

2024 年度 機械情報夏学期 ソフトウェア第二

担当：岡田 慧 k-okada-soft2@jsk.t.u-tokyo.ac.jp

1. プログラミング環境

ソフトウェア第二ではソフトウェアの設計と実装、およびその基礎となる計算機システムについて講義する。特に、ロボットに代表される機械システム、知的ソフトウェアと連携した「大規模」ソフトウェアの開発に必要な知見と経験を得ることを目的とする。

ソフトウェアは自分自身の手を動かしてプログラミングし、自らデバッグしながら、内容を確認し理解を深めることが重要であるため、適宜実例や実習を交ぜながら進めていく予定である。

1 Ubuntu / Linux / Kernel / OS / Distribution

Ubuntu とは Linux のディストリビューション一つである。

Linux は Unix と呼ばれるオペレーティングシステム (OS) の一種である。Unix は 1969 年に AT&T¹ のベル研究所²においてケン・トンプソン、デニス・リッチャー³ によって開発された。Unix は高級言語の一つである C 言語で記述されていることもあり、様々なハードウェアに移植されるとともに、大学や研究機関で広く利用されるようになったため、直接的な派生物⁴ や、Unix 互換の OS が多数開発された。<https://www.levenez.com/unix/unix.pdf> を参照すると、Android も iOS も MacOS も Unix 派生物あるいは互換の一つである OS であることがわかる。

厳密には Linux はカーネルだけを指す。カーネルとは OS の中核の機能を指し、メモリ、CPU、(キーボード、ストレージ、USB などの I/O、モニタ) などの入出力の各種リソースを管理する。カーネルを狭義の OS と呼ぶこともある（図 1,2）。しかし、一般に Linux と呼ぶときはウインドウシステム、開発言語やライブラリなどのツール、ブラウザや事務処理のアプリケーション、Web やデータベースのサービスなど様々な機能を含んだ広義の OS を指すことが多い。また、カーネル以外のこれらのツールの大部分は GNU プロジェクト⁵ で開発されたものが多く、Linux が OS として機能するために GNU プロジェクトで開発物が必要不可欠との立場から、GNU/Linux と呼ぶのが正しい、という意見もある (https://en.wikipedia.org/wiki/GNU/Linux_naming_controversy)。

Linux におけるディストリビューションとは、カーネル、ライブラリからアプリケーションのソフトウェア群を一つのパッケージとしたものであり、そのパッケージに含まれているソフトウェア

¹ アメリカ最大の電話会社。1877 年にグラハム・ベルが興したベル電話会社が前身。1974 年に司法省による反トラスト法訴訟を経て 1984 年に地域ベル電話会社に分割される。2005 年に地域ベル会社により買収される。子会社にアメリカ 3 大携帯会社の一つ AT&T Mobility がある。残りの 2 つは T-Mobile、Verizon Wireless

² Unix、C/C++ 言語の開発で計算機科学分野において多大な貢献を果たした研究所。一般には基礎科学、物性物理学、電子工学などの研究で有名。7 つのノーベル物理学賞を受賞している。技術者倫理で取り上げている「論文捏造（著者：村松秀）」のヘンドリック・シェーンによる常温超電導研究もベル研で実施されていた。

³ 「プログラミング言語 C」（カーニハム＆リッチャー、通称 K&R）の著者の一人でもある。

⁴ 派生物とは、ソースコードそのものの変更物だけでなく、ソフトウェアをリンクしたものや、ソフトウェアの依存性を持つ成果物のことをいう。

⁵ <https://ja.wikipedia.org/wiki/GNU>

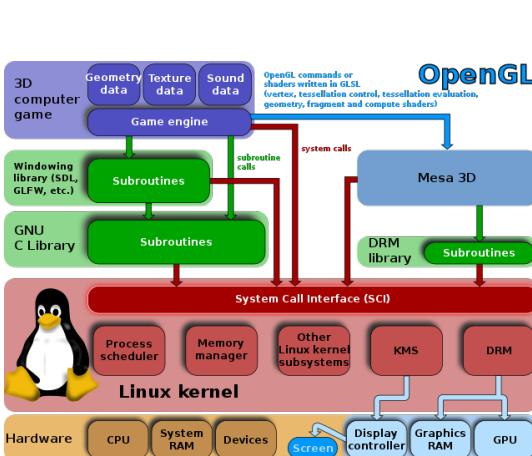


図 1: The connection between the Linux kernel and OpenGL-based video games
https://en.wikipedia.org/wiki/Linux_kernel

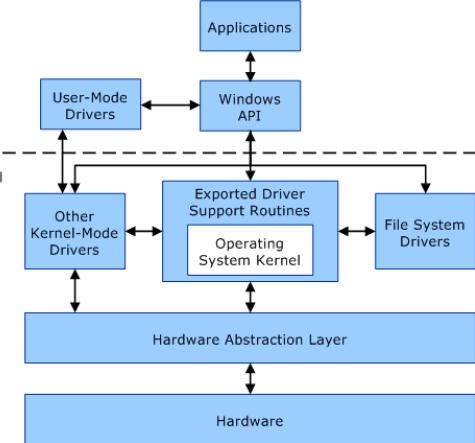


図 2: Windows OS におけるユーザー モードとカーネル モードより <https://docs.microsoft.com/ja-jp/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode>

(あるいはそのソフトウェアのバージョン)であればそれぞれ衝突することなく、共存することが期待できる。さらに、これらのソフトウェア群をインストールするシステムとサービスを提供し、適宜アップデートできるようになっている。

逆に言えば、あるディストリビューションを用いた場合、それ以外の方法⁶でインストールすることは極めて危険であり、ディストリビューションの提供する方法で利用することが強く求められている。

Ubuntu は現時点でもっとも使いやすいディストリビューションであろう。他のディストリビューションは https://en.wikipedia.org/wiki/Linux_distribution を参照すること。

各ディストリビューションは定期的に最新バージョンをリリースしており、Ubuntu の場合のリリーススケジュールは https://en.wikipedia.org/wiki/List_of_Ubuntu_releases#Version_timeline で確認できる。

計算機に Ubuntu をインストールする方法は 3 つある。

1. ハードディスク 1 台全ての領域に Ubuntu をインストールする。ハードディスクに存在した情報は全て消去される。
2. ハードディスクに既に存在した OS の領域を変更し、空いた領域に Ubuntu をインストールする。計算機の起動時にはどちらの OS を立ち上げるかを選択する（これをデュアルブートという）。
3. 仮想マシンを使って複数の OS をインストールする。

情報理工 PC はデュアルブートにより Windows と Ubuntu の両方の環境を実現している。このやり方は、<https://wiki.ubuntulinux.jp/UbuntuTips/Install/InstallDualBoot> や <https://www.pc-koubou.jp/magazine/35542> に情報がある。ただし前者は情報が古く、後者は公式の

⁶ 例えば./configure; make; sudo make install などの指示は極めて危険であることを理解するべきである。

文章ではなく正確性にかけるので、それぞれ注意が必要である。しかも、失敗すると既に存在する OS の領域が消えてしまうので細心の注意が必要である。

本節の以下は、ソフトウェア第二の初回は各自の計算機に講義に必要なプログラミング環境を構築する方法を説明する。情報理工 PC を貸与されていない者や、私物の計算機上に Ubuntu 環境を構築したい者は参考にすると良い。

また、情報理工 PC を貸与されている者も、仮想マシンの概念についてはしっかり理解しておいてほしい。

2 仮想マシン

仮想マシンとは計算機システムを仮想化 (virtualization)/エミュレーション (emulation) することであり、ハードウェアまたはソフトウェアで計算機や OS を動かすことを可能にする。

仮想マシンを実現するための制御プログラムを仮想マシンモニタ (VMM: Virtual Machine Monitor) や、ハイパーバイザ (Hypervisor) と呼ぶ。これは、OS またはカーネルをスーパーバイザ (Supervisor) と呼ぶことに対して、super-なものを、さらに上位から制御するものとして hyper-な層として定義されている。ハイパーバイザ自体が走っている計算機を「ホスト」(または「ホスト計算機」、「ホスト OS」等) と呼び、ハイパーバイザ上で走る仮想的な計算機を「ゲスト」(または「ゲスト計算機」、「ゲスト OS」等) と呼ぶ。

ハイパーバイザを用いて OS 自体を仮想化する方法には、図 3 に示すように、大きく既存の OS をそのまま実行できるこれを完全仮想化 (Full Virtualization) と、動作させる OS に変更を加えて実行する準仮想化 (Paravirtualization) が存在する。さらにハイパーバイザを利用せずに OS のカーネル自体は共有しながら異なるユーザ環境を提供する OS レベル仮想化技術も近年盛んに利用されつつある。

完全仮想化と準仮想化ではゲスト OS が走るレイヤーが異なり、準仮想化環境ではカーネルモードで走ることにより、例えば図 6 のように複数のゲスト OS 間でのハードウェアデバイスの共有を可能にしている。

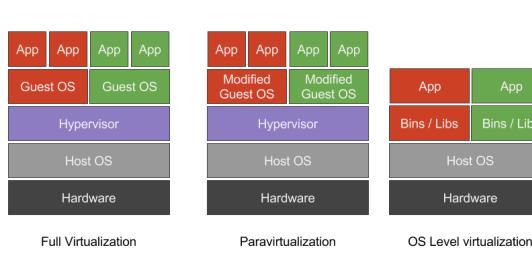


図 3: 完全仮想化、準仮想化、OS レベル仮想化
Credit <https://blog.zwindler.fr/2016/08/25/when-should-we-have-containers/>

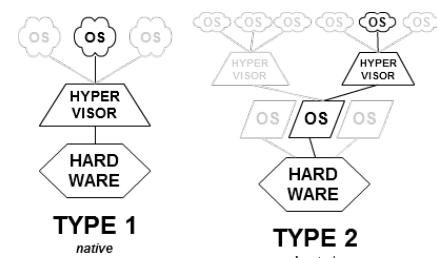
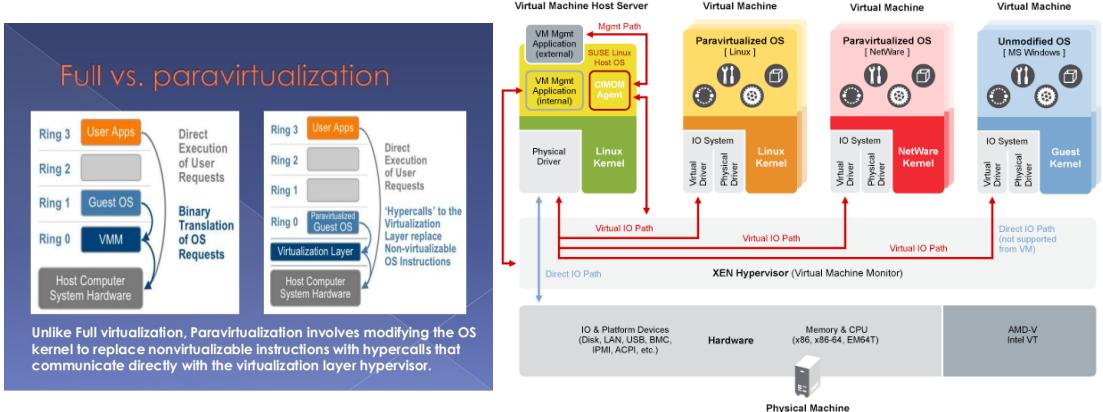
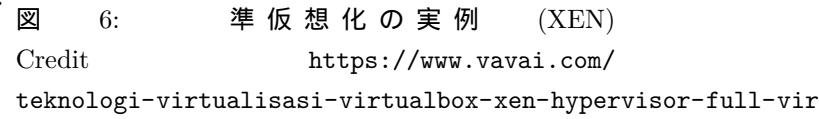


図 4: Type 1 and Type 2 hypervisors
Credit <https://en.wikipedia.org/wiki/Hypervisor>

ハイパーバイザ自体は図 4 に示すの Type1 (Native, Bare-Metal)、Type2 (Hosted) の 2 種類がある。Type1 ではハイパーバイザ自体が直接ホストのハードウェア上で動くものであり、その上でゲスト OS を稼働させる。これは狭義のハイパーバイザとも言われる。

図 5: 完全仮想化と準仮想化 Credit <https://www.slideshare.net/GauravSuri1/virtualization-and-hypervisors>図 6: 準仮想化の実例 (XEN)
Credit <https://www.vavai.com/teknologi-virtualisasi-virtualbox-xen-hypervisor-full-vir>

Type2 では、通常の OS 上で動くハイパーバイザである。ゲスト OS (とハイパーバイザ自体) はホスト OS 上の一つのプロセスとして実現される。

OS レベル仮想化技術ではゲストとしてカーネル自体を動かすのではなく、カーネル上で稼働するソフトウェアや、そのソフトウェアが利用するライブラリやツールを隔離された環境で実行することで、ホスト OS 上で複数のゲスト OS (正確にはカーネル以外の OS 機能) を利用可能にするものである。最近は docker と呼ばれるサービスが広く利用されている。

なお、Ubuntu 上で Docker 環境を試したい者は次のコマンドを試してみるとこと。ここでは Ubuntu の 23.04 を実行することができる。

```
$ docker run --rm -ti ubuntu:23.04 /bin/bash .
```

利用可能なイメージのリストは <http://hub.docker.com/search> で調べることができる。

また、QEMU というエミュレータ（ハイパーバイザ）を通じて、小型 PC や携帯電話で利用されている ARM CPU 上で動く Ubuntu を実行することもできる。

```
$ sudo apt install qemu-user-static .
$ docker run --rm -ti arm64v8/ubuntu:22.04 bin/bash .
```

Windows は近年 WSL(Windows Subsystem for Linux) と呼ばれる仮想化サポートを強力にすすめている。これまでに OS レベルの仮想化を用いた WSL 1 から、ベアメタル型の準仮想化を用いた WSL 2 へと発展してきている⁷。

Docker Desktop on Windows では図 8 の構成となり、ハイパーバイザ上の Linux カーネル上で OS 仮想化されたゲスト環境を駆動している。

3 仮想マシン (Docker) を使った Ubuntu 環境の構築

3.1 Docker Desktop for Windows を用いた Ubuntu 仮想マシン環境の構築

Docker Desktop for Windows を用いる方法を <https://docs.docker.jp/docker-for-windows/install.html> に沿って説明する。

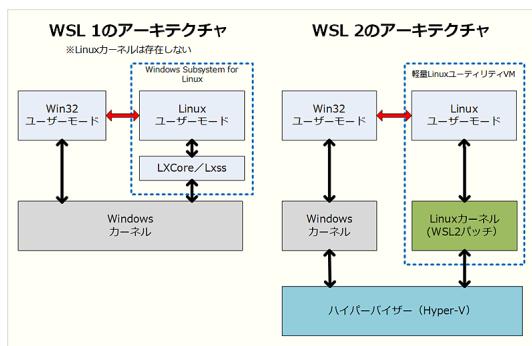


図 7: WSL と WSL2 のアーキテクチャの
違い Credit <https://www.atmarkit.co.jp/ait/articles/1906/14/news019.html>

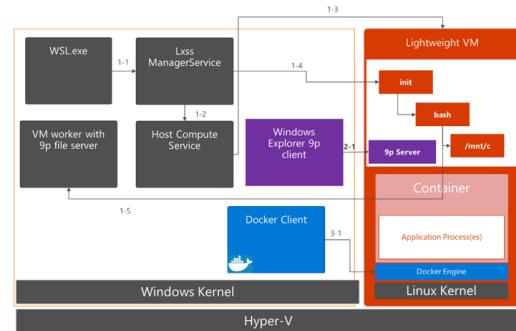


図 8: Docker for Windows の構成
<https://forums.docker.com/t/is-there-a-pictorial-diagram-of-how-wsl-2-docker-docker-desktop/100071>

3.1.1 システム要件の確認

Docker Desktop for Windows を利用するには Version 1903 以上、または Build 18362 以上の Windows 10 である必要がある。

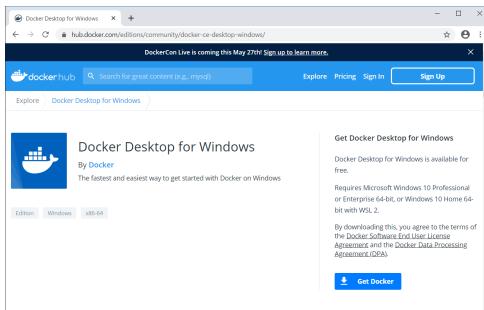
Windows のバージョンの確認は以下で行うことができる。

<https://docs.microsoft.com/en-us/windows/wsl/troubleshooting#check-your-build-number>

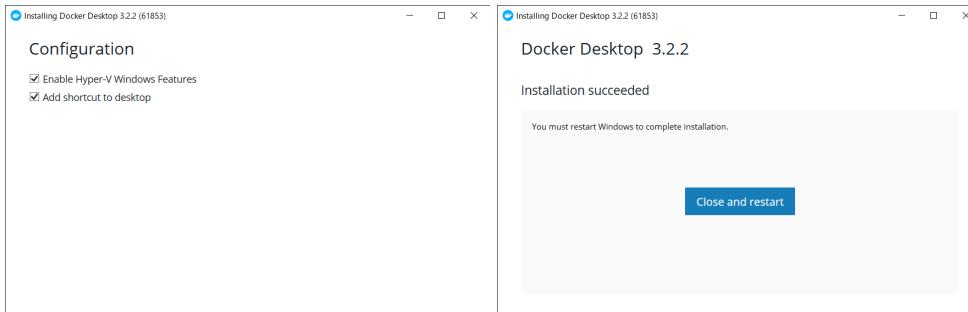
3.1.2 Docker Desktop for Windows のダウンロードとインストール

Docker のインストーラは以下からダウンロードする。

<https://hub.docker.com/editions/community/docker-ce-desktop-windows/>



インストーラをダウンロードしたら実行しインストールする。

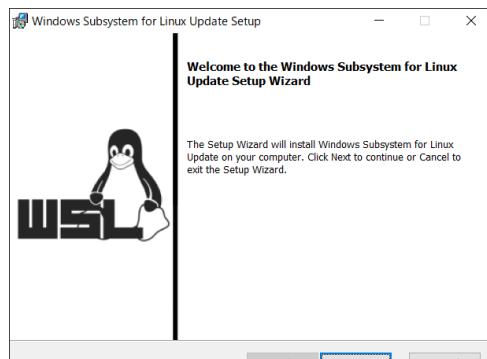


設定時のチェックはすべてデフォルトのままで良い。このとき Hyper-V を有効になっていることを確認することができる。

最後に、WSL2 のインストールが必要であるとのメッセージが出てくる。



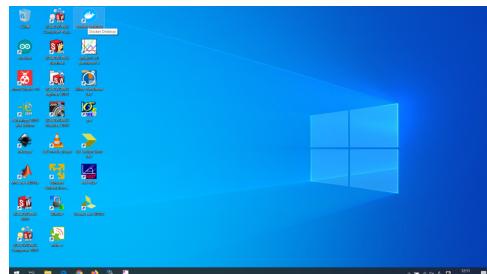
指示に従い <https://aka.ms/wsl2kernel> から「x64 マシン用 WSL2 Linux カーネル更新プログラム パッケージ」をダウンロードしインストールする。



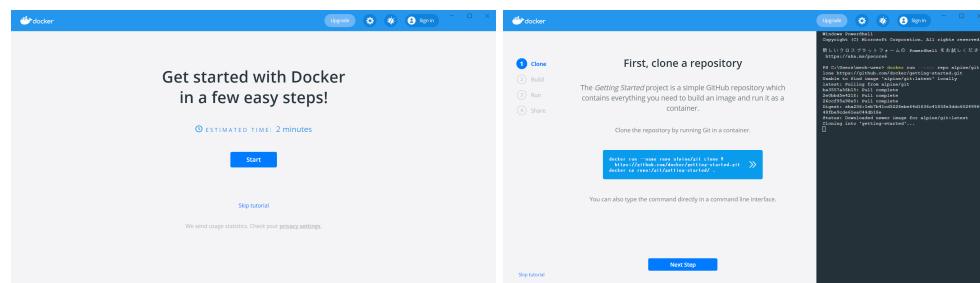
以上のインストールが終わったら再起動すること。

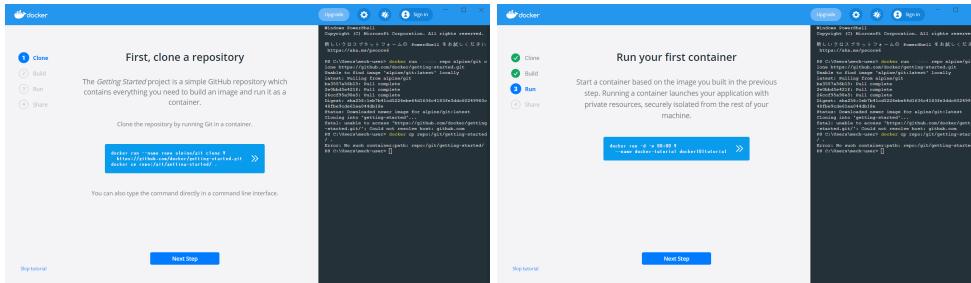
3.1.3 Docker Desktop for Windows の開始

デスクトップから Docker for Windows を開始する。



最初に開始すると以下のようにチュートリアルが実行するが、このまま実行してもエラーになるようだ。

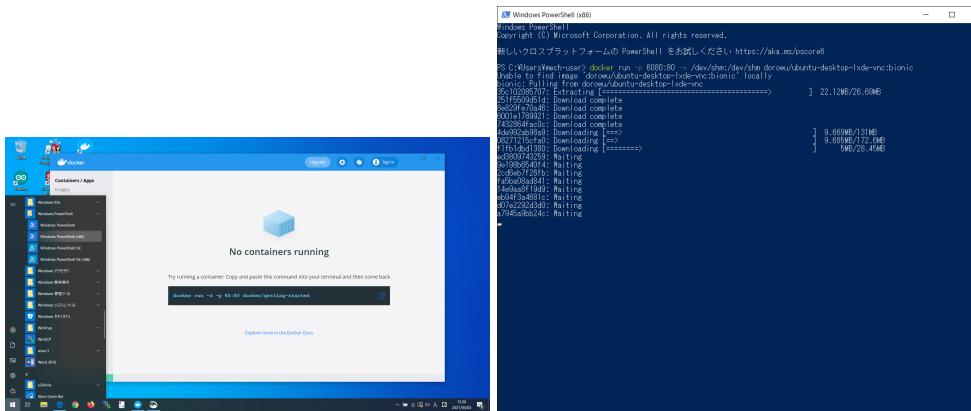




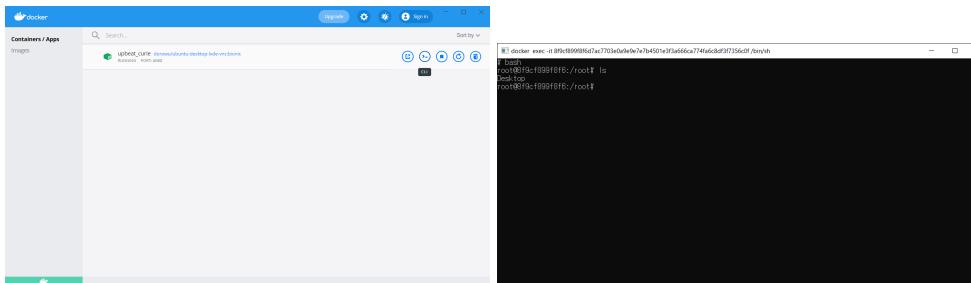
ここでは、PowerShell から Docker コンテナを実行する。PowerShell は左下の Windows アイコンから「Windows Power Shell」を探してほしい。

さらに Docker コンテナを実行するには PowerShell 上で

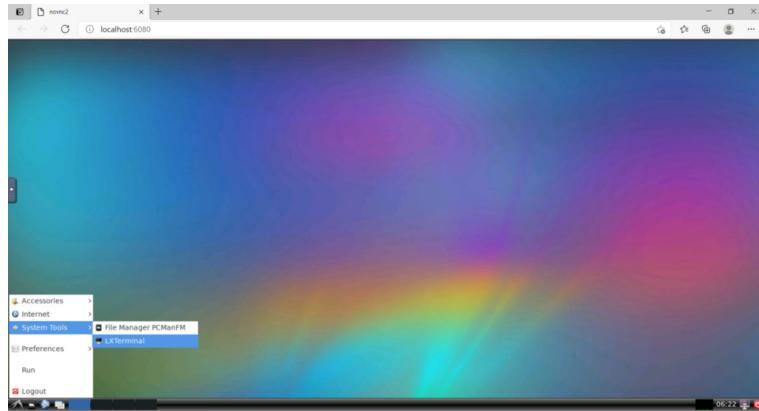
```
docker run -p 6080:80 -v /dev/shm:/dev/shm dorowu/ubuntu-desktop-lxde-vnc:bionic
を打ち込む。
```



Docker コンテナが実行されたら、Docker for Windows のアプリケーションが以下のようになる。このとき CLI アイコンを実行すると Linux のターミナルが実行できる。



また、<http://localhost:6080> にアクセスするとデスクトップが表示される。



3.2 Docker Desktop for Mac を用いた Ubuntu 仮想マシン環境の構築

Docker Desktop for Mac を用いる方法を <https://docs.docker.com/desktop/mac/install/> に沿って説明する .

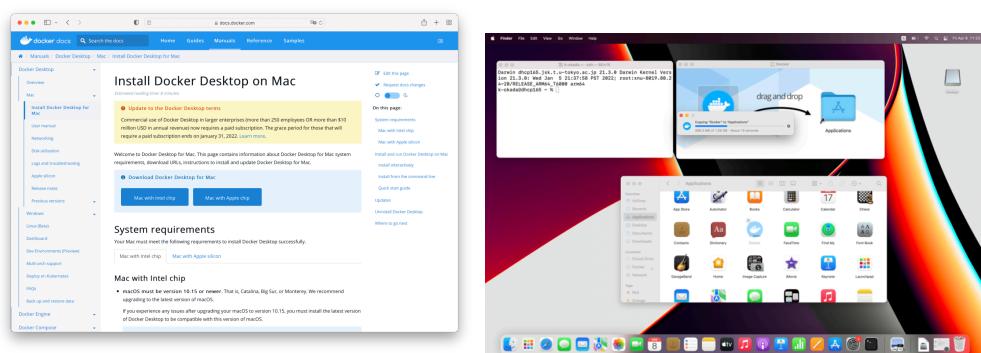
3.2.1 システム要件の確認

Docker Desktop for Mac を利用するには macOS Version が 10.15 以上である必要がある .

3.2.2 Docker Desktop for Mac のダウンロードとインストール

Docker のインストーラは Intel 用 , Apple Chip(ARM) 用の 2 つがある . 各自のマシンに応じてダウンロードする .

インストーラをダウンロードしたら実行しインストールする .

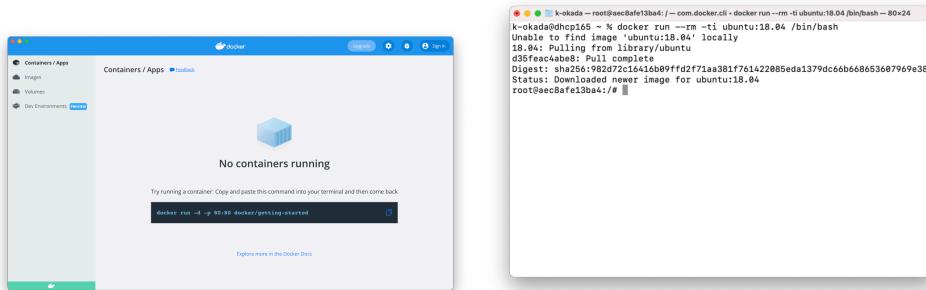


3.2.3 Docker Desktop for Mac の開始

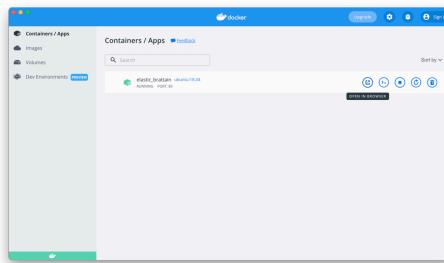
Docker Desktop for Mac を開始する .

最初に開始すると以下のように Mac の Terminal から docker コマンドを実行するよう指示がされる .

これに従っても良いし ,Mac の Terminal から \$ docker run --rm -ti ubuntu:18.04 /bin/bash . として Ubuntu と同様に Docker を実行することが出来るようになる .



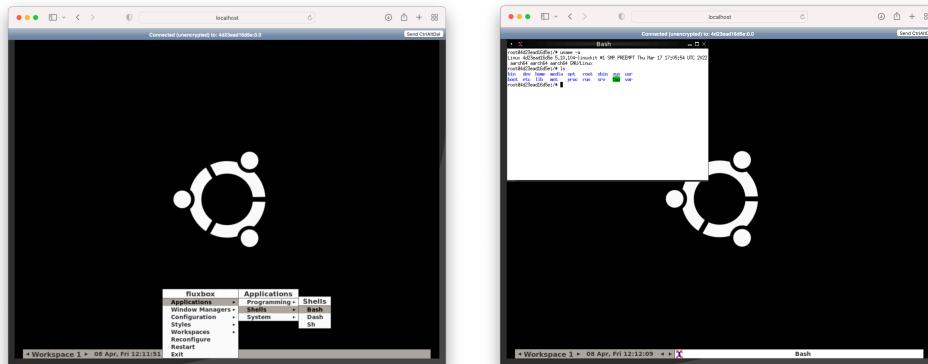
Terminal から docker を起動すると , Docker Desktop for Mac から以下のように起動しているコンテナを確認することが出来る .



また ,新たに Mac の Terminal から docker run --rm -p 8080:8080 -ti ubuntu:18.04 /bin/bash+
と-p 8080:8080 を付けて再度 docker を立ち上げ ,

```
apt update
apt install -y fluxbox novnc x11vnc xvfb
export DISPLAY=:0.0
Xvfb :0 -screen 0 1024x768x24 -listen tcp -ac &
x11vnc -forever -shared &
websockify --web /usr/share/novnc 8080 localhost:5900 &
fluxbox
```

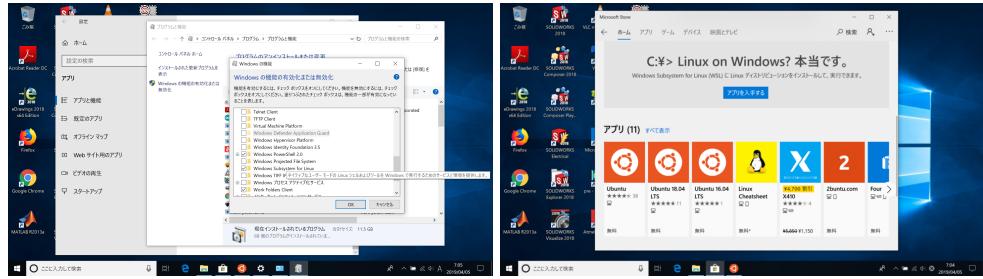
と順番にコマンドを実行し , http://localhost:8080/vnc_auto.html にアクセスするとデスクトップが表示される .



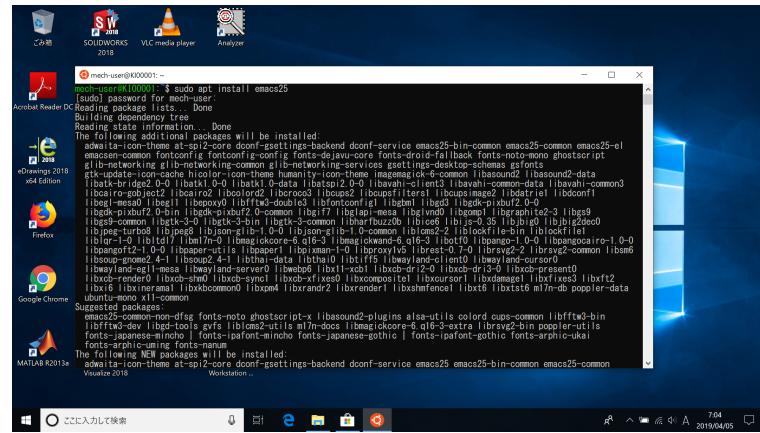
4 仮想マシン (WSL) を使った Ubuntu 環境の構築

Windows 上で Linux(Ubuntu) の機能が提供された Windows Subsystem for Linux が利用できるようになっている。

スタートボタンから「アプリと機能」を選択し一番下の「プログラムと機能」をクリックする。次に「Windows 機能の有効化または無効化」から「Windows Subsystem for Linux」にチェックを入れ、画面に従って再起動する。



次に Microsoft Store から「Windows Subsystems for Linux」と「Ubuntu 20.04 LTS」をインストールする。最後にスタートから Ubuntu を起動し、ユーザ名パスワードを設定して終了となる。



5 Linux の使い方

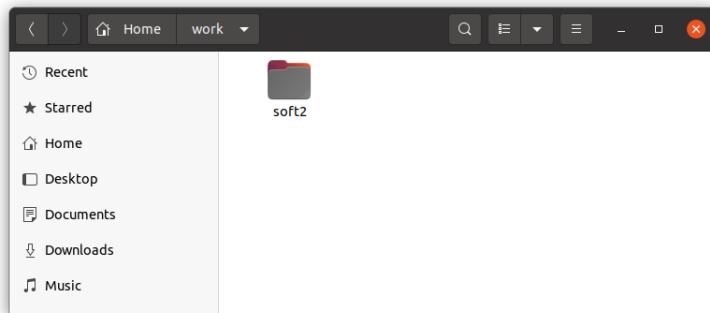
以下で Linux の基本的な使い方を示す。Linux は Unix の互換 OS であることから、これは Unix の基本的な使い方になり、マウスを多様する GUI (Graphical User Interface) ではなく、ターミナルに一行づつコマンドを入れる CLI(Command Line Interface) の基礎となる。

通常 Linux 上で行うすべての GUI 操作は CLI 経由で実現できる。キーボードから手を話すことなく指先だけですべての作業が実現できると大変効率が良いので、ぜひ慣れてほしい。

5.1 shell の立ち上げ方

パネルに置いたランチャや左下の「アプリケーション」から「端末」あるいは「Terminal」選択、あるいは [Ctrl]+[Alt]+[T] で端末を起動する。これで shell が立ち上がる

例えば、以下の図にあるように、ホームディレクトリ以下に work というディレクトリを作り、その下にさらに soft2 というファイルを作成したいしよう。



これは、shell を用いて以下のように実現できる。

```
mech-user@KI00000:~$ mkdir work
mech-user@KI00000:~$ cd work
mech-user@KI00000:~/work$ pwd
/home/mech-user/work
mech-user@KI00000:~/work$ mkdir soft2
mech-user@KI00000:~/work$ ls
soft2
mech-user@KI00000:~/work$ ls -al
合計 12
drwxr-xr-x  3 mech-user mech-user 4096 2010-04-03 18:02 .
drwxr-xr-x 27 mech-user mech-user 4096 2010-04-03 18:02 ..
drwxr-xr-x  3 mech-user mech-user 4096 2010-04-03 18:02 soft2
mech-user@KI00000:~/work$ cd soft2
mech-user@KI00000:~/work/soft2$ ls
mech-user@KI00000:~/work/soft2$ pwd
/home/mech-user/work/soft2
mech-user@KI00000:~/work/soft2$ cd ..
mech-user@KI00000:~/work$ pwd
/home/mech-user/work
```

- **mkdir** - Make directories: 現在、ホームディレクトリ()にあり、work ディレクトリを作成
- **cd** - Change the shell working directory: work ディレクトリに移動

- pwd - print name of current/working directory: 現在のディレクトリを表示
- ls - list directory contents: ディレクトリの内容を表示
 - a - do not ignore entries starting with .. : .. は現在のディレクトリ, .. は一つ上のディレクトリ
 - l - use a long listing format : ファイルの所有者, 作成時期などを表示

実際には ,

```
mech-user@KI00000:~$ mkdir -p work/soft2
mech-user@KI00000:~$ cd work/soft2
mech-user@KI00000:~/work/soft2$ xdg-open .
```

と , UNIX コマンドを駆使してディレクトリの作成 , 移動 , フォルダの表示が可能である .
 (岡田自身は基本的な Unix コマンドは京都大学人文科学研究所の安岡孝一先生による <http://kanji.zinbun.kyoto-u.ac.jp/~yasuoka/publications/dareUni/> にあるドキュメントの Unix 講座や /bin/sh 講座を見て勉強していた .

大変古いので仕様が変わっている部分もある例えは , Unix 講座の第 2 回に出てくるメールコマンドなどは設定していないためつかえないであろう . 第 3 回や第 4 回を読むと良い .) .
 ~ はチルダとよみ , ホームディレクトリ (/home/mech-user) を意味する . つまり ,

```
mech-user@KI00000:~/work$ cd ~
mech-user@KI00000:~$ 
mech-user@KI00000:~$ pwd
/home/mech-user
```

とするとホームディレクトリ (/home/mech-user) に戻ることができる .

5.2 Shell コマンド

Unix のコマンドはシェル (shell) と呼ばれるプログラムを介して実行される . cd,ls,mkdir も shell コマンドの一つである . shell には多くの種類があるが , ここではデフォルトとなっている bash のコマンドについて説明する

5.2.1 補完

コマンドプロンプトで Tab キーを押すことでコマンド名やファイル名を補完することができる .

```
$ ls
Desktop  Documents  Downloads  Pictures  Public
```

となっている時に Tab キーを押すことで

```
$ ls De[Tab]
$ ls Desktop
```

と補完できる . [Tab] は実際に打ち込むのではなく , Tab キーを押す , という意味である .
 また候補が複数ある場合は Tab キーを 2 回打ち込むことで候補が表示される .

```
$ ls D[Tab] [Tab]
Desktop/ Documents/ Downloads/
```

5.2.2 リダイレクト・パイプ

リダイレクトとは、標準入力、標準出力、標準エラー出力⁷出力の入出力先をデフォルトの端末（キー ボードとモニタ）からファイルなどに変更することが出来る機能である。

パイプを使うことで他の標準出力を標準入力に繋げることが出来る。コマンドラインの中では | を用いて 2 つのコマンドをつなげる。すなわち、[CMD1] | [CMD2] のようにして、前のプログラム ([CMD1]) の標準出力を、後ろのプログラム ([CMD2]) の標準入力につなげることができる。このように、一つのことを上手くやる小さいプログラムを作成し、これをパイプで組み合わせて様々な複雑な処理を実現出来るのが、UNIX の特徴であり、かつ UNIX 哲学 (https://en.wikipedia.org/wiki/Unix_philosophy) と呼ばれるものであるので是非覚えておこう。

リダイレクト・パイプを使う場面でよく使われるコマンドに以下がある。

- grep PATTERNS [FILE...]: FILE の中で PATTERNS にマッチする行だけを表示する
- wc [FILE]...: FILE の中の行数、ワード数、バイト数を表示する。
- sort [FILE]...: FILE の各行をソートする
- more [FILE]...: FILE を表示する。スペースキーで次のページを表示、b キーで前のページに戻る
- less [FILE]...: more の機能表示バージョン

また、ps -aux とすると、現在計算機上で走っているすべてのプロセス⁸を表示することができる。例えば、ユーザ名毎に整理して表示するためには、

```
$ ps -aux | sort
```

または、

```
$ ps -aux | sort | less
```

として表示できる。また、mech-user が走らせているプロセスの一覧は

```
$ ps -aux | grep mech-
```

として表示できる。また、そのプロセスの数を調べるには、

```
$ ps -aux | grep mech- | wc
```

⁷fprintf(stdout, "HelloWorld")としたときの stdout が標準出力への表示という意味である。この部分を、stderr とした場合は標準エラー出力となる。また、fscanf(stdin, "%d", &d) の stdin が標準入力を意味している

⁸プロセスについては「10. オペレーティングシステム」で紹介する。

5.2.3 制御構造

bashにおいても if 文や for 文が利用できる .

```
$ if [ -e ~/Documents ]; then ls -al ~/Documents; fi
$ for dir in ~/D*; do ls -ald $dir; done
```

5.2.4 履歴

コマンドの履歴は記憶されている . 同じコマンドを何回も打ち込む場合は履歴を使う事が出来る .

C-p : 一つ前のコマンドを表示
C-n : 一つ後のコマンドを表示
C-r : 後方へのインクリメンタルサーチ , Enter で見つかった候補を実行 , C-g で検索の中 断

後述する emacs キーバインドとの類似性に注意しよう .

5.2.5 フィアル名展開

文字列中に*,?,[が含まれていた場合ファイル名展開が行われる .

* : 0 文字以上の任意の文字列にマッチ
? : 1 文字以上の任意の文字列にマッチ
[..] : ブラケットで含まれている文字の 1 文字にマッチ .

```
$ ls ~/D*
```

5.2.6 ジョブ

各プログラム（プロセス）群はジョブと呼ばれる単位で管理される .

例えば\$ emacs とすると , emacs を停止するまではこのターミナルでは作業ができないが , \$ emacs & とすると emacs を起動した後もターミナルで引き続き他の作業が可能になる . これは emacs ジョブをバックグラウンドで動かしているからである .

現在のジョブの状態は jobs というコマンドで確認できる . fg コマンドでバックグラウンドで動 い い て いる ジョブを フォアグラウンドに持ってくる事が出来る . C-z は フォアグラウンドで動 い て いる ジョブを 停止させる . これを バックグラウンドで動かすには bg コマンドを利用する . フォア グラウンドで走っているジョブを強制終了するには C-c を押す .

6 Ubuntu の使い方

Ubuntu では deb パッケージという方式を用いて各種ソフトウェアやツールを管理、配布している。ユーザはこれを用いて簡単にソフトウェアをインストールする事ができるが、まずは初めて OS を立ち上げたあとに行ったら良い作業をいくつか紹介する。

情報理工 PC を有しているものは「Ubuntu での初期設定」は飛ばし、その次の節である「ソフトウェアのアップデート」から読んでほしい。

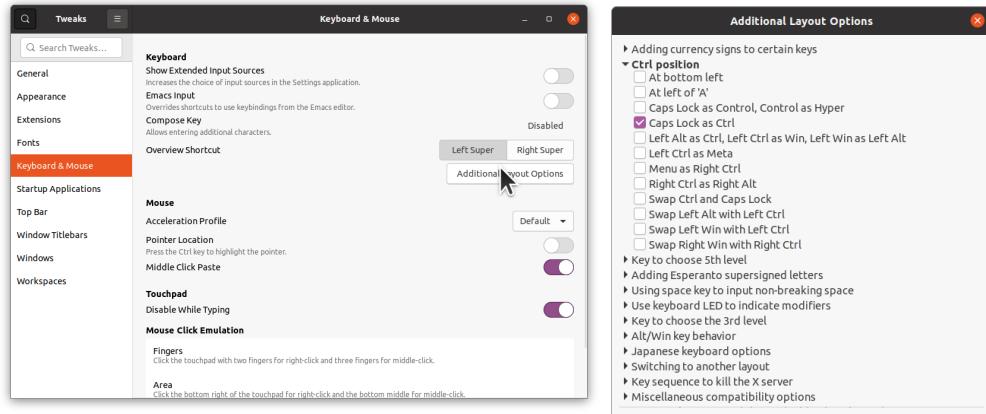
6.1 Ubuntu での初期設定

6.1.1 Ctrl キーと CapsLock キーの交換

プログラム開発の場面では Ctrl キーを良く使う。小指が届きやすい CapsLock キーを Ctrl として振る舞うように設定すると柄しやすい。

Tweaks (gnome-tweaks) を起動し、Keyboard & Mouse - Additional Layout Options Caps Lock behavior - Caps Lock as Ctrl を選択する

```
$ gnome-tweaks
```



6.1.2 端末のランチャをパネルに追加

最初に「アプリケーション」「アクセサリ」に順にカーソルを合わせ、そこで出てきた「端末」を右クリックして「このランチャをパネルに追加」を選択し、パネルに端末のランチャを置くと使いやすい。



6.2 ソフトウェアのアップデート

ここではあとで紹介する emacs というエディタをインストールする場合を説明しよう。まず、最初に

```
$ sudo apt update
```

として、ソフトウェアをアップデートする。この際パスワードを聞かれるので、上記のユーザーアカウントの設定で入力したパスワードを入力する。sudo はルートアカウント（管理者権限）でコマンドを実行する。管理者権限ではシステムを破壊することが可能なので実行時には細心の注意を望むこと。

また、

```
$ sudo apt dist-upgrade
```

とすると、Ubuntu の最新バージョンにアップグレードが可能である。

ただし新しいリリース（22.04 や 23.04）へのアップグレードは行わないこと。講義は全員が 20.04 を用いていることを前提にしている。

6.3 パッケージを探す

未インストールのソフトウェアを利用しようとすると以下の様なメッセージができる場合がある。

```
$ xemacs  
Command 'xemacs' not found, but can be installed with:  
sudo apt install xemacs21
```

これは xemacs というコマンドがないのでインストールする必要があるということを伝えている。インストールする際には、インストールするパッケージ名を探す必要がある。

また、別の方法として、

```
$ apt search xemacs
```

とすると xemacs というキーワードに関係のあるパッケージを探すことができる。

6.4 パッケージをインストールする

パッケージ名が分かれば

```
$ sudo apt install xemacs
```

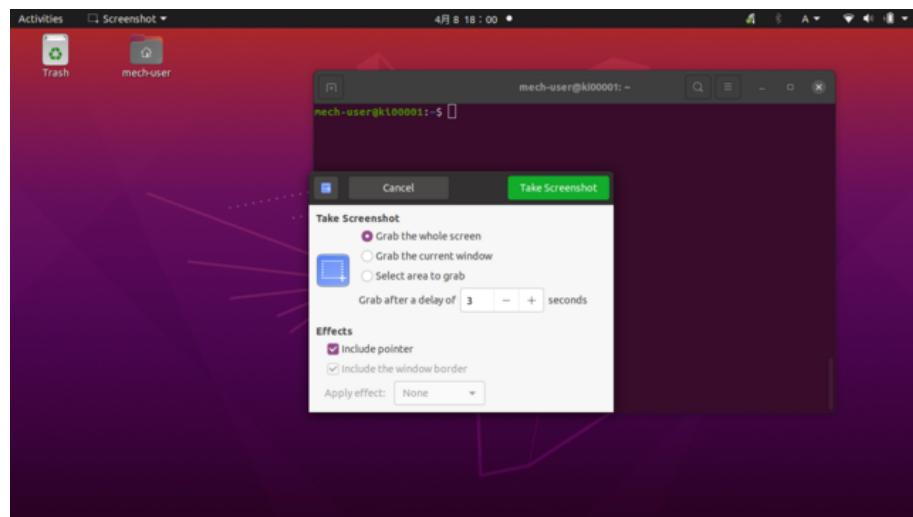
とすると、

```
パッケージリストを読み込んでいます... 完了
依存関係ツリーを作成しています
状態情報を読み取っています... 完了
以下のパッケージが新たにインストールされます:
  xemacs
アップグレード: 0 個、新規インストール: 1 個、削除: 0 個、保留: 477 個。
3,369 kB のアーカイブを取得する必要があります。
この操作後に追加で 17.8 MB のディスク容量が消費されます。
続行しますか? [Y/n]
```

と表示される。ここで“Y”と打つか、あるいはリターンを押せばインストールが続行され xemacs が利用可能になる。

6.5 画面キャプチャの仕方

キーボードの PrtSc キーを押すと全画面キャプチャが可能になる。また Alt + PrtSc キーを同時に押すと、アクティブな（前面の）画面をキャプチャできる。あるいは「スクリーンショット」というプログラムを探すかコマンドラインから gnome-screenshot --interactive を実行するキャプチャした画像は適切な名前をつけて保存しよう。



7 エディタの使い方

7.1 Emacs エディタの使い方

エディタは Emacs を使おう⁹。

```
$ emacs -nw
```

として立ち上がる。

オリジナルは 1975 年にガイ・スティールとリチャード・ストールマンが開発した TECO エディタのエディタ・マクロ (Editor MACroS) であり、現在の Emacs は 1984 年にリチャード・ストールマンが開発を始めた GNU Emacs が現在も使われている。

ちなみに、ガイ・スティールは Scheme 言語、Java 言語の開発にも携わった計算機科学者であり、C 言語、FORTRAN, CommonLisp の ANSI 標準化にも参加した。

一方、リチャード・ストールマンは GNU プロジェクトの主催者であり、様々な GNU フリーソフトウェアを開発してきている。である。GNU のソフトウェア全てインターネット上のサーバ (<ftp://ftp.gnu.org/gnu/>) で公開されダウンロードできる。

C-z とすると、shell に戻る。C-z とは Ctrl キーを押してから z を押す、という操作を意味している。これは、プロセスをバックグラウンドで実行している状態である。ここで、fg と打ち込めば、また Emacs に戻ることができる。

7.1.1 ~/.emacs.d/init.el

~/.emacs.d/init.el は、Emacs の起動時読み込まれるファイルである。Emacs から ~/.emacs.d/init.el を開いて、どのような設定があるのかを眺めてみよう。Emacs が読み込むファイルは、EmacsLisp という Lisp 言語で記述されている。

Emacs が立ち上がったら C-x C-f と打ち込み (Ctrl キーを押しながら x を押し、次に、Ctrl キーを押しながら f を押す。Ctrl キーは押しっぱなしでもよい)、リターンを押す。ミニバッファ (画面下側の 1 行の部分) に Find file: ~/ と表示されるので、ここで、.emacs.d/init.el と打ち込みリターンを押す。これで ~/.emacs.d/init.el が表示される。もし、途中で操作が分からなくなったら C-g と入力するとよい。入力がキャンセルされミニバッファが初期状態に戻る。

これができたら、~/.emacs.d/init.el に、

```
;;; M-x g goto-line
;;; デバッグをする際にこれがあると便利である
(global-set-key "\M-g" 'goto-line)
```

という行を追加しよう。そして、Emacs を起動しなおして、~/.emacs.d/init.el を開き、M-g (ESC キーを押して離して g キーを押す) と入力し、行番号を指定すると、指定した行へジャンプすることを確認しよう。

なお、~ が含まれている PDF をコピー & ペーストすると違う文字になることがあるので、ここは必ず自ら打ち込むこと。~ の使いに不安であれば、/home/mecch-user/.emacs.d/init.el としても同じである。

ついでに、

⁹<https://ja.wikipedia.org/wiki/エディタ戦争>

```
(global-set-key "\C-h" 'backward-delete-char)
```

という行を書いておくと C-h と入力するとデリートキーを入力するのと同じ効果を得られる。これも大変便利なので、是非設定しておいてほしい。

7.1.2 Emacs の操作

emacs の操作、特にキーボードショートカットを覚えるのは、多少の時間を要するが、確実に作業効率は向上するので、ぜひ習得してもらいたい。

以下のコマンドを覚えれば基本的な操作は十分であろう。

- カーソル移動
 - C-f カーソルを一文字右に移動
 - C-b カーソルを一文字左に移動
 - C-p カーソルを一行上に移動
 - C-n カーソルを一行下に移動
 - C-a カーソルを行頭に移動
 - C-e カーソルを行末に移動
- 画面の分割
 - C-x 2 画面を上下二分割にする
 - C-x 3 画面を左右二分割にする
 - C-x o 画面分割を移動する
 - C-x 1 画面の分割を元に戻す
- ファイルの読み書き
 - C-x C-s ファイルに現在の内容をセーブ
 - C-x C-f ファイルをオープン
 - C-x C-w ファイルの名前を変更して保存
- Emacs の終了
 - C-x C-c Emacs を終了
- 削除
 - C-d カーソルのある位置にある文字を削除
- 検索
 - C-s カーソルのある位置以降をインクリメンタル検索
- カット & ペースト
 - C-k カーソルのある位置の以降の文字を切り取る
 - C-y C-k できりとった部分をカーソル位置に張り付ける

基本については Emacs Beginner's HOWTO (<http://linuxjf.sourceforge.jp/JFdocs/Emacs-Beginner-HOWTO.html>) に書いてある ,2.3 キーボードの基本 (<http://linuxjf.sourceforge.jp/JFdocs/Emacs-Beginner-HOWTO-2.html#ss2.3>) が参考になる .

よりアドバンスドな内容については GNU Emacs マニュアル (<http://www.bookshelf.jp/texi/emacs-20.6-man-jp/emacs.html>) のマークとリージョン (http://www.bookshelf.jp/texi/emacs-20.6-man-jp/emacs_10.html) について見てみるとよい . リージョンの選択 , コピー , ペースト , ができるようになれば , 一人前だろう .

7.1.3 Emacs 上の shell

Emacs 上で bash を立ち上げることができる .

Emacs 上で , M-x shell と打ち込む (ESC キーを押して離して x を押し , shell と入力する) と , 普通の shell の画面と同様のコマンドプロンプトが Emacs に表示されることが分る . あとは通常と同じようにコマンドを打ち込み , 実行することができる . このモードを「 shell モード」と呼ぶ . shell モードの利点は , プログラムの実行結果やデバッグ文等のメッセージ表示が , エディタ上に残るので , その部分をコピーペーストしてレポートを作成したり , メールで転送したりするなどの作業が容易にできる点である .

7.2 vi エディタの使い方

エディタは vi を使おう¹⁰ .

```
$ vi  
または  
$ vi ファイル名
```

として立ち上がる .

オリジナルは 1976 年ビル・ジョイにが BSD を公開するにあたり開発していた Pascal コンパイラを作成するために作られたものである . この段階では ex と呼ばれるラインエディタであり、その後改良が加えられ vi として公開された . vi はコンパクトで負荷が小さくテキストファイルの一部を変更したり、通信速度のネットワークの先にあるマシンでの編集に向いている . vi/vim では複数のモードを切り替えて利用する点に特徴がある .

現在ではオランダ人のプログラマ Bram Moolenaar が 1991 年に開発した vim が広く使われている .

C-z とすると , shell に戻る . C-z とは Ctrl キーを押してから z を押す , という操作を意味している . これは , プロセスをバックグラウンドで実行している状態である . ここで , fg と打ち込めば , また vi に戻ることができる .

7.2.1 vi の操作

vi にはモードがある . タイプした文字がそのまま反映される編集モードと , 以下の各種コマンドが使えるコマンドモードである . 起動直後はコマンドモードになる .

¹⁰<https://ja.wikipedia.org/wiki/エディタ戦争>

vi の操作、特を覚えるのは、多少の時間を要するが、覚えないと利用できないので、ぜひ習得してもらいたい。

以下のコマンドを覚えれば基本的な操作は十分であろう。

- 編集モードへ移るためのキー

- i 現在のカーソルの位置から挿入
- R 現在のカーソルの位置から置換
- A 現在行の末尾に追加
- O 現在行の前の行に挿入
- o 現在行の次の行に挿入
- ESC 編集モードからノーマルモードに戻る

- カーソル移動

- l カーソルを一文字右に移動
- h カーソルを一文字左に移動
- k カーソルを行上に移動
- j カーソルを行下に移動
- 0 カーソルを行頭に移動
- \$ カーソルを行末に移動
- Enter 次の行の先頭へ
- w 次の単語へ
- b 前の単語へ

- ファイルの読み書き

- :e [ファイル名] ファイル名を指定してオープン
- :w [ファイル名] ファイルの名前を変更して保存

- vi の終了

- :q セーブせずに終了
- :w セーブするが終了しない
- :wq セーブして終了

- 削除

- x 1 文字削除

- 検索

- /正規表現 前方検索
- ?正規表現 後方検索
- n 次の候補
- N 前の候補

- カット&ペースト

- yy 1 行コピー
- dd 1 行カット
- p ペースト

8 タイピングソフトの導入

emacs 用のタイピングソフト trr をインストールして，.emacs ファイルに設定を追加してみよう。
trr のソースはダウンロードは以下の様にダウンロードするする .git コマンドが見つからないものは sudo apt-get install git としてインストールできる。

```
mech-user@ki00001:~$ cd ~/.emacs.d/  
mech-user@ki00001:~/emacs.d$ git clone https://github.com/kawabata/emacs-trr
```

としてソースコードをダウンロードし ~/.emacs.d/init.el に

```
(add-to-list 'load-path "/home/mech-user/.emacs.d/emacs-trr/")  
(setq trr-japanese t) ;; uncomment this to play with Japanese mode  
(require 'trr)
```

と追加して書いておけば，(もう一度 emacs を立ち上げるか M-x load-file で init.el を読み込めば)
M-x trr としてタイピングソフトが立ち上がる。300 点が業界標準とされているので，ぜひ頑張ってほしい。

9 様々なプログラミング言語

世の中には様々な言語が存在し、活用されている。ここでは幾つかのプログラミング言語について、復習プログラムと等価のプログラムを紹介する。

世の中にどのようなプログラミング言語があるかは <http://helloworldcollection.de/> などを参照されたい。

またプログラミング言語を学ぶ際の心構えとしては Google の研究本部長を努めている Peter Norvig (https://en.wikipedia.org/wiki/Peter_Norvig) によるエッセイ (<https://www.yamdas.org/column/technique/21-daysj.html>) を参考にされたい。

9.1 C

C 言語は 1972 年に AT&T ベル研究所で Dennis Ritchie と Ken Thompson により開発された。もともと、UNIX の開発と並行して開発が進められており、高水準言語（高級言語）でありながら、ハードウェアへのアクセスが可能な低水準（低級言語）も可能である。

```
/* test0.c */
#include <stdio.h>

int test(int i, int j) {
    return (i * j);
}

int main(int argc, char *argv) {
    int i,j,k;
    i = 3;
    j = 2;
    k = test(i,j);
    if (k > 5) printf(">5\n");
    else printf("<=5\n");
    return 0;
}
```

C 言語のソースコードは、

```
$ ls
test0.c
$ gcc -o test0 test0.c
test0 test0.c
$ ./test0
```

として、実行ファイルにコンパイルし、実行することが出来る。
その過程でどのような処理が行われているかを理解するためには、以下のプログラムを実行してみよう。

```
$ ls
test0.c
$ gcc -S test0.c
$ cat test0.s
$ as -o test0.o test0.s
$ objdump -d test0.o
$ gcc -o test0 test0.o
$ nm test0.o
$ nm test0
$ ls
test0.c test0 test0.o test0.s
$ ./test0
```

ここで実行したコマンドの意味は、以下のとおりである。

- gcc -S ... C プログラムをアセンブリ言語に変換
- gcc -c ... C プログラムをバイナリ（機械語ファイル）に変換
- as ... アセンブリ言語を バイナリ（機械語ファイル）に変換
- gcc -o ... 出力ファイルの指定
- gcc ... 実行ファイル生成（必要に応じてコンパイル・リンク）
- nm ... シンボル（関数・変数）のアドレス一覧を出力
- objdump -d ... 逆アセンブル（機械語からアセンブリ言語を生成）

test0.s は cat, more, less 等のコマンドで中身を見る能够である。test0.o はバイナリファイルのため、これらのコマンドで中を見てもよくわからない。そういう場合は hexdump test0.o などとしてみよう。

また、ARM プロセッサ用のバイナリも以下のようにして出力し、逆アセンブルで確認することが出来る。

```
arm-none-eabi-gcc -O0 -c -o test0.o test0.c ; arm-none-eabi-objdump -d test0.o
```

arm-none-eabi-gcc は sudo apt install gcc-arm-none-eabi としてインストール可能である。

9.1.1 リダイレクト・パイプの復習

リダイレクトは標準入力、標準出力、標準エラー出力の入出力先を端末（キーボードとモニタ）からファイルに変更する能够がある。

プログラムでは入出力ファイルのオプションに - を指定することで標準入力から読みこむ、あるいは標準出力に書き出すように指示することが可能な場合が多い。また、特にファイルが指定されていなければ、標準入力からファイルを読み込むことが可能な場合が多い。

例えばアセンブリ言語に変換し、さらに機械語ファイルに変換する処理をあえてパイプを使って実現すると以下のようになる。

```
$ gcc -o - -S test0.c | as -o test0.o
```

また、ファイルの行数が多くターミナル上でスクロールしてしまう場合は

```
hexdump test0.o | less
```

等とするのもよい。

パイプを使うことで他の標準出力を標準入力に繋げることが出来る。このように、一つのことを上手くやる小さいプログラムをパイプで組み合わせることが出来るのが、UNIX の特徴であるので是非覚えておこう¹¹。

9.2 Emacs Lisp

EmacsLisp は Emacs で利用されているプログラム言語 LISP の方言である。LISP 自体は 1958 年に John McCarthy により人工知能用のプログラム言語として設計され、ガベージコレクション、動的型付け、高階関数等、多くの先進的なアイデアを生み出してきた。Emacs のほとんどの機能は EmacsLisp を用いて記述されており、Emacs 自体の設定や拡張にも用いられている。trr を利用するために記述した `init.el` もまた EmacsLisp 言語で記していた。

9.2.1 バッファとミニバッファの利用

Emacs を起動し、最初 C-x b (Ctrl キーを押しながら x キーを押し、両方のキーを離す。次に、b キーを押して、リターンを押す) と *scratch* というバッファに遷移する。これは Emacs-Lisp のインタプリタになっている。ここで、

```
(+ 137 349)
```

と打ち込み、行末で C-j を打つと

```
(+ 137 349)  
486
```

という結果が表示されるはずです。括弧始まって括弧で終わる一連の塊を式と呼び、C-j を押すことでこの式を評価しているという。

ミニバッファとは emacs の画面の下にある 1 行の領域である。この部分にメッセージを表示するには

```
(message "Hello World!")
```

と打ち込み C-j とする。

¹¹https://en.wikipedia.org/wiki/Unix_philosophy

```
(message "Hello World!")
"Hello World!" <-- 返り値
```

と、表示され message 関数の結果が返ると共に、ミニバッファにも結果が表示される。以降「<-- 返り値」と表示される部分は、インタプリタの返り値であり、入力する必要がない。

また、EmacsLisp では関数定義の先頭に (interactive) を入れると、キーボード操作で直接呼び出すことができるインタラクティブ関数となる。例えば以下の様な定義をすることで、C-c-h (Ctrl キーを押して c を押し、Ctrl を離して h を押す) とするとバッファに Hello, world を挿入し、ミニバッファメッセージを表示する。

```
(defun hello-world ()
  (interactive)
  (message "Insert Hello, world")
  (insert "Hello, world\n"))
(define-key global-map "\C-ch" 'hello-world)
```

ここでは、それぞれの式の塊の終わりである、それぞれ 4 行目の... world\n)) と、5 行目の... 'hello-world) の行末で C-j を押して式を評価する必要がある。

(current-time-string) という EmacsLisp の組み込み関数を知っていれば以下の様にして現在時刻を挿入する関数とキー割り当てが可能になる。

```
(defun insert-time ()
  (interactive)
  (insert (current-time-string)))
(define-key global-map "\C-ct" 'insert-time)
```

9.2.2 EmacsLisp による復習プログラム

プログラムは以下のようになる。

```
(defun test (i j)
  (* i j))

(setq i 3)
(setq j 2)
(setq k (test i j))
(if (> k 5)
    (print ">5")
    (print "<=5"))
```

各関数定義の最後の行末で C-j を押して式を評価するか、(load-file "./test0.el") としてファイルを読み込み、(main) として main 関数を実行する。何れも*scratch*バッファで実行すること。

関数は defun を用いて定義し、i * j を (* i j) と記述する。

ここで括弧ではさまれた部分の左端の要素を演算子 (operator) と呼び、他の要素を被演算子 (operands) という。式の評価は演算子が指定する手続き (procedure) を被演算子の値である引数 (arguments) に作用させて得る。

この様に演算子を被演算子の前に置く書き方を前置記法 (prefix notation) とか、ポーランド記法 (polish notation) と呼ぶ。この記法のメリットは任意個の引数をとる手続きを許し、また入れ子にして式を拡張することが容易であるという点にある。

9.3 Scheme

Scheme は Lisp の方言の一つとして、1975 年頃に当時 MIT AI Lab の学生だったガイ・スティールとジェラルド・サスマンにより設計された。末尾再帰や継続といった技術が採用されている。

Scheme は多くの実装が有るが、ここでは川井史郎氏が開発している Gauche(ゴーシュ) を利用してみよう。

インタプリタは gosh として立ち上げる。

```
$ cat test0.scm
(define (test i j)
  (* i j))

(define i 3)
(define j 2)
(define k (test i j))
(if (> k 5)
    (print ">5")
    (print "<=5"))

$ gosh
gosh> (load "./test0.scm")
>5
#t
```

Scheme インタプリタがインストールされていない場合は、

```
$ sudo apt install gauche
```

でインストールしよう。

9.4 FORTRAN

FORTRAN は 1954 年に John Backus により開発された高水準言語であり、主に数値計算用に用いられている。

FORTRAN は最初期のプログラミング言語であり、パンチカードでプログラムを記述していた名残から、プログラムを 7 衔目から始め 1 行が 80 衔という制約や、変数や手続き名が最大 6 文字という制約等があった。一方で、当初から複素数型をサポートするなど数値計算向けの機能は充実していた。近年は Fortran と表記し、構造化プログラミングやオブジェクト指向プログラミングなどへも対応している。

以下のようにしてプログラムを実行することができる。

```
$ cat test0.f
C      test0
10   FUNCTION TEST(I, J)
      INTEGER I, J
      TEST = I * J
      RETURN
END FUNCTION TEST

100  INTEGER I, J, K
     I = 3
     J = 2
     K = TEST(I, J)
     IF ( K > 5 ) THEN
       PRINT *, '>5'
     ELSE
       PRINT *, '<=5'
     ENDIF
300  END

$ gfortran test0.f
$ ./test0
>5
```

FORTRAN コンパイラがインストールされていない場合は、

```
$ sudo apt install gfortran
```

でインストールしよう。

9.5 Pascal

Pascal は 1970 年に Niklaus Wirth により開発された構造化プログラミング言語。教育用を目的としつつも、強力な言語であり、初期の Macintosh の OS やアプリケーションも Pascal で記述されていた。

構造化を特徴としスコープの概念、ブロック構造等その後のプログラミング言語の方向性を決めた機能が多く含まれている。

Pascal のコンパイラは全てのラベル、変数、関数などは使用に先立って定義し、コンパイラが 1 回の走査で実行ファイルを生成できる LL(1) パーサである。また、C 言語のような「int x」(「型名 变数名」) という順序の変数定義ではなく、「var x : integer」(「变数名 型名」) という順序で変数を定義する。この仕様は Go 言語などに引き継がれている。また、関数の返り値は、関数名:= 返り値 と書いて指定する。

Pascal コンパイラがインストールされていない場合は、

```
$ sudo apt install fp-compiler-3.0.4
```

でインストールしよう。

```
$ cat test0.p
program test0;
var
  a : integer;
  b : integer;
  k : integer;
function test(i, j :  integer): integer;
var
  result : integer;
begin
  result := i * j;
  test := result;
end;

begin
  a := 2;
  b := 3;
  k := test(a, b);
  if k > 5 then
    writeln('>5')
  else
    writeln('<=5');
end.
$ fpc ./test0.p
$ ./test0
> 5
```

9.6 Java

Java は 1995 年ごろサン・マイクロシステムズで開発されたオブジェクト指向言語。以下の実行例で分かるように javac というコンパイラで.class ファイルと呼ばれる中間言語を作成し、これを実行する java と呼ばれるプログラムで実行する。このとき中間言語はプラットフォームに依存しない形で設計されているため、.class を別のマシン（例えば Windows から Linux）へ転送し java で実行しても同じ結果を得ることができる。初期の Java 言語は、ちょうどインターネットブラウザが黎明期にリリースされており、インターネット上でのプログラムのやり取りや、Web 上でのグラフィカルな表現とインタラクティブな操作を実現するための言語環境として広まっていった。現在は、サーバ側、クライアント側、組み込み、などで広く使われている。

Java による実行例を以下に示す。

```
$ cat test0.java
class test0 {
    public static int test(int i, int j) {
        return ( i * j );
    }
    public static void main(String args[]) {
        int i, j, k;
        i = 3;
        j = 2;
        k = test(i,j);
        if ( k > 5 ) {
            System.out.println(">5");
        } else {
            System.out.println("<=5");
        }
    }
}
$ javac test0.java
$ ls
test0.class test0.java
$ java test0
> 5
```

javac,java などのコマンドが無い場合は

```
$ sudo apt install default-jdk
```

でインストールしよう。途中で DJL ライセンスを受け入れるか聞かれるので承認する必要がある。

9.7 Python

Python は 1995 年ごろに Guido van Rossum が開発したオブジェクト指向プログラミング言語。インタプリタ¹²として動くため、動的型付け、ガベージコレクションなどを特徴とする。また、以下にあるようにブロックの範囲をインデントで表す点に特徴がある。これは C 言語などでは括弧によってブロック構造を表すのと対照的である。見た目とブロック構造を対応つけることで、同じ構造のプログラムは同じ見た目となり、保守性の高いコードを実現できる。本体の仕様はシンプルにし、必要な機能は拡張モジュールで提供するようにしている。おそらく、現在最も勢いのある言語で、様々な拡張モジュールが利用でき便利な環境が実現されている。Google や Yahoo! 等でも利用されていることで有名。

¹² プログラミング言語には大きくインタプリタ言語とコンパイル言語がある。C 言語のようなコンパイル言語ではソースコードからオブジェクトコードを生成しプログラムを実行するが、インタプリタ言語ではソースを逐次解釈しながら実行する。そのため、言語に対して、それを実行するためのインタプリタ環境が必要。これをしばしばランタイム環境と呼ぶ。ただし、インタプリタとコンパイルの両方の側面を持つ言語も存在する。例えば Java .javac を使ってコンパイルするが、その実行する java は class ファイルのランタイム環境と言える。が、Java で対話的に開発することはほとんどないであろう。

```
$ cat test0.py
def test(i,j):
    return ( i * j )

i = 3
j = 2
k = test(i, j)
if k > 5:
    print(">5")
else:
    print("<=$")

$ python test0.py
```

注意したいのは python ではタブでブロックの範囲を指定するので，if 文の次の行などは必ずタブで字下げすること．

また Python はインタプリタとして，プログラムを編集しつつテストしつつ開発を進めることができる．python としてインタプリタを稼動させ，編集したファイルを読み込むことで，プログラムを実行できる．

```
$ python
>>> exec(open("./test0.py").read())
>5
```

ここで test0.py の test 関数を以下のように変更する．

```
def test(i,j):
    print "i = " + repr(i) + ", j = " + repr(j);
    return ( i * j)
```

そして，引き続きインタプリタでファイルを読み込む，変更を反映させた結果を見ることができる．また，関数のみの部分動作確認も可能である．

```
$ python
>>> exec(open("./test0.py").read())
i = 3, j = 2
> 5
>>> test(3,4)
i = 3, j = 4
12
```

慣れてくると，インタプリタでのプログラム開発はやめられなくなる．

9.8 Go

Google が 2009 年から開発してきているプログラミング言語で設計にロブ・パイク，ケン・トンプソンが関わっており，インタプリタ型の動的型付け言語の容易なプログラミングと、コンパイラ型の静的型付け言語の効率と安全性を兼ね備える事を狙っている．

go build でファイルをコンパイルし , go run でファイルをコンパイルした後 , 実行する .

```
$ cat test0.go
package main

import "fmt"

func test(i int, j int) int {
    return (i * j)
}
func main() {
    i, j := 3, 2
    var k int = 0;
    k = test(i, j)
    if k > 5 {
        fmt.Println(">5")
    } else {
        fmt.Println("<=5")
    }
}

$ go build test0.go
$ go run test0.go
>5
```

go では環境変数を設定することでクロスコンパイルが可能である . 例えば以下のように GOARCH を arm64 とすることで , ARM チップ向けの実行ファイルが生成され , GOOS を windows とすることで , Windows 向けの実行ファイルが生成されていることがわかる .

```
$ go build test0.go
$ file test0
test0: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
      with debug_info, not stripped
$ GOARCH=arm64 go build test0.go
$ file test0
test0: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), statically
      linked, with debug_info, not stripped
$ GOOS=windows go build test0.go
$ file test0.exe
test0.exe: PE32+ executable (console) x86-64 (stripped to external PDB),
      for MS Windows
```

Go が入っていない場合は

```
$ sudo apt install golang-go
```

でインストールしよう .

Emacs 上で go プログラムを記述するときは sudo apt install golang-mode として Go 言語用の emacs モードをインストールしておくとよいだろう .

10 Makefile

今後、演習を進めていくに当たって、Makefile はの知識があると都合が良い事が多いだろう。

10.1 Makefile 基本

Makefile のルールの基本は次の形である。

ターゲット:依存するファイル
コマンド

コマンドラインから

make (ターゲット)

とすることで、「ターゲット」が存在しない、あるいはターゲットよりも新しい「依存するファイル」場合にコマンドが実行される。また、コマンド行の始まりは必ずタブコードを入力する。タブコードは Tab と書かれたキーを押すことで入力される。Tab キーを押すとカーソルが数文字分右にずれ、字下げされる。見た目はスペースを複数入力するのと同じだが、実際には 1 文字文のデータからなり制御文字¹³の一種である。Emacs では C-b や C-f でカーソルを移動させると、タブを入力した部分はスペースではなく、字下げされていることが分かる。

例えば main.c, hello.c, goodbye.c の 3 つのソースコードがあり、ここから test1 が作成される場合を考える。ソースコードの例は以下に示す。

```
/* main.c */
#include <stdio.h>

void hello();
void goodbye();

int main() {
    printf("main.\n");
    hello();
    goodbye();
    return 0;
}
```

```
/* hello.c */
#include <stdio.h>

void hello(void) {
    printf("hello!!\n");
}
```

```
/* goodbye.c */
#include <stdio.h>

void goodbye(void) {
    printf("goodbye!!\n");
}
```

コマンドラインから make と打ち込んだら test1 という実行ファイルが作られるようにするには、Makefile というファイルに以下のように記述する。

```
test1: main.c hello.c goodbye.c
        gcc -o test1 main.c hello.c goodbye.c
```

これでコマンドラインから make と打ち込むと、make コマンドはデフォルトで Makefile という名前のファイルを参照し、依存関係を判断し以下のようにコンパイルする。

¹³制御文字は端末を制御するための特殊文字で、例えばリターンキーを打つとデータとしては一文字の 0x0a というコードが入力されるが、表示のときはそこで改行する約束になっている。Tab キーも同様に 0x09 というコードが入力され、指定された場所まで字下げする

```
$ ls
Makefile main.c hello.c goodbye.c
$ make test1
gcc -o test1 main.c hello.c goodbye.c
```

10.2 依存関係の記述

上の例では hello.c が更新された場合でも main.c や goodbye.c のコンパイルが行われてしまい、非効率である。これを回避するためには以下のように main.c から main.o を作るルールと、hello.c から hello.o を作るルールと、goodbye.c から goodbye.o を作るルールと Makefile に書き、test2 は hello.o と goodbye.o と main.o から作られるようにする。そうすると、hello.c が更新された場合、make test2 とすると hello.c のみ再コンパイルされ、不要な main.c や goodbye.c の再コンパイルは行われない。

```
test2: main.o hello.o goodbye.o
      gcc -o test2 main.o hello.o goodbye.o
main.o: main.c
      gcc -o main.o -c main.c
hello.o: hello.c
      gcc -o hello.o -c hello.c
goodbye.o: goodbye.c
      gcc -o goodbye.o -c goodbye.c
```

10.3 自動変数

ルールを書くときに同じことを何回も書くことを避けるために、自動変数が用意されている。

```
test3: main.o hello.o goodbye.o
      gcc -o $@ $^
main.o: main.c
      gcc -o $@ -c $^
hello.o: hello.c
      gcc -o $@ -c $^
goodbye.o: goodbye.c
      gcc -o $@ -c $^
```

\$@ ターゲットのファイル名、\$^ は依存するファイル名であるため、この Makefile は上の例と同じことになる。

\$% ターゲットがアーカイブメンバだったときのターゲットメンバ名

\$< 最初の依存するファイルの名前

\$? ターゲットより新しいすべての依存するファイル名

\$^ すべての依存するファイルの名前

\$+ Makefile と同じ順番の依存するファイルの名前

\$* サフィックスを除いたターゲットの名前

さらに，ファイル名.c からはファイル名.o を作るルールは，次のように書ける．

```
% .o: %.c
    gcc -o $@ -c $<
```

こうすると汎用なルールになり，Makefile は以下のように 4 行でシンプルに書くことができる．

```
test4: main.o hello.o goodbye.o
        gcc -o $@ $^
% .o: %.c
    gcc -o $@ -c $<
```

Makefile は奥が深い¹⁴ので，google で色々と調べてみると良い．

11 CMake

Makefile より奥が深いツールとして CMake がある．これは Makefile を自動で作成するものである．OS によって動的ライブラリの拡張子などが異なるが，そのような違いを吸収する，より抽象度の高いツールになっている．

CMake では Makefile の代わりに CMakeList.txt というファイルを利用する．以下のようなファイルを準備する．

```
add_executable(test4 main.c hello.c goodbye.c)
```

CMake では通常ソースディレクトリが含まれているディレクトリは別のディレクトリで作業する．例えば以下のようにして build というディクレクトリを作りその中で cmake コマンドを実行する．引数は CMakeLists.txt が存在しているソースディレクトリである．cmake コマンドにより Makefile が作成されるので，make ファイルを実行する．make で利用される中間ファイルは全て build 以下に作成されるので，必要なくなればこのディレクトリ消去すれば良い．また，ソースディレクトリに不要な中間ファイルが作成されない（汚染されない，という）．

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

¹⁴とはいえる，奥が深いものも考え方である．最初の設計がもっとよければ簡単に使えるツールになるのに，と思われるものは多い．そのようなものを高林哲さんはバッドノウハウ (BK) と読んで論考している (<http://0xcc.net/misc/badknowhow.html>)．ただ BK として引き継がれているものは，そのようなノウハウの塊で使いづらい以上の利便性を提供しているともいえる．

より複雑な CMakeList.txt の例を以下に紹介する。動的ライブラリを作成している例になっている。

```
cmake_minimum_required(VERSION 2.8)
include_directories(include)
add_library(hello_goodby SHARED hello.c goodbye.c)
add_executable(test4 main.c)
target_link_libraries(test4 hello_goodby)
```

12 宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし、また講義中にでてきたキーワードについて知らなかつたもの、興味のあるものを調べ「学籍番号.txt」としてアップロードすること。テキストファイルはワードファイルなどだと確認出来ないことがあるため、emacs/vi 等のテキストエディタを使って書こう。プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい。

画像で提出する場合は、各自のマシンの Mac アドレスが分かるようにすること。例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい。
ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと。また授業中に質問した者はその旨を記すこと。質問は成績評価時の加点対象となる。

キーワード：OS, 仮想マシン, Unix, Linux, Ubuntu, ディストリビューション, Unix 哲学とパイプ, 広義の/狭義の, 仮想マシン, ハイパーテザ, ゲスト OS, コンテナ, ホームポジション(タピング), コンパイラ, インタプリタ, アセンブラー, 構造化プログラム言語, 関数型言語, パーチャルマシン(言語実行系)

1. これまでの各自の経験で「広義の/狭義の」を使ったら良い場面、概念を探し、説明してみよ。
2. これまでの各自の経験、あるいは将来起こりそうな事柄で「1一つのことを行い、またそれをうまくやるプログラムを書け。協調して動くプログラムを書け。」という UNIX 哲学が当てはまる場面を探し、説明してみよ。
3. trr に挑戦し、スコアを報告せよ (1. の ICCAD を推奨する)
4. C 言語による復習プログラムを emacs または vi で記述し実行せよ。
5. 以下のコードを Makefile というファイルに記述し、

```
all: c scheme pascal java python go fortran

c: test0.c
    gcc -o test0-c test0.c
    ./test0-c

emacs: test0.el
    emacs --load ./test0.el

scheme: test0.scm
    gosh -e '(progn (load "./test0.scm") (exit))'

pascal: test0.p
    fpc -otest0-p ./test0.p
    ./test0-p

java: test0.java
    javac test0.java
    java test0

python: test0.py
    python test0.py

go: test0.go
    go build -o test0-go test0.go
    ./test0-go
    go run test0.go

fortran: test0.f
    gfortran test0.f -o test0-f
    ./test0-f

go-arm64:
    GOARCH=arm64 go build -o test0-go-arm64 test0.go
    file test0-go-arm64

clean:
    -rm -f test0.s test0.o test0-c test0-f test0-p test0.class test0-go test0-go-arm64
```

make と実行すると

```

mech-user@ki00001:~/work/soft2/0410$ make
gcc -o test0-c test0.c
./test0-c
>5
gosh -e '(progn (load "./test0.scm") (exit))'
>5
fpc -otest0-p ./test0.p
Free Pascal Compiler version 3.0.4+dfsg-23 [2019/11/25] for x86_64
Copyright (c) 1993-2017 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling ./test0.p
Linking test0-p
/usr/bin/ld.bfd: warning: link.res contains output sections; did you forget -T?
25 lines compiled, 0.1 sec
./test0-p
>5
javac test0.java
java test0
>5
python test0.py
>5
go build -o test0-go test0.go
./test0-go
>5
go run test0.go
>5
gfortran test0.f -o test0-f
./test0-f
>5

```

のように表示されるようにせよ .

- 宿題に取り組んでいる際にエラーが出た場合は , その画面やプログラムをコピーしておくこと . 次週の課題で利用する .
- 普段 GUI を利用しているものも , 本講義内ではターミナルになれておいてほしい . 特に emacs/vi のキーバインディングは業界標準になっているので , 覚えておいて損はない . 例えば gmail なども

```

Previous message in an open conversation p
Next message in an open conversation n

Newer conversation k
Older conversation j

```

というショートカットキーが存在する¹⁵ . これは ,

- Emacs
 - * C-p カーソルを一行上に移動
 - * C-n カーソルを一行下に移動
- vi
 - * k カーソルを一行上に移動
 - * j カーソルを一行下に移動

を知っていると直感的に利用できる .

¹⁵Keyboard shortcuts for Gmail: <https://t.ly/hEXB>

13 分からないときは

演習中や、自分でプログラミングを試してみるような場合に、色々とわからないことが出てくるだろう。そのような場合、講義中で良いのですぐに聞いてほしい。また、それ以外のときも k-okada-soft2@jsk.t.u-tokyo.ac.jp に聞いてほしい。

その際、何に困っているか？どういう現象かを適切に伝えるため、相手の計算機でも同じ問題が生じるような手続きを伝えることが重要である。実はこの様に問題を適切に把握し他人にも伝えることが出きるようになるのが一番難しい。これができれば、分からぬことの 90% は解決したようなものだ。分からぬことが出てきたら、何が分からぬか、どう分からぬかを上手に伝えるためにはどうしたらいいか、考えてみよう。また、他人の質問に答えるのも大変勉強になるので積極的にトライして欲しい。いずれにせよ、自ら質問してみる、答えてみるという経験を通じて適切な質問ができるようになれば、自らトラブルを修正することができるようになるので積極的に機会を捉えてほしい。

プログラミングに関しては英語情報のほうが多いので検索する言語に英語を含めるのが良い。そこで stackoverflow, askubuntu や ubuntuforums などの情報がでてくれれば正解に近づいている、と判断できる。

ただし、インターネットの情報は常に正しいとは限らないので要注意!!! 鵜呑みにしてはいけない。必ず、複数の情報源から総合的に判断すること。とくに、動きませんでした、ダメでした、というだけの報告、問題を解決しているが、実は本質的な解決になっていない報告等がよくあるので要注意。stackoverflow のように vote システムがあるものを参考にするのが良い（例えばこの資料を作成中に <https://github.com/MicrosoftDocs/WSL/issues/499#issuecomment-538650974> を参考にして、Vote した）。

また、Linux でコマンドの役割やオプションを知りたい時は、man を活用しよう。Linux では端末を開いて¹⁶、man gcc, man ar, man objdump, man man, man ps, man uniq, man sort, man cut 等々としてみよう。

¹⁶ 例えば背景を右クリックして「端末を開く」を選択すると「gnome 端末」が起動する。

A ChatGPTによる回答例

2023/04/09 11:42

ICCADでの回路設計



ICCADでの回路設計



ttrに挑戦し、スコアを報告せよ (1. の ICCAD を推奨する)



申し訳ありませんが、私は人工知能の言語モデルであり、"ttr" が何を指すのかを理解できません。もし質問者が "ttr" が何であるかを説明してくれれば、できる限りお答えします。また、ICCADとは、電子回路設計に関する国際会議であり、私がどのように推薦することができるのかも詳しく教えてください。



ttrはemacs用のタイピングソフトで<https://github.com/kawabata/emacs-ttr>からダウンロードできます



ありがとうございます。私はプログラミングの専門家ではありませんが、"ttr"をダウンロードして試してみることができます。

ただし、私は言語モデルであり、物理的なキーボードを持っていないため、実際のタイピングスコアを報告することはできません。あなたが "ttr" を試してみて、どのように感じたかや、機能の詳細についての質問があれば、お答えできるかもしれません。



ttrに挑戦し、スコアを報告せよ



申し訳ありませんが、私は物理的なキーボードを持っていないため、"ttr"を利用して実際にタイピングスコアを報告することはできません。しかし、以下のリンクを参照することで、"ttr"の使い方や機能について学ぶことができます。

<https://github.com/kawabata/emacs-ttr>



もし、"ttr"を使用している上で何か問題がある場合や質問がある場合は、遠慮なくお知

Send a message...



ChatGPT Mar 23 Version. Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts

A. ChatGPT による回答例

2023/04/09 11:47 C言語プログラムの実行

三 C言語による復習プログラムを emacs または vi で記述し実行せよ。

申し訳ありませんが、私は人工知能のモデルであり、C言語のコンパイル環境を持っていないため、C言語の復習プログラムを実行することはできません。しかし、以下はC言語で "Hello World" を表示するプログラムの例です。emacsまたはviでテキストエディタを開き、以下のコードを入力してください。

```
arduino
#include <stdio.h>

int main() {
    printf("Hello World");
    return 0;
}
```

このプログラムを保存してから、ターミナルで以下のコマンドを使用してコンパイルおよび実行できます。

```
bash
gcc -o hello hello.c
./hello
```

これにより、"Hello World" がターミナルに表示されます。

2023/04/09 11:47 C言語による復習プログラムを C 言語以外の言語で記述し実行せよ。 このときも emacs または vi で記述すること。

ここではPythonで簡単なプログラムを作成し、emacsまたはviでテキストエディタを開いて編集し、実行する方法を説明します。Pythonはインタプリタ言語であるため、コンパイルは必要ありません。

まず、以下のプログラムを作成して保存します。

```
python
print("Hello World")
```

保存したら、ターミナルで以下のコマンドを使用してプログラムを実行します。

```
bash
python3 filename.py
```

ここで "filename.py" は先ほど保存したファイル名です。ターミナルに "Hello World" と表示されるはずです。

次に、emacsまたはviでファイルを開きます。ターミナルで以下のコマンドを使用して、EmacsまたはViを開くことができます。

Send a message...

ChatGPT Mar 23 Version. Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts
<https://chat.openai.com/chat/817adee2-51ef-4de4-8dc7-27a296e9c11>

1/1

2023/04/09 11:51 C言語プログラムの実行

三 Go言語で記述して実行してください

以下は、Go言語で "Hello World" を表示するプログラムの例です。Emacsまたはviでテキストエディタを開き、以下のコードを入力してください。

```
go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

このプログラムを保存してから、ターミナルで以下のコマンドを使用してコンパイルおよび実行できます。

```
go
go run hello.go
```

これにより、"Hello, World!" がターミナルに表示されます。

2023/04/09 11:51 C言語プログラムの実行

三 上記のプログラムがすべて make コマンドで実行されるようにしてみよ。

了解です。Makefileを使用して、プログラムをコンパイルして実行できます。以下は、C言語の "Hello World" プログラムの例です。

まず、以下のコードを含むファイル 'hello.c' を作成します。

```
c
#include <stdio.h>

int main() {
    printf("Hello World");
    return 0;
}
```

次に、以下のコードを含む 'Makefile' を作成します。

```
make
all: hello

hello: hello.c
    gcc -o hello hello.c

clean:
    rm -f hello
```

これで、'make' コマンドを使用して 'hello' プログラムをコンパイルできます。ターミナルで以下のコマンドを入力してください。

Send a message...

ChatGPT Mar 23 Version. Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts
<https://chat.openai.com/chat/817adee2-51ef-4de4-8dc7-27a296e9c11>

1/1

2023/04/09 12:07

日本語：「広義の/狭義の」の使い方について English: "Usage of broad/narrow definition"

日本語：「広義の/狭義の」の使い方について English: "Usage of broad/narrow definition" +



これまでの各自の経験で「広義の/狭義の」を使ったら良い場面、概念を探し、説明してみよ。



ごめんなさい、私は日本語の回答をすることができますが、質問が明確ではありません。どのような経験について話しているのか、また「広義の/狭義の」をどのように使うべきかを理解する必要があります。もう少し詳細な質問をしていただけますか？私は喜んでお答えします。



「広義の」の用例を教えてください。



「広義の」とは、ある概念をより包括的に捉えるときに使われる言葉です。以下に、一般的な用例をいくつか紹介します。

1. 「人間」の概念

「人間」という言葉は、狭義には人間種に属する生物を指しますが、広義にはその他の意味も含みます。例えば、人間の文化や社会的行動、感情や知性、身体的能力や健康など、人間としての多様な側面が含まれます。

2. 「芸術」の概念

「芸術」という言葉も、狭義には美術、音楽、文学などの具体的な分野を指しますが、広義には、創造的な表現全般を含みます。例えば、ダンス、演劇、映画、デザイン、アニメーション、コンピューターゲームなど、多様な表現形式が広義の芸術に含まれます。

3. 「自由」の概念

「自由」という言葉は、狭義には政治的自由、経済的自由、表現の自由など特定の意味を持ちますが、広義には人間が自己決定をするための権利や、束縛や拘束からの解放、精神的な自由など、幅広い意味を含みます。



これらの例からもわかるように、「広義の」という言葉は、ある概念の本質的な特徴

Send a message...



ChatGPT Mar 23 Version. Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts

2023/04/09 12:07 UNIX哲学の例

☰ UNIX哲学の例 +

 れまでの各自の経験、あるいは将来起こりそうな事柄で「一つのことを行い、またそれをうまくやるプログラムを書け。協調して動くプログラムを書け。」というUNIX哲学が当てはまる場面を探さし、説明してみよ。



UNIX哲学には、シンプルで小さく、一つのことをうまく行うプログラムを書くことが重要であり、また、それらを組み合わせて協調して動作するプログラムを書くことも重要です。以下には、これらの原則が当てはまるいくつかの場面を示します。

1. Webサイトの構築: Webサイトを構築する際に、UNIX哲学は非常に重要です。Webサイトを構築するためには、複数のプログラムが必要になります。たとえば、Webサーバー、データベース、画像処理、セキュリティなどのプログラムが必要です。しかし、UNIX哲学では、それぞれのプログラムが一つのことをうまく行い、それらを組み合わせてWebサイトを構築することが重要です。
2. ソフトウェアの開発: ソフトウェアの開発においても、UNIX哲学は非常に重要です。ソフトウェアを開発するためには、多数のプログラムが必要になります。たとえば、データベース、ネットワーキング、グラフィックス、デバッグなどのプログラムが必要です。しかし、UNIX哲学では、それぞれのプログラムが一つのことをうまく行い、それらを組み合わせてソフトウェアを開発することが重要です。
3. データ分析: データ分析においても、UNIX哲学は非常に重要です。データ分析には、多数のプログラムが必要になります。たとえば、データ収集、データクリーニング、データ処理、データ可視化などのプログラムが必要です。しかし、UNIX哲学では、それぞれのプログラムが一つのことをうまく行い、それらを組み合わせてデータ分析を行うことが重要です。
4. クラウドコンピューティング: クラウドコンピューティングにおいても、UNIX哲学は非常に重要です。クラウドコンピューティングには、多数のプログラムが必要になります。たとえば、サーバー仮想化、データストレージ、ネットワーキング、監視などのプログラムが必要です。しかし、UNIX哲学では、それぞれのプログラムが一つのことをうまく行い、それらを

Send a message...  

ChatGPT Mar 23 Version. Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts

<https://chat.openai.com/chat/ee9a4155-c390-4fca-8c1a-6ba294413fde> 1/1