメント https://docs.python.jp/3/howto/functional.html#generator-expressions-and-list-comprehensions

[3] 組み込み関数 **all** - Python 3.6.5 ドキュメント https://docs.python.jp/3/library/functions.html#all

[4] Why does next raise a 'StopIteration', but 'for' do a normal return? - Stack Overflow https://stackoverflow.com/a/14413978

[5] itertools - Python 3.6.5 ドキュメント https://docs.python.jp/3/library/itertools.html

[6] 闇 Pythonista 入門 (Python ワンライナーのテクニック集) http://cocu.hatenablog.com/entry/2014/04/06/021355

2 安全に使う Raspberry Pi の GPIO

情報工学課程 2 回生 寺村英之 (aka. ikubaku or codeworm)

皆さんこんにちは、寺村英之です。今回は Raspberry Pi の GPIO を扱う時に気をつけたいことについて書こうと思います。

まずは Raspberry Pi に関して少し解説します。

2.1 Raspberry Pi の特徴と GPIO

Raspberry Pi はクレジットカードサイズ[*1]のパワフルなマイコンボードです。自分でプログラムを設計して様々なものを作ることをできます。Raspberry Pi の機能はパソコンと似ていて、ディスプレイとキーボード、マウスにつなぐとこれだけで開発をすることができます。読者の方々の中には、Raspberry Pi を使って AI スピーカーやロボットなどを作ったことがある方がいらっしゃるかもしれません。

Raspberry Pi は、普通のパソコンと違って GPIO という機能を持っています。これによって搭載されている端子から任意の信号を送り出し[*2]、あるいは受け取って外部の機器を制御したりセンサから値を得たりすることができます。これを応用すればロボットやスマートドアロックなどを作ることができるでしょう。

2.2 GPIO の起動時状態

マイコンボードの GPIO を使うにあたって気をつけるべきこととして、起動直後の GPIO の状態があります。これをよく確認しておかないとつないでいる機器が誤動作したり最悪の場合 Raspberry Pi が破損することがあります。GPIO の初期状態について説明する前にまず、その前提となる GPIO の動作モードについて簡単に解説します。

[*1] もう少し大きい。

[*2] もちろん限界もあり、例えば極めて高速にスイッチングができないことなどがある。Raspberry Pi 側にハードウェア支援がある I^2S などはその限りではない。

2.2.1　GPIO の動作モード

GPIO の端子それぞれには多くの場合次の 3 つの状態があります。

1. 入力か出力か
2.（出力端子ならば）最初の出力レベル（電圧）
3. プルアップ・プルダウン抵抗の有無

もし GPIO が入力であるならばその端子はあたかも大きな抵抗に接続されているかのように振る舞います。したがって極端な話、ここに + 側の電源を直接繋いだとしても問題は起こりません。しかし出力に設定されている場合は少し違います。出力に設定されているならば、その端子は直接 + 側の電源か GND に接続されたようになります。この端子を GND や低いインピーダンスの回路に接続すると、過電流が流れて回路が破損することがあります。

出力端子に設定されている端子はある出力レベルを持ちます。出力端子は、それぞれのレベルに対応する側の電源端子に接続されているように振る舞います。消費電流の低いものは直接駆動することができますが、前述のように、例えば直接接地すると回路が破損する原因になります。

GPIO 端子によってはプルアップ・プルダウン抵抗を持っていることがあります。これによって、何も入力されていないとき、あるいは出力端子であっても、実際には駆動されていないとき（例えばオープンドレインである端子）にどちらかのレベルに固定することができます。マイコンボードによってはこれをソフトウェア側から切り替えることができます[*3]。しかし、外部の回路でさらにプルアップ・プルダウンされていると問題が起こることがあります。例えばプルアップ・プルダウン抵抗が混在していると電圧レベルが中途半端な位置になってしまいます。また、混在していなくても、信号線全体では抵抗値が下がってしまいその線を駆動しづらくなってしまうことがあります。

2.3　Raspberry Pi の GPIO 初期状態

RaspberryPi の GPIO の初期状態は本家の説明書によれば、起動時のすべてのの入力端子状態は次のようになっています [1]。

- 入力状態

[*3] 切り替えられなければ GPIO の状態ではないともいえる。

- プルアップ・プルダウン抵抗の設定はデフォルトのものが使われる

プルアップ抵抗の設定値は Raspberry Pi に使われている SoC[*4]のデータシート [2] の 102 ページにある表に書かれています。ちなみに、このデータシートは BCM2835 という初代 Raspberry Pi などに使われている SoC のものです。しかしこの文書に書かれている部分は、今の所最近使われている SoC でも特に変わっていないそうなので [3]、この資料を参照すれば良いです。では、実際にデータシートを見ていきましょう。

データシートの 91 ページを参照すると、`GPFSELn`レジスタ群のすべてが `0`になっています。したがって、実際に最初は入力に設定されていることがわかります。また、先程述べた表によると、大体の端子が最初はプルダウンされるようです。しかし一部の端子はプルアップされるらしく注意が必要です。

実は、この資料に当たる前に[*5]テスターと他のマイコンボードを使って起動時の GPIO 端子の様子を観察してみました。そのときに、少し気になる挙動をする端子があったので表 2.1 に示します。観察したのは Raspberry Pi 3（RS Components 製日本モデル Ver. 1.2）、OS には Raspbian（NOOBS-2.9.0 からインストール）に LXQt デスクトップ環境を導入したものを用いました。またシリアルコンソールとシリアルポートを両方無効化するように `raspi-config`で設定しました[*6]。H レベルに駆動されているのかプルアップされているのかはテスターに出る電圧の高さで切り分けました[*7]。

表 2.1 特殊な挙動をする GPIO 端子

GPIO 番号	最終的な状態	備考
2	H レベルに駆動	
3	H レベルに駆動	
14	H レベルに駆動	起動中に電圧レベルが変化する
15	プルアップ	

ここで GPIO 番号とは GPIO の識別番号であって、物理的なピン番号（ピンヘッダのピンの通し番号）ではないことに注意してください。GPIO 番号は Raspberry Pi の説明書における GPIO の節[*8]で使われている番号を指します。

[*4] System on a Chip。CPU の他ペリフェラルモジュールやディスプレイコントローラなど一通りの回路（チップセット）を 1 つの IC に収めたもの。
[*5] 厳密には BCM2835 のデータシートに当たる前に。
[*6] 他の設定はインストール時から触っていない。
[*7] プルアップで電圧が高くなっている場合は駆動されている場合よりも低く出る。
[*8] https://www.raspberrypi.org/documentation/usage/gpio/README.md

実際に調べてみると、プルアップ・プルダウン抵抗の状態からデータシートにあったGPIO 番号と、Raspberry Pi の説明書における GPIO 番号は一致しているらしいことがわかりました。しかし表 2.1 のように例外があります。以下でそれらの端子について解説します。

GPIO2 と 3 は I^2C バスに使うことができる端子です。これは I^2C の仕様 [4] でバスが動いていないときは、すべての信号線が H レベルでなければならない[*9]ために、Raspberry Pi 側で H レベルを出力するようにしているのだと思われます。

GPIO14 と 15 は UART に使われます。初期設定ではれらをパソコンのシリアルポートに接続することで、Tera Term などの端末エミュレータから Raspberry Pi を操作することができます。今回は無効にしていましたが、起動時に電圧レベルの変化などが見られました。

これらの端子の状態はデータシートにある初期状態とは異なっています。おそらく起動時に読み込まれる `bootcode.bin`などのファームウェアや、OS によって起動時に操作されているためだと思われます。特に I^2C バスを使うように設定していた場合などはそうなる傾向があるのかもしれません。しかし UART に使われる端子のように機能を無効化していても操作が行われる端子があります。その上、これらの挙動は OS やファームウェアの更新で変わってしまうかもしれません。

これらから、Raspberry Pi で GPIO を用いる際には GPIO の初期状態に注意する必要があることがわかりました。特に特殊な機能を持っている端子とプルアップ抵抗が適用されている端子は特別の注意が必要です。一方それ以外の端子はプルダウン抵抗の存在にだけ注意すれば、当分の間は問題ないでしょう。

2.4 おわりに

今回は Raspberry Pi の GPIO を使うにあたって注意するべきことについてまとめました。Raspberry Pi に他の機器の出力端子を接続したときや、予期せず Raspberry Pi が再起動したときに困ったことにならないように、気をつけて回路を設計しましょう。

ここで少し宣伝です。私は主に PC やマイコンボードなどを使って何か作ることをやっています。これまではほとんど自己満足におわっていましたが、最近は Twitter などで作っている過程やできたものを紹介したりしています。興味のある方は ID: @ikubaku10

[*9] 出典の 8 ページ 3.1.1 に “When the bus is free, both lines are HIGH.” とある。ちなみに I^2C の仕様ではバスマスタやスレーブの内部でプルアップすることは認められておらず、外側でプルアップしなければならない。

でツイートしているのでぜひご覧ください。では、また。

参考文献

[1] “GPIO - Raspberry Pi Hardware Documentation” Raspberry Pi Foundation https://www.raspberrypi.org/documentation/hardware/raspberrypi/gpio/README.md - 2018/11/6 閲覧

[2] “BCM2835 ARM Peripherals” Broadcom Inc. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf - 2018/11/6 閲覧

[3] “BCM2837 - Raspberry Pi Hardware Documentation” Raspberry Pi Foundation https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md - 2018/11/6 閲覧

[4] “UM10204 I^2C-bus specification and user manual Rev. 6 — 4 April 2014” NXP Semiconductors N.V. https://www.nxp.com/docs/en/user-guide/UM10204.pdf - 2018/11/6 閲覧

3 C言語による色付き出力

情報工学課程 2 回生 兼光 琢真

3.1 はじめに

プログラミングを試したことがある人は文字列を出力することからまず始めたのではないでしょうか？典型例として “Hello, World!” を表示させたり…．さて，その出力に色をつけてみよう，というのが本記事の内容です．

また，先にお断りしておきますが，出力に色がつかない環境もあります．True Color に対応しているターミナルに文字列を出力するとうまく動きます．

プログラミングを知っている人も，そうでない人も，少しでも楽しんでいただけることを願って書きました．

3.2 単語の意味

プログラム (コマンドなども) を実行すると文字がずらっとでてきたりするウィンドウのことを**ターミナル**と呼びます．コマンドプロンプトや端末エミュレーターなどとも聞きますが，本記事内では統一してターミナルと呼びます．

コンピューターで扱える文字には “あいう …” などの普通の文字の他にも，改行を表す改行文字というものがあります．ちなみにプログラミングで改行文字を扱うときは，よく’\n’ で表します．こういった文字は普通の文字とは違い，そのものが表示されるわけではありません．そういう文字を**制御文字** (control sequence) と呼びます．

ANSI エスケープシーケンス (ANSI escape sequence) は制御文字のようなものでターミナルの出力を操れます．つまり，特別な意味を持つ文字列のことです．これで “ターミナルに出力される文字列の色を変更する役割をもつ文字列” を表すことができます．

ANSI エスケープシーケンスで色を変更するときは，その色を **RGB** で指定します．RGB は色を 3 つの Red，Green，Blue 要素に分解して，それらを 0 から 255 の間の整数

表 3.1 RGB による色の表現

色	(R, G, B)	16 進表記
赤	(255, 0, 0)	0xFF0000
緑	(0, 255, 0)	0x00FF00
白	(255, 255, 255)	0xFFFFFF

で表したものです．また，文字で RGB を表すときは 16 進表記をよく使います．カラーコードで検索すると，色と 16 進表記との対応がすぐに得られると思います．

さらに本記事内では RGB 全色が表現できる方式を **True Color** と呼びます．

今回は ANSI エスケープシーケンスを利用して，True Color に対応しているターミナルに色付き文字列を出力します．

3.3 実行例

まずは実際に色がつけられることを図 3.1 で確認します．`set_terminal_color` という名前の実行可能ファイルを実行していますが，その際に背景色 (`0xFF00FF`) と表示させたい文字列 (`Hello!`) を指定しています．

図 3.1 出力に色が付きました!

次に，このプログラムとその仕組みを紹介します．最後に別の (多分) 面白そうなプログラムとその実行例を載せてこの記事を終わります．

3.4 メイン関数

```
1 #include <stdio.h>   // 出力のため
2 #include <stdlib.h> // 数の変換のため
3
4 #include "setcolor.h" // 色を変えるため
5
6 int main(int argc, char const *argv[]) {
```

```
7       // 背景色と表示したい文字列が渡されていそうか確認
8       if (argc >= 3) {
9           // ここに背景色が文字列で入る
10          char const *hex_color_code = argv[1];
11          // 16 進数値の背景色を整数値に変換
12          unsigned int rgb = strtoul(hex_color_code, NULL, 16);
13          // 変換して得た整数値が RGB 値だと分かるよう準備
14          struct rgb_color color = rgb_from(rgb);
15
16          // 出力先の背景 (BACKGROUND) の色を変える
17          set_terminal_color(BACKGROUND, color);
18          // 実行時に渡された文字列を表示してから改行する
19          puts(argv[2]); putchar('\n');
20          // 一応，変えた色を戻しておく
21          reset_terminal_color(RESET);
22      }
23      // いつでも正常終了します
24      return 0;
25  }
```

ソースコード 3.1　main 関数

はじめの方に include している setcolor.h に出力する文字列の色を変える関数などを定義しています (表 3.4).

表 3.2　setcolor.h で定義するもの

定義するもの	役割
enum ground_type 型	背景，文字を表す
enum reset_type 型	背景，文字，その両方を表す
union rgb_color 型	RGB 値を表す
set_terminal_color 関数	色を変える
reset_terminal_color 関数	色を戻す

3.5　setcolor.h

setcolor.h には表 3.4 に示したことしか書かれていません.

```
1  #ifndef SETCOLOR_H_
2  #define SETCOLOR_H_
3
4  #include <stdint.h>
5
6  enum ground_type {
7      FOREGROUND = 38,
8      BACKGROUND = 48,
9  };
10
11 enum reset_type {
12     RESET = 0,
13     RESET_FOREGROUND = 39,
14     RESET_BACKGROUND = 49,
15 };
16
17 struct rgb_color {
18     uint8_t r;
19     uint8_t g;
20     uint8_t b;
21 };
22
23 struct rgb_color rgb_from(int);
24 struct rgb_color rgb(int, int, int);
25 void set_terminal_color(enum ground_type type, struct rgb_color
       color);
26 void reset_terminal_color(enum reset_type type);
27
28 #endif /* SETCOLOR_H_ */
```

ソースコード 3.2 setcolor.h

3.6 setcolor.h の実装

上の `setcolor.h` に定義した関数は次の `setcolor.c` で実装されています．こちらは ANSI エスケープシーケンスによる色の変え方を知れば，ソースコード 3.3 の通りたったの数行で実装できます．ですから，先にそれを紹介します．

改行文字 ('\n') があるように Esc 文字 ('\x1b') があります．表 (3.6) に示した通り ANSI エスケープシーケンスを出力すると，True Color に対応しているターミナルはその効果の通りに振る舞います [1].

表 3.3 ANSI エスケープシーケンスの例

ANSI エスケープシーケンス	効果
\x1b[0m	色も含めすべての効果を消す
\x1b[38;2;R;G;Bm	文字の色を (R, G, B) に指定
\x1b[39m	文字の色をデフォルトに戻す
\x1b[48;2;R;G;Bm	背景の色を (R, G, B) に指定
\x1b[49m	文字の色をデフォルトに戻す

```c
#include <stdio.h> // printf 関数で出力するため

#include "setcolor.h"

void set_terminal_color(enum ground_type type, struct rgb_color color) {
    printf("\x1b[%d;2;%d;%d;%dm", type, color.r, color.g, color.b);
}

void reset_terminal_color(enum reset_type type) {
    printf("\x1b[%dm", type);
}

struct rgb_color rgb(int r, int g, int b) {
    return (struct rgb_color ) { r, g, b } ;
}

struct rgb_color rgb_from(int code) {
    return rgb((code & 0x00FF0000) >> 16, (code & 0x0000FF00) >> 8, code & 0x000000FF);
}
```

ソースコード 3.3 setcolor.c

3.7 別の実行例

少し長いですがソースコードを載せたあとに実行例を示します．出力画面は少し面白いと思います．

```
1  #include <stdio.h>  // 出力のため
2  #include <unistd.h> // Unix 系依存．処理を少しの時間とめるため
3
4  #include "setcolor.h" // 色を変えるため
5
6  #define IS_IN(x, min, max) ((min) <= (x) && (x) <= (max))
7  #define ABS(x) ((x) >= 0 ? (x) : -(x))
8
9  // hsl:  Hue, Saturation, Lightness
10 struct rgb_color hsl(double H, double S, double L) {
11     if (!IS_IN(H, 0, 360) || !IS_IN(S, 0, 1) || !IS_IN(L, 0, 1))
           {
12         // TODO parameter がおかしいから処理を終了するコード
13     }
14     double C = (1 - ABS(2 * L - 1)) * S;
15     double Hp = H / 60;
16     double Hp_mod2 = Hp - 2 * (int) (Hp / 2);
17     double X = C * (1 - ABS(Hp_mod2 - 1));
18
19     double R_1 = 0, G_1 = 0, B_1 = 0;
20     switch ((int) Hp) {
21     case 0:
22         R_1 = C;
23         G_1 = X;
24         break;
25     case 1:
26         R_1 = X;
27         G_1 = C;
28         break;
29     case 2:
30         G_1 = C;
31         B_1 = X;
32         break;
33     case 3:
```

```
34          G_1 = X;
35          B_1 = C;
36          break;
37      case 4:
38          B_1 = C;
39          R_1 = X;
40          break;
41      case 5:
42          B_1 = X;
43          R_1 = C;
44          break;
45      default:
46          break;
47      }
48
49      double m = L - C / 2;
50
51      double r = (R_1 + m) * 255;
52      double g = (G_1 + m) * 255;
53      double b = (B_1 + m) * 255;
54
55      return rgb(r, g, b);
56  }
57
58  void print_space_line(size_t columns) {
59      while (columns--)
60          putchar(' ');
61      putchar('\n');
62  }
63
64  int main() {
65      double H = 0, S = 1, L = .9, hue_step = 7;
66
67      // 0.02秒おきに次のブロックを実行し続ける
68      while (!usleep(20000)) {
69          // 色相パラメーターを少し変える
70          if ((H += hue_step) > 360)
71              H -= 360;
72          // 背景色を設定
73          set_terminal_color(BACKGROUND, hsl(H, S, L));
74          // 1行(36個分のスペース)を出力
75          print_space_line(36);
```

```
76         }
77 }
```

ソースコード 3.4　setcolor を用いた別のプログラム

実は色の表し方は RGB の他にも HSL という方式があります．この方式では Hue(色相)，Saturation(彩度)，Lightness(輝度) の 3 つのパラメーターで色を指定します．出力は RGB 値を扱いますが，ソースコード上では HSL で色を指定したいので，HSL から RGB に変換する `hsl` 関数を [2] にある通り素直に実装しています．この実装がソースコードの大部分を占めています．下から十数行の `main` 関数を読めば何をするプログラムなのか分かると思います．

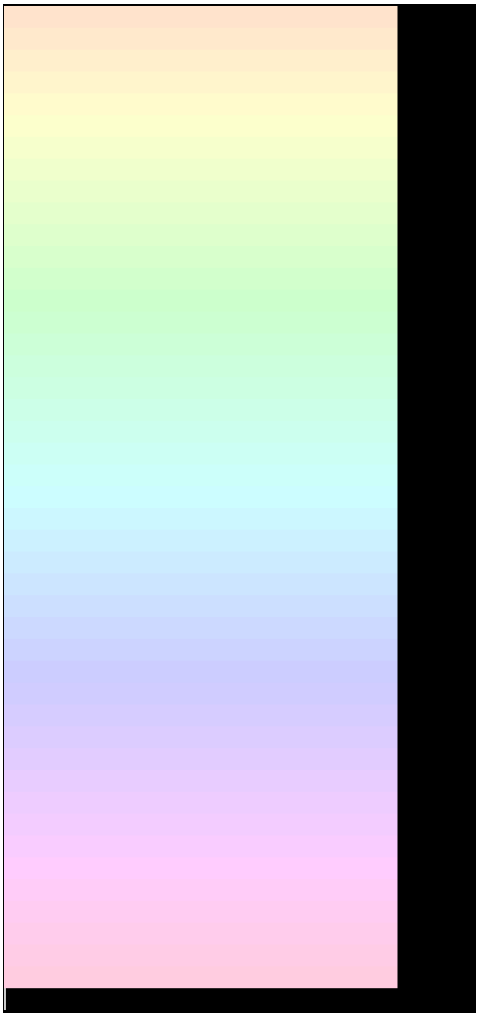

図 3.2　ターミナルを飾るグラデーション

図 3.2 が実行例です．スクリーンショットにしたら止まってしまいますが，実際にはプログラムを強制終了させるまで色が連続的に変化し続けます．

観察していると感じたことですが，色が大きく変わるように見える部分と，あまり変わらないように見える部分があります．HSL の色相を表すパラメーター，H は各行で同じだけ変化させているのに，そうなるのは不思議だなあと思ったりしました．何にせよぼんやり見ていると落ち着きました．

3.8 終わりに

私は普段からこんな感じのちょっとしたものを作って遊んだりしていますが，ソースコードを書いては，投げっぱなしになることがほとんどです．しかし，今回は自分のソースコードを見直す機会が何度もありました．正直なところ，動きはしますが良いプログラムでは無いと思います．だから，より良いプログラムと明確な文章を書けるよう，これからも精進していきたいと思います．

また，プリントされたものは白黒なので，色の様子が分かりづらいかもしれません．KITCC のホームページからカラーの PDF 版を見ることができます．そちらでもご覧いただけると嬉しいです．最後まで読んでいただきありがとうございました．

参考文献

[1] ANSI Escape code: https://en.wikipedia.org/wiki/ANSI_escape_code
[2] HSL and HSV: https://en.wikipedia.org/wiki/HSL_and_HSV

4 自作 OS をネットワーク対応してみる

情報工学課程 4 回生 松永大輝

4.1 はじめに

この記事は，趣味で作っている簡単な OS[1] にネットワーク機能を載せてみたときの作業記録のようなものです．

4.2 NIC ドライバの実装

まずはネットワークインターフェースカード (NIC) のドライバを書きます．NIC ドライバが実装すべき関数としては，オープン，クローズと送信 (Tx)，受信 (Rx) があれば最低限なんとかなりそうです．ソースコード 4.1 に定義を示します．

```
1 struct netdev_ops {
2   int (*open)(int minor);
3   int (*close)(int minor);
4   int (*tx)(int minor, struct pktbuf *pkt);
5   struct pktbuf *(*rx)(int minor);
6 };
```

ソースコード 4.1 NIC ドライバのインターフェース

今回は Realtek の RTL8139 という Ethernet NIC を用いることにします．RTL8139 を選んだ理由としては，

1. 機能が少なく，仕組みが単純でプログラミングが楽
2. QEMU でサポートされている

が挙げられます．

RTL8139 に関して，以下のような資料があります．

- RTL8139D(L) データシート [2]
- RTL8139(A/B) Programming guide V0.1[3]
- OSDev RTL8139[4]

存在確認

`rtl8139_probe()`(ソースコード 4.2) は OS 起動時に一度だけ呼ばれ，RTL8139 の存在を確認します．RTL8139 は PCI バス接続であるため，RTL8139 のベンダ ID とデバイス ID を持つデバイスを検索することで存在確認ができます．

```
1  #define RTL8139_VENDORID 0x10ec
2  #define RTL8139_DEVICEID 0x8139
3
4  DRIVER_INIT int rtl8139_probe() {
5    struct pci_dev *thisdev = pci_search_device(RTL8139_VENDORID,
         RTL8139_DEVICEID);
6    if(thisdev != NULL) {
7      rtl8139_init(thisdev);
8    }
9    return (thisdev != NULL);
10 }
```

ソースコード 4.2 rtl8139_probe()

初期化

RTL8139 が存在していた場合，`rtl8139_init()`(ソースコード 4.3) で初期化を行います．

まず，workqueue を送信用と受信用の 2 つ準備しています．これについては後述します．

その次に，I/O ベースアドレスと割り込みラインの情報を PCI コンフィグレーション空間から読みだしています．どちらもデバイスの制御に必須の情報です．I/O ベースアドレスが分かると，それにオフセットを加算することで RTL8139 の各レジスタへアクセスできるようになります．表 4.1 に RTL8139 の主なレジスタを示します．完全なレジスタ一覧はデータシートにあります [2]．次の for 文では，IDR0-5 レジスタから MAC アドレスを取得しています，

その後は，PCI バスマスタリング (DMA のような機能) の有効化，パワーオン，ソフトウェアリセット，受信バッファの物理アドレスの設定，割り込みマスクの設定，受信動

表 4.1 RTL8139 の主なレジスタ

レジスタ	オフセット	説明
IDR0-5	0x0	ID Register 0-5
TSD0-3	0x10	Transmit Status of Descriptor 0-3
TSAD0-3	0x20	Transmit Start Address of Descriptor 0-3
RBSTART	0x30	Receive (Rx) Buffer Start Address
CR	0x37	Command Register
CAPR	0x38	Current Address of Packet Read
IMR	0x3C	Interrupt Mask Register
ISR	0x3E	Interrupt Status Register
RCR	0x44	Receive (Rx) Configuration Register
CONFIG0-1	0x51	Configuration Register 0-1

作の設定，送受信開始と続きます．

最後に，先程ゲットした割り込みライン番号の割り込みハンドラ (rtl8139_inthandler()) を登録し，割り込みを受け付けられる状態にします．

```
1  void rtl8139_init(struct pci_dev *thisdev) {
2    rx_wq = workqueue_new("rtl8139_rx_wq");
3    tx_wq = workqueue_new("rtl8139_tx_wq");
4
5    rtldev.pci = thisdev;
6    rtldev.iobase = pci_config_read32(thisdev, PCI_BAR0);
7    rtldev.iobase &= 0xfffc;
8    rtldev.irq = pci_config_read8(thisdev, PCI_INTLINE);
9    rtldev.rxbuf_index = 0;
10   rtldev.txdesc_head = rtldev.txdesc_tail = 0;
11   rtldev.txdesc_free = TXDESC_NUM;
12   /* 省略 */
13
14   struct ifaddr *eaddr = malloc(sizeof(struct ifaddr)+
         ETHER_ADDR_LEN);
15   eaddr->len = ETHER_ADDR_LEN;
16   eaddr->family = PF_LINK;
17   for(int i=0; i<ETHER_ADDR_LEN; i++)
18     eaddr->addr[i] = in8(RTLREG(IDR)+i);
```

```
19
20    /* 省略 */
21
22    //enable PCI bus mastering
23    u16 pci_cmd = pci_config_read16(thisdev, PCI_COMMAND);
24    pci_cmd |= 0x4;
25
26  #define PCI_CONFIG_ADDR 0xcf8
27  #define PCI_CONFIG_DATA 0xcfc
28    u32 addr = (thisdev->bus<<16) | (thisdev->dev<<11) | (thisdev->
          func<<8) | (PCI_COMMAND) | 0x80000000u;
29    out32(PCI_CONFIG_ADDR, addr);
30    out16(PCI_CONFIG_DATA, pci_cmd);
31
32    //power on
33    out8(RTLREG(CONFIG1), 0x0);
34    //software reset
35    out8(RTLREG(CR), CR_RST);
36    while((in8(RTLREG(CR))&CR_RST) != 0);
37    //set rx buffer address
38    out32(RTLREG(RBSTART), KERN_VMEM_TO_PHYS(rtldev.rxbuf));
39    //set IMR
40    out16(RTLREG(IMR), 0x55);
41    //receive configuration
42    out32(RTLREG(RCR), RCR_ALL_ACCEPT | RCR_WRAP);
43    //enable rx and tx
44    out8(RTLREG(CR), CR_RE|CR_TE);
45    //setup interrupt
46    idt_register(rtldev.irq+0x20, IDT_INTGATE, rtl8139_inthandler);
47    pic_clearmask(rtldev.irq);
48  }
```

ソースコード 4.3 rtl8139_init()

パケットバッファ

プロトコルスタックにてパケットを生成する場面では，上位層から下位層へとパケットの先頭にヘッダを付け足していきます．逆に，受信パケットの処理を行う際は，下位層から上位層へとヘッダを取り除いていきます．メモリ領域の先頭アドレスを持ち運ぶのでは，かなりやりずらそうです．

そこで，pktbuf というものを用意します．pktbuf には次のような操作が可能です．

- pktbuf_alloc：指定サイズの pktbuf を生成
- pktbuf_create：指定されたメモリ領域，サイズ，メモリ解放関数で pktbuf を生成
- pktbuf_free：メモリ解放
- pktbuf_get_size：サイズの取得
- pktbuf_reserve_headroom：先頭領域の確保
- pktbuf_add_header：先頭領域の追加
- pktbuf_remove_header：先頭領域の削除
- pktbuf_copyin：メモリ領域をコピー

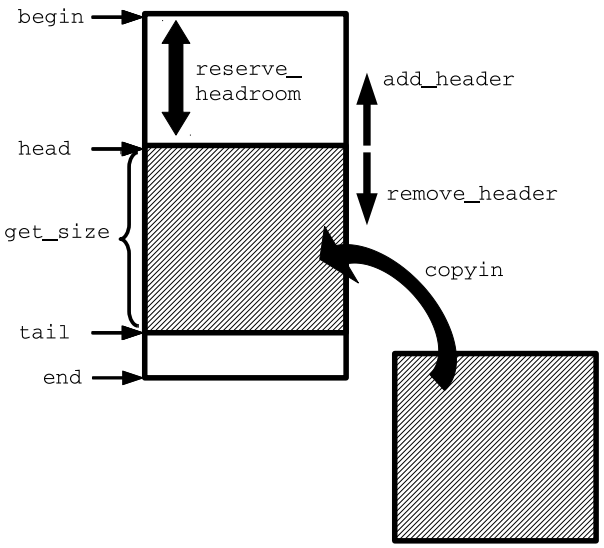

図 4.1 pktbuf

pktbuf は begin - end，head - tail の 4 つのポインタで管理されており，begin と end はメモリ領域の先頭と末尾を指しています (図 4.1)．head と tail は，使用中の領域の先頭と末尾を指しており，ヘッダの追加/削除によって変化します．head を加算することでヘッダを削除します．head を追加されうるヘッダサイズだけ加算しておくことで，後々 head を減算しながらヘッダの追加ができます．

なお，メモリ領域の拡張はできないため，あらかじめ十分な量のサイズを確保しておく必要があります．

pktbuf のようなネットワーク用バッファとして，Linux では skb，FreeBSD では mbuf が用意されています [5][7]．共に非常に高機能かつ複雑です．

パケットの送信

パケット送信のインターフェースが，rtl8139_tx() です．送信キューにパケットを入れ，rtl8139_tx_all() の実行を workqueue に依頼します．

workqueue は遅延実行のための機構で，Linux カーネルの同名の機能を参考にしています [6][7]．1 つの workqueue に 1 つカーネルスレッドが対応し，依頼された関数の実行を順次行います．該当スレッドがスケジューリングされるまで関数の実行が遅延されます．

rtl8139_tx_all() は rtl8139_tx_one()(ソースコード 4.4) を繰り返し呼びます．rtl8139_tx_one() は送信キューからパケットを 1 つ取り出して送信を行います．

RTL8139 には送信ディスクリプタが 4 つあります．各ディスクリプタは 1 つのパケットの送信を管理します．送信の仕方は簡単で，空いている送信ディスクリプタにパケットの先頭物理アドレスとサイズを設定するだけです．割り込みで送信の成功/失敗が通知されます．送信が完了すればそのディスクリプタは次の送信に使えるようになります．4 つのディスクリプタは順繰りに使用していきます．

rtldev.txdesc_free は空き送信ディスクリプタ数を，rtldev.txdesc_head は次の送信ディスクリプタ番号を示します．TSAD0-3 レジスタにはパケットの先頭物理アドレスを，TSD0-3 レジスタの下位 13bit にはパケットサイズを設定します．

```
1  int rtl8139_tx_one() {
2    /* 省略 */
3
4    if(rtldev.txdesc_free > 0) {
5      struct pktbuf *pkt = NULL;
6      if(queue_is_empty(&rtldev.txqueue)) {
7        error = -1;
8        goto out;
9      }
10
11     pkt = list_entry(queue_dequeue(&rtldev.txqueue), struct
           pktbuf, link);
12
13     out32(RTLREG(TX_TSAD[rtldev.txdesc_head]), KERN_VMEM_TO_PHYS(
           pkt->head));
14     rtldev.txdesc_pkt[rtldev.txdesc_head] = pkt;
15     out32(RTLREG(TX_TSD[rtldev.txdesc_head]), pktbuf_get_size(pkt
           ));
16     rtldev.txdesc_free--;
17     rtldev.txdesc_head = (rtldev.txdesc_head+1) % TXDESC_NUM;
```

```
18      /* 省略 */
19    }
20
21    /* 省略 */
22  }
```

ソースコード 4.4 パケット送信関連の関数

パケットの受信

パケット受信のインターフェースが，rtl8139_rx() です．受け取ったパケットは最終的に受信キューに入るので，それを待ちます．

パケットが NIC に到着すると，まず割り込みが発生します．割り込みが発生すると，rtl8139_isr()(ソースコード 4.5) が呼ばれます．割り込みは受信時だけでなく，送信完了時やエラー検出時にも発生します．

rtl8139_isr() では，まず ISR レジスタを読んで割り込みの原因を取得します．次の while 文では，送信ディスクリプタに空きができたかチェックします．

その後は割り込みの原因に応じて処理を行います．送信完了であれば次の送信を依頼，受信完了であれば rtl8139_rx_all()(ソースコード 4.6) の実行を workqueue に依頼します．

rtl8139_rx_all() では，rtl8139_rx_one() を繰り返し呼び出し，可能であれば一度に複数パケットを受信します．パケットを受信した場合は Ethernet フレーム処理関数 ether_rx() の実行を workqueue に依頼します．ether_rx() は Ethernet フレームを解析して IP などの上位層へとパケットを引き渡します．

workqueue に処理を依頼することで，複数パケットの受信やパケットの処理を割り込みの外で実行できます．それにより，割り込みコンテキストでの実行時間を短縮できます．

rtl8139_rx_one() は 1 つの Ethernet フレームを受信します．受信バッファはリングバッファとして使用されます．rtldev.rxbuf_index で受信バッファの次の読み出し位置を管理しています．受信した Ethernet フレームの情報とサイズは，先頭に付加された 4byte に格納されています．rx_status と rx_size にそれぞれを読みだしています．

その後，エラーチェックを行い，Ethernet フレームをコピーし受信キューへ入れ，次の受信バッファ読み出し位置を更新します．

```
1  void rtl8139_isr() {
2    u16 isr = in16(RTLREG(ISR));
3
4    while(rtldev.txdesc_free < 4 &&
```

```
5       (in32(RTLREG(TX_TSD[rtldev.txdesc_tail]))&(TSD_OWN|TSD_TOK))
            == (TSD_OWN|TSD_TOK)) {
6       struct pktbuf *txed_pkt = rtldev.txdesc_pkt[rtldev.
            txdesc_tail];
7       if((txed_pkt->flags & PKTBUF_SUPPRESS_FREE_AFTER_TX) == 0) {
8         pktbuf_free(txed_pkt);
9       }
10      rtldev.txdesc_pkt[rtldev.txdesc_tail] = NULL;
11      rtldev.txdesc_tail = (rtldev.txdesc_tail+1) % TXDESC_NUM;
12      rtldev.txdesc_free++;
13    }
14
15    if(isr & ISR_TOK)
16      workqueue_add(tx_wq, rtl8139_tx_all, NULL);
17
18    if(isr & ISR_ROK)
19      workqueue_add(rx_wq, rtl8139_rx_all, NULL);
20
21    if(isr & ISR_TOK)
22      out16(RTLREG(ISR), ISR_TOK);
23    if(isr & (ISR_FOVW|ISR_RXOVW|ISR_ROK))
24      out16(RTLREG(ISR), ISR_FOVW|ISR_RXOVW|ISR_ROK);
25
26    pic_sendeoi(rtldev.irq);
27  }
```

ソースコード 4.5 rtl8139_isr()

```
1   void rtl8139_rx_all(void *arg UNUSED) {
2     int rx_count = 0;
3     while(rtl8139_rx_one() == 0)
4       rx_count++;
5
6     if(rx_count > 0) {
7       thread_wakeup(&rtl8139_ops);
8       workqueue_add(ether_wq, ether_rx, (void *)DEVNO(RTL8139_MAJOR
            , RTL8139_MINOR));
9     }
10  }
11
12  int rtl8139_rx_one() {
13    /* 省略 */
```

```
14    if((in8(RTLREG(CR)) & CR_BUFE) == 1) {
15      error = -1;
16      goto err;
17    }
18
19    u32 offset = rtldev.rxbuf_index % RXBUF_SIZE;
20    u32 pkt_hdr = *((u32 *)(rtldev.rxbuf+offset));
21    u16 rx_status = pkt_hdr&0xffff;
22    u16 rx_size = pkt_hdr>>16;
23
24    if(rx_status & PKTHDR_RUNT ||
25       rx_status & PKTHDR_LONG ||
26       rx_status & PKTHDR_CRC ||
27       rx_status & PKTHDR_FAE ||
28       (rx_status & PKTHDR_ROK) == 0) {
29      puts("bad packet.");
30      error = -2;
31      goto out;
32    }
33
34    if(!queue_is_full(&rtldev.rxqueue)) {
35      char *buf = malloc(rx_size-4);
36      memcpy(buf, rtldev.rxbuf+offset+4, rx_size-4);
37      struct pktbuf *pkt = pktbuf_create(buf, rx_size-4, free, 0);
38      queue_enqueue(&pkt->link, &rtldev.rxqueue);
39    } else {
40      error = -3;
41    }
42
43  out:
44    rtldev.rxbuf_index = (offset + rx_size + 4 + 3) & ~3;
45    out16(RTLREG(CAPR), rtldev.rxbuf_index - 16);
46    /* 省略 */
47  }
```

ソースコード 4.6 パケット受信関連の関数

4.3 TCP/IP プロトコルスタックの実装

TCP/IP プロトコルスタックは，以前にマイコンで動かすために実装したもの [9] を移植しました (酷い部分が多々あり，かなり手を加えましたが).

受信パケットの処理では，ヘッダを見て適切な処理を行い上位層へ引き渡します．送信パケットは，ヘッダを付けて下位層に委ねます．最終的に Ethernet 処理部が NIC ドライバの送信関数を呼び出すことでパケットが出ていきます．

基本的にはこれだけなので，RFC を読むなりしてひたすら実装していきます．TCP の実装は特別辛かったです (lwIP なり uIP なりを移植するのが良いと思われます)．

4.4 おわりに

TCP/IP プロトコルスタックの解説に関しては力尽きたため，単に RTL8139 ドライバを書いた話になってしまいました．

最終的にはソケットを実装し，システムコールも追加してユーザランドから TCP/UDP で通信できるようになりました．そして，リモートログインをできるようにしてみました．

なお，私の作っている OS“tinyos” のソースコードは GitHub で公開しています．

https://github.com/matsud224/tinyos

参考文献

[1] tinyos https://github.com/matsud224/tinyos

[2] RTL8139(D)L Data Sheet http://www.cs.usfca.edu/~cruse/cs326f04/RTL8139D_DataSheet.pdf

[3] RTL8139(A/B) Programming guide V0.1 http://www.cs.usfca.edu/~cruse/cs326f04/RTL8139_ProgrammersGuide.pdf

[4] OSDev.org https://wiki.osdev.org/Main_Page

[5] Network buffers The BSD, Unix SVR4 and Linux approaches https://people.sissa.it/~inno/pubs/skb-reduced.pdf

[6] オペレーティングシステム II(2010 年) 筑波大学 情報科学類 講義資料 http://www.coins.tsukuba.ac.jp/~yas/coins/os2-2010/

[7] Linux カーネル 2.6 解読室 (ソフトバンククリエイティブ)

[8] Understanding TCP/IP Network Stack & Writing Network Apps https://www.cubrid.org/blog/understanding-tcp-ip-network-stack

[9] tinyip https://github.com/matsud224/tinyip

[10] 詳解 TCP/IP Vol.1 プロトコル (W・リチャード・スティーヴンス, ピアソン・エデュケーション)

5 （寄稿）ソフトウェアを作る、もしくはプログラミング以外に必要なもの

KITCC OB、元主務・元サーバ管理者　分散システム研究室　川崎　真

5.1 はじめに

我々は日々多くのソフトウェアを利用している。たとえば、この記事を執筆するのにはテキストエディタの Vim を使用している。更に、そのテキストエディタの使用のためには、デスクトップ環境・日本語入力・ファイルシステム・オペレーティングシステムなどなどが必要になる。

当然のことだが、これらのソフトウェアは誰かが（もしかしたら利用者自身かもしれないが）開発したものである。現代のコンピューティング環境において、全てのソフトウェアを 1 人で開発することは不可能である。1 人で全てを作れないということは、他人の作ったソフトウェアを使うということである。

ここで、利用者としての視点と製作者の視点から見ると、ソフトウェアを作るというのはプログラミングをするだけでは不十分であることがわかる。以下のようなことは、単にプログラミングをするだけでは解決しない問題である。

- そのソフトウェアは本当に正しく動くのか
- そのソフトウェアはどうやって手に入れればいいのか
- そのソフトウェアを使える状態にするにはどうすればいいのか
- そのソフトウェアは今後きちんとアップデートされるのだろうか
- そのソフトウェアを使うのは合法だろうか
- そのソフトウェアは悪意ある他者の攻撃に対して耐えられるだろうか

これらの問題を解決し、実際に広く多くの利用者に利用されるソフトウェアを作成するには、プログラミングが出来るだけでは不十分である。つまり、ソフトウェア開発という活動はプログラミングを含むが、プログラミングだけではなく、プログラミングが大半を

占めるわけでもない、もっと大きな活動なのだ。

この文章では、ソフトウェアを作る上で必要になってくるであろう事柄に大まかに触れていく。

5.2　そもそも何を作るのか

5.2.1　ソフトウェア開発の必要性

ソフトウェアを作るには労力がかかる。一方で、我々は無制限の労力をかけることはできないし、ソフトウェアを開発する以外にもやるべきことは沢山ある。そもそも、実用ソフトウェアというものは何らかの問題を解決するために使用するものなのだから。ソフトウェアを作ること自体が目的にはならないはずである。そこで、実際にソフトウェアを作成する必要があるのか、代替となるよりよい方法はないのか、作るとしたらどれくらいまでをソフトウェアで解決するのかを考える必要がある。

答えが自明ではない複数の選択肢から最良の選択肢を選ぶという行為を意思決定という。ソフトウェアを作成するという決断を下すのは意思決定のひとつであり、難しい問題である。おそらく、エンジニアという職業を単なる作業者から区別する理由は、このような意思決定の有無にあると思われる。ソフトウェアを作るかどうか以外にも、ソフトウェア開発の様々な場面で意思決定が登場してくる。より妥当な意思決定を行うためには、それぞれの技術的な選択肢や非技術的な選択肢に精通し、過去の経緯や現在の傾向、将来の予測などをよく知っておかなければならない。だから、必然的に優秀なエンジニアは技術に詳しい人物になる。しかし、技術に詳しいだけでは優秀なエンジニアになれるとは限らないのだ。

5.2.2　要件

要件とは、その製作物が満たすべき条件のことである。

ソフトウェアは何らかの動作をするはずなので、どのような動作をするソフトウェアを作成したいのかを考える必要がある。

簡単なソフトウェアでは、要件はそもそも単純なものなのでわざわざ明記されないことが多い。ソフトウェアの規模が大きくなってくると、要件を定義して文章にしたり、重要な要件を漏れ無く洗い出したり、あまり重要ではない要件を削除したりする必要が出てくる。

多人数でソフトウェア開発をする場合は、各人の想像がバラバラなためにうまくコミュニケーションが噛み合わないことがある。作るソフトウェアは何をどれくらい行い、何は

しないのかの大枠をはっきり定めておくことで、コミュニケーションがより円滑になる。

ソフトウェアを作成するというとき、全ての要件が設計や実装に先立って揃っていなければならないという必然性はない。ウォーターフォール型の開発プロセスでは最初に全ての要件定義を行うが、これは発注者と受注者の契約を単純にするために過ぎない。

アジャイル型の各種開発手法では、要件の定義を設計や実装と交互に行い、実際に動くソフトウェアを見ながら必要な要件を洗い出していく。最初に全てを決めて「もしかしたら必要になるかも」という要件を多く盛り込むより、実際に要る要件だけを決めて実装するほうが手間が少なくなる。このことは You ain't gonna need it (YAGNI) という原則でも示されている。本当に使うまでは機能は実装しないほうがよい。

5.3 手を動かす

5.3.1 設計

設計とは製作物の構造について意思決定を行うことである。

よくある勘違いとして、設計は考えて作るものであり手を動かす必要はないというものがある。しかし、実装をするに従って最初の設計では実装が難しかったり、根本的な欠陥があって設計を変えずに修正できないということが判明したりする。

実際には、設計は手を動かすものである。登山家は険しい山を登るときに、何回かアタックをして情報を収集したり、過去の登頂の記録を分析して情報収集したりする。ソフトウェアの設計においても、同じように試したり過去のソフトウェアの評価などを行って、設計の参考にすべきである。まず試験的に初期設計に従った最小限の試作品を作成したり、判断の上で必要な情報を得るための実験的実装を行ったりして、設計の良し悪しや設計上の判断の材料を得た上で、本実装の設計をするのだ。

設計に関するものとして、形式手法というものがある。形式手法とは、仕様を（専用の）言語で記述し、その仕様に対して数学的な証明やシミュレーションを行うことで設計の正当性を検証する手法である。

特に、分散システムではネットワーク越しの通信が多用され、しかもその通信の信頼性や遅延が不確定で、取りうるパターンが非常に多いので、形式手法を用いて不正な状態にならないかを検証することの効果が高い。実際、AWS での DinamoDB や S3 などは形式手法を用いて仕様のバグを検証している。

その他に、コンパイラの最適化なども形式手法による検証が行われることがある。

5.3.2 実装

実装は実際にプログラムを作成する段階である。

実の所、実装と設計、実装とテスト、設計とテストはそれぞれ明確に区別されるものではない。細かな部分の仕様を文章に書くよりも実際にコードにしたほうが良いし、コードを書いたらテストをすべきであるし、設計通りに動くことを確かめるには手動か自動かはともかくテストを行わなければならない。例えば、テスト駆動開発は、自動化テストを記述してから実装を行うことで、常にテストされた実装を得つつ、モジュールの使い方がテストとして書かれることで設計を見直しながら実装していく手法である。

もちろん、実装技術は重要な要素である。要求される動作や想定されるデータの性質などを考慮しながら、適切なデータ構造やアルゴリズムを選択する必要がある。つまり、実装も意思決定を含む工程なのだ。

意思決定という面では、実装に用いる言語やライブラリ、フレームワークの選択も重要な意思決定である。

2010 年代では、1 つのプログラミング言語が作業用スクリプトにも実験的試作にも Web サーバーにも OS にも向くなどということはない。それぞれの言語は、速度・機能・文法・並列処理と並行処理のサポートの程度、コンパイル速度、学習の容易さ、過去のソースコードとの互換性などに関するトレードオフにおいて、それぞれ異なる選択をしている。したがって、言語を選択する際も各言語の特性を把握して選択することで、実装が簡単になり、その言語には向かない問題を解くために多大な労力をかけなくて済むようになる。

ライブラリやフレームワークに関しては、ライブラリやフレームワークの選択をどうするかという問題と、フレームワークに乗っかるのかライブラリを使って自分で書くのかという問題の 2 つの問題がある。ライブラリやフレームワークは、使用者が多いほどバグや使用上の注意についての情報が多く手に入る。一方で、使用者が多くなるほどまで成長したライブラリやフレームワークは古いものであることが多い。最新のライブラリやフレームワークを使用する場合は、他のライブラリなどとの相互運用性やセキュリティ面での問題があることが多い。また、頻繁なバージョンアップに追従する必要もある。

5.3.3 テスト

テスト駆動開発などの実装に織り込まれたテスト手法が存在しても、独立したテストの必要性は存在する。

正しさの確認のレベルとして、verification と validation の 2 種類がある。

verification は仕様などとの合致の確認である。ユニットテストなどは verification の一種であり、仕様と実装の一致を検査する。verification に関する技法は数多くあり、自動化テスト、型検査、形式手法などによって、プログラムが要求される動きをしていることを保証することができる。しかし、verification は仕様を満たしていることを検証するのであって、それが実世界において正しいかどうかを確認できるわけではない。

validation は実世界での意味的な正しさの検証である。つまり、validation では実装だけでなく仕様までも対象にして、実際に動作が求められているものか、その動作に意味があるのかを確認する。validation の結果はその検証の文脈、つまりソフトウェアが使用される環境に依存する。例えば、米国では誕生日や登録された実名をデフォルトで公開する設計は妥当かもしれないが、日本においてはおそらく妥当ではないばかりか、セキュリティ上の問題にさえなりえるだろう。Skype には iOS の場合のみ通知に実名が表示されるという問題があったが、この問題が問題であることを認識すること自体、日本のネット利用の文化を把握していなければ不可能なものである。このことからわかるように、エンジニアは技術だけでなく、社会的環境についても広く理解している必要がある。

validation の中でも重要なものとして UI の検証がある。原理的に UI はソフトウェアの外部との接点であるため、内部で完結する verification だけではその適切さを確認できない。また、UI は利用者の属性によって適切かどうかが大きく変わる。サーバー管理者に大きな GUI で選択肢を表示するのは適さないだろうが、老人が使う場合には適切な実装となり得る。

ソフトウェアに含まれる概念の抽象性も validation に関わってくる。抽象的な構造を採用すると型システムによる検証や形式手法に向き、モジュール性も高くなる。一方で、利用者からの観点からすれば、ソフトウェアが示す概念が抽象的すぎると、利用者がソフトウェアを使って解決したい具体的な問題との間に乖離が発生してしまう。最終的な目的は利用者の問題の解決なのだから、抽象性や一般化の程度が低くても、ユーザーが理解して実際に使用できるような概念を提示するべきである。

5.3.4　内部向けドキュメント

ドキュメント化は、そのソフトウェアの内部構造の理解のために重要である。

1 つのソフトウェアが最初から最後まで 1 人の製作者によって管理され、製作者が全ての詳細まで記憶しているということはまずない。内部向けドキュメントは、仕様文書・実装・テストなどからでは理解することが不可能な情報を記録するためにある。

内部ドキュメントにおいて重要な情報として、過去の意思決定に関する情報がある。意

思決定においては、その時点での環境や人員などのリソース制約、当時必要とされていた要件などが影響してくる。そのような当時の経緯を正しく記録しておかないと、現状の設計や実装が単に誤ったものであるという理解が生まれてしまう。現状の仕様などが過去に他者との合意の結果して決められたものなら、それを勝手に変更することは契約違反になるかもしれない。また、過去には妥当だった選択が現在では妥当ではないかもしれない。どのような状況の変化があり、その結果現在できる改善は何かを把握するためには、内部向けドキュメントが過去の状況を記録していなければならない。

5.4 作るだけでなく、使えるようにする

5.4.1 ソフトウェアの提供

ソフトウェアは使用されるためにある道具である。つまり、ソフトウェアは作成されるだけでは意味がなく、使用できるようになっていなければならない。大抵の場合ソフトウェアの製作者と利用者は異なるので、利用者にソフトウェアを提供する必要がある。そして、ソフトウェアの性質によってソフトウェアの提供方法は変わってくる。

利用者の手元で動くソフトウェアは利用者の手元の機器にインストールされる必要がある。そのためには、インストーラを作成して配布したり、書庫ファイルを展開するだけで動作するようにしたりする必要がある。パッケージマネージャーのある環境に対しては、パッケージを作成して登録することで、インストールや更新を一貫した方法で行うことができる。スマートフォンのような環境では、プラットフォーム固有のアプリ配布方法がある。

Web サービスとして提供されるソフトウェアもある。サービスを提供するには、それを動作させる環境を用意しなければならない。サーバーを設置したり借りたりし、その上に作成したソフトウェアの動作環境を整え、ソフトウェアをインストールして動作させる。近年ではクラウドサービスの固有の機能を用いて、具体的にサーバーをメンテナンスすることなく、作成したソフトウェアが共有環境上で動作するサーバーレスアーキテクチャも存在する。

5.4.2 ユーザーズマニュアル

理想的にはソフトウェアの UI のみで使用法がわかることが望ましいが、実際には利用者にソフトウェアの使用法や注意点を伝えるためのユーザーズマニュアルなどが必要になることが多い。特に、設定項目が多かったり、設定内容によってセキュリティ上の問題が発生したり、機能自体がプログラマブルな場合、詳細なドキュメントが必要になる。

ユーザーズマニュアルは対象とする利用者の理解レベルに合わせて作成されなければならない。専門家が使うソフトウェアと、一般消費者が使うソフトウェアではマニュアルで説明すべきレベルや、マニュアルの誤りによって起きる問題の程度が異なる。

5.4.3 保守

ソフトウェアは最初から完全なわけではなく、バグ修正や機能追加の必要性が生じる。したがって、最初にソフトウェアを作成したあとも長期に渡ってソフトウェアを管理する必要がある。

利用者の手元で実行するソフトウェアでは、新たなバージョンを作成しリリースすることになる。手元のデータや設定を引き継ぎつつバージョンアップする方法を提供するほうがよい。

Web サービスは利用者が更新のタイミングを制御できないため、無停止でメンテナンスを行うことが多い。停止してメンテナンスを行う場合、事前の告知や利用者との折衝などが必要になってくる。

5.4.4 セキュリティ

セキュリティは重要な要件である。ソフトウェアがセキュリティ上の問題を抱えている場合、それは単にそのソフトウェアだけの問題ではなく、そのソフトウェアが動作しているシステム全体の問題になる。

ソフトウェアのセキュリティを確保するためには、プログラムに生じうる様々な脆弱性に関する知識や、システムへの攻撃や侵入がないかの監視、最新のセキュリティ関係の情報の収集などが必要になる。

5.5 終わりに

ソフトウェアを作成するにはソフトウェアに関わる様々な要素を整える必要がある。この記事ではすべての要素に触れたわけではないが、これらの要素を考慮して実際に利用できるソフトウェアを作成し公開して欲しい。

編集後記

編集担当の兼光です．今回の Lime58 号は，お楽しみいただけましたでしょうか?

実は，当初は記事の集まりが悪く，はじめての編集作業ということもあり，どうなることかと思っていました．しかし，そんな不安が続いたのも，つかの間のことでした．厳しいスケジュールだったと思いますが，Lime 制作に快く協力してくれた皆様には，心から感謝します．おかげさまで，物事は取り組めば案外どうにかなるのだ，という危うい教訓も得られました．

最後になりましたが，ここまで目を通していただき，ありがとうございました．次号も暖かく見守っていただけると嬉しく思います．

平成 30 年 11 月 08 日
編集担当　兼光 琢真

Lime Vol. 58
平成 30 年 11 月 23 日発行 第 1 刷

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