システムソフトウェア

2020年度

xv6のインストールと実行

月曜7-8限・木曜7-8限(Zoom)

講義担当：渡部卓雄 (Takuo Watanabe)

http://titech-os.github.io

e-mail: takuo∅c.titech.ac.jp

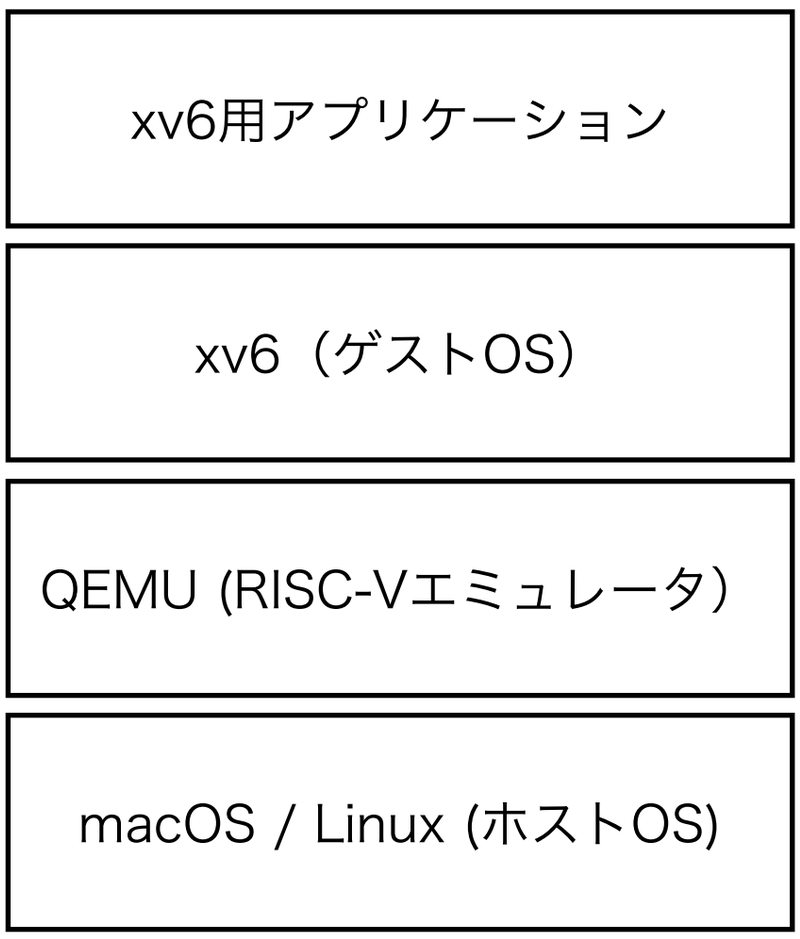
本日のメニュー

- xv6のインストールの仕方・使い方
  - パソコンにインストールして実行する方法について，講義Webサイトに簡単な解説を掲載している
    https://titech-os.github.io/xv6.html

xv6

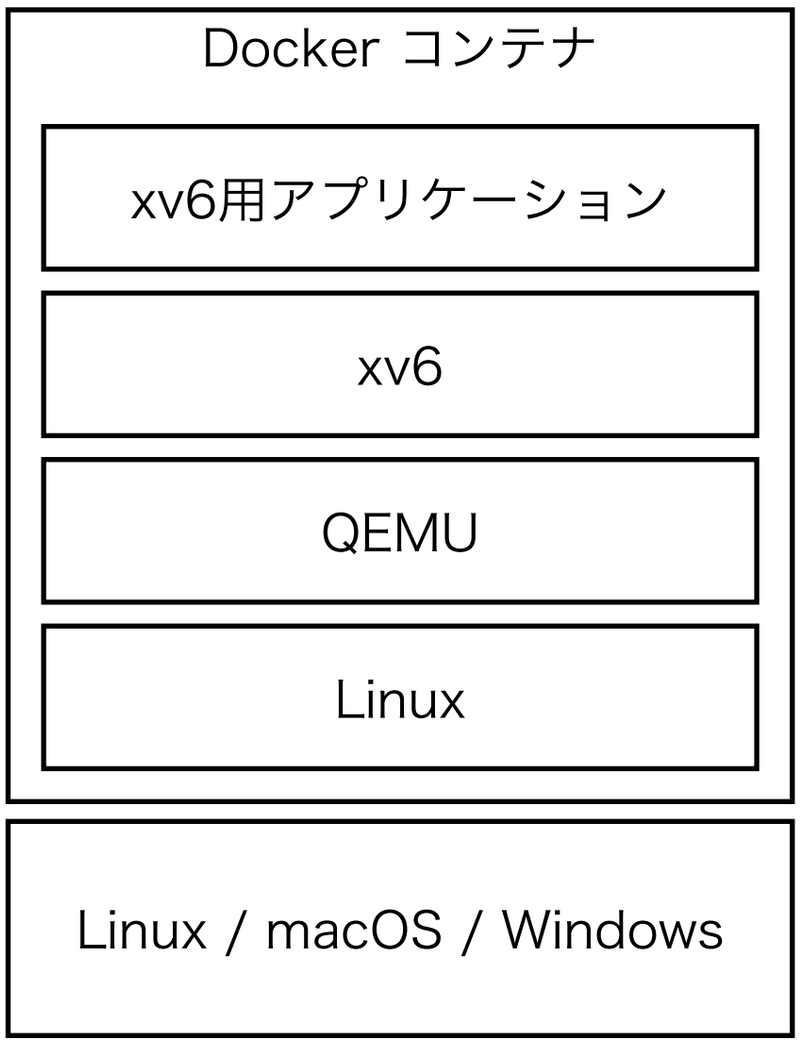
- MITで作られた教育用OS
  - https://github.com/mit-pdos/xv6-riscv
  - RISC-V マルチコアマシンで動作する.
  - 旧バージョンはx86(i386)
  - C（一部アセンブラ）で記述されている.
    - .c, .h, .S ファイルの合計は9819行
- 関連：Unix v6
  - ベル研究所で1975年に作られた初期のUnix
  - DECのPDP-11というマシンで動作する.
  - 古い構文の（pre-K&R）Cで記述されている.
- xv6はv6の移植ではなく，新規に作成したもの.

パソコン上でのxv6の実行(1)



- macOS/Linux上でRISC-Vエミュレータ（仮想マシン）QEMUを動作させ，その上でxv6を実行する.
  - xv6のカーネルやアプリケーションプログラムのソースコードはホストOS上でコンパイル・リンクし，仮想ディスクに入れておく.

パソコン上でのxv6の実行(2)



- Dockerコンテナ内でLinuxを動かし，その上でQEMUを使ってxv6を実行する
- 開発ツールを含むDockerイメージを用意してある
  - wtakuo/xv6-env

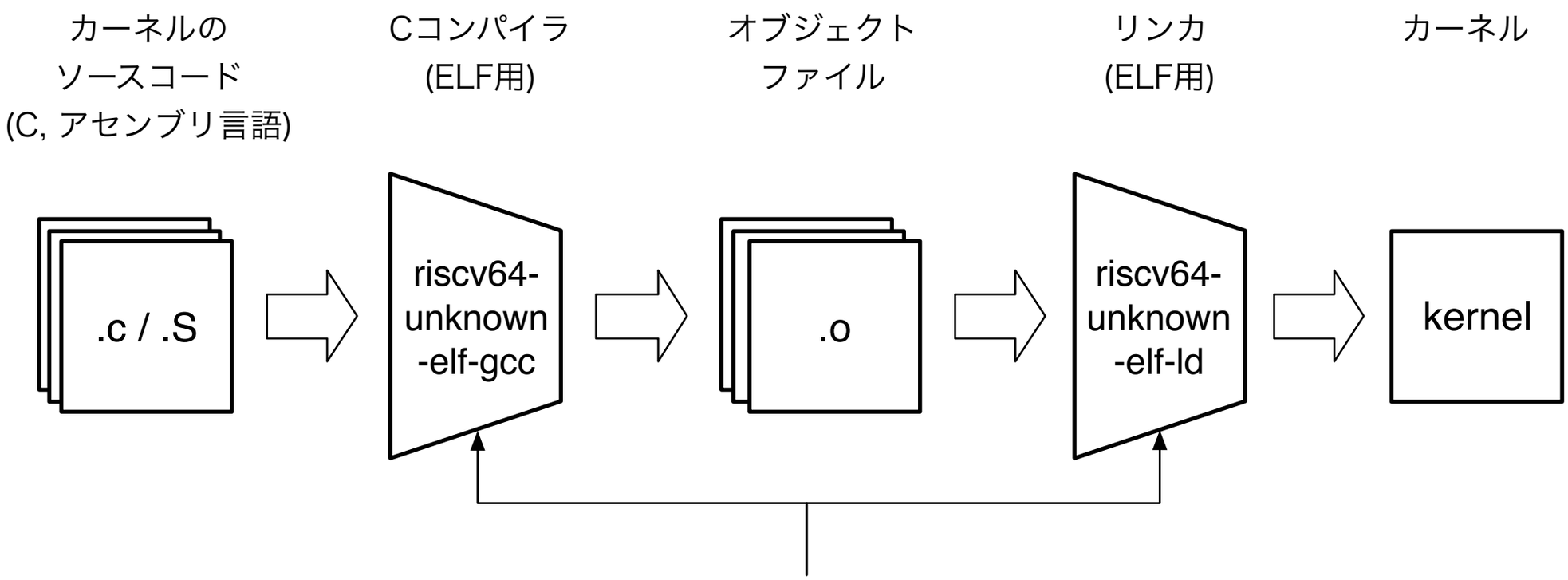
パソコン上でのxv6の実行(3)

- WindowsのLinux仮想実行環境WSL2上でLinuxを動かし，その上でQEMUを使ってxv6を実行する．
- QEMUはWSL2を介さなくてもWindows上で動作可能だが，クロスコンパイラ等がLinuxでないと動作しない．

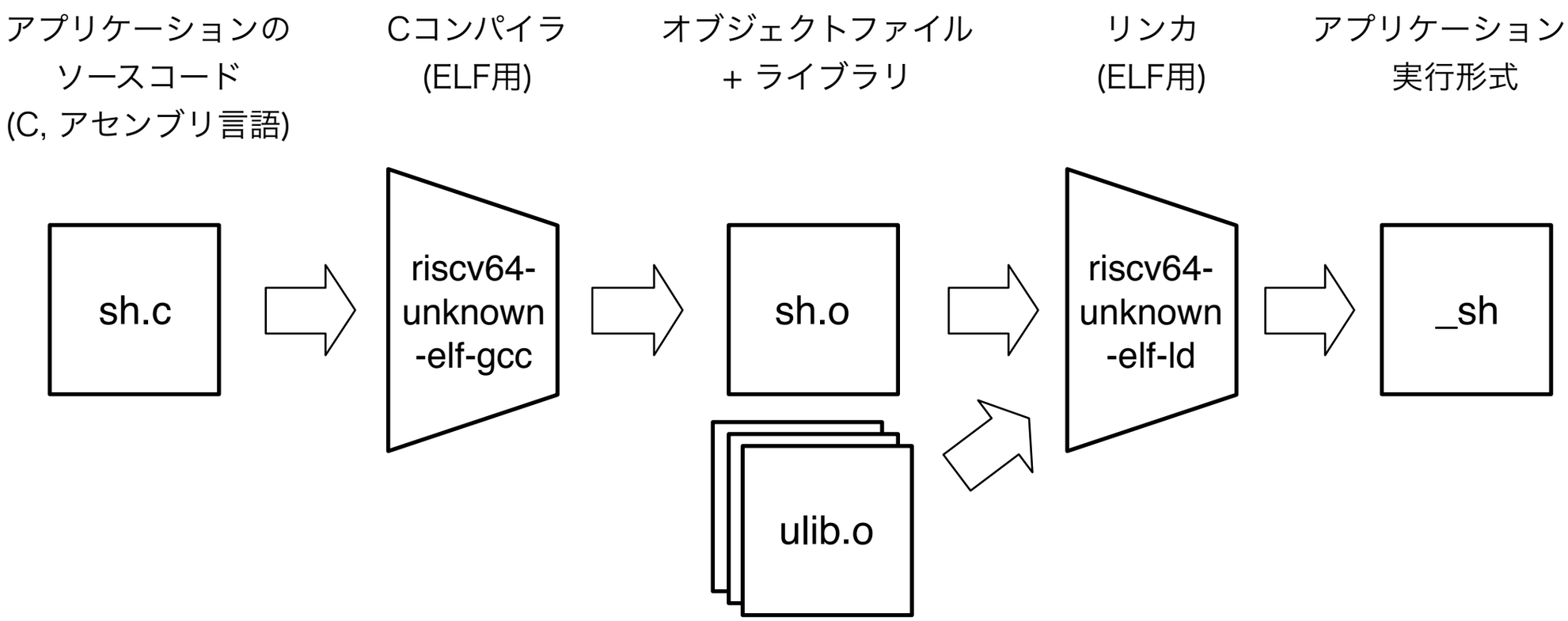
QEMUが模倣するPC

- 構成
  - CPU: RISC-V (64ビット)
  - メモリ: 512MB
  - HDD(IDE)×2
    - 1台はブート用，もう1台はファイルシステム用
  - PS/2キーボード
  - ディスプレイ
- 以上の構成はQEMUの設定で変更できる
- その他，ネットワークや時計等のデバイスも（ドライバを書けば）使うことができる

カーネルのコンパイル

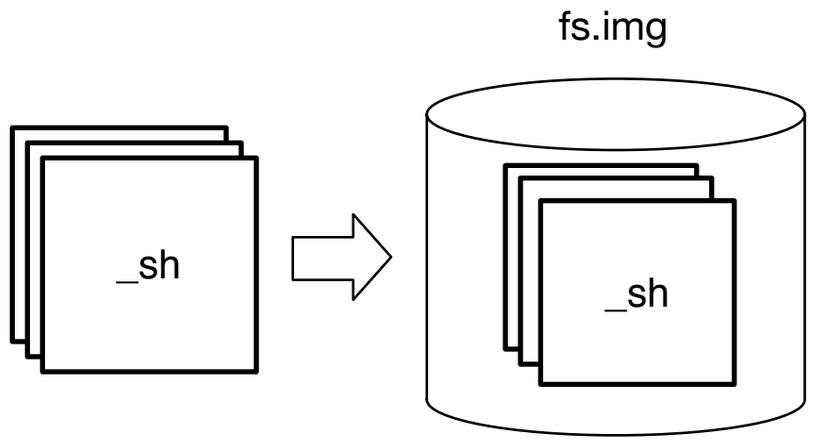

カーネルの
ソースコード
(C, アセンブリ言語)
Cコンパイラ
(ELF用)
オブジェクト
ファイル
リンカ
(ELF用)
カーネル
.c / .S
riscv64-
unknown
-elf-gcc
.o
riscv64-
unknown
-elf-ld
kernel
ホストOS（Mac OS X）上で動作する

アプリケーションのコンパイル



仮想ディスク

- QEMUが模倣するコンピュータは，ファイルシステム用の「ディスク」を持つ.



- カーネルのコードはディスク上にはなく，起動時にメモリに直接読み込まれる.

ビルド用ツール

- macOS/Linux(x86)用のCコンパイラ(clang, gcc)は使えない.
  - アーキテクチャとバイナリ形式が異なるため
    - xv6はRISC-V/ELF．macOSはx86-64/Mach-O.
- Homebewやaptなどを使ってRISC-Vのバイナリを出力するコンパイラ(gcc)と関連ツールをインストールしておく

```
$ ls /usr/local/bin↵
riscv64-unknown-elf-addr2line     riscv64-unknown-elf-gcov-tool
riscv64-unknown-elf-ar            riscv64-unknown-elf-gdb
riscv64-unknown-elf-as            riscv64-unknown-elf-gdb-add-index
riscv64-unknown-elf-c++           riscv64-unknown-elf-gprof
riscv64-unknown-elf-c++filt       riscv64-unknown-elf-ld
riscv64-unknown-elf-cpp           riscv64-unknown-elf-ld.bfd
riscv64-unknown-elf-elfedit       riscv64-unknown-elf-nm
riscv64-unknown-elf-g++           riscv64-unknown-elf-objcopy
riscv64-unknown-elf-gcc           riscv64-unknown-elf-objdump
riscv64-unknown-elf-gcc-9.2.0     riscv64-unknown-elf-ranlib
riscv64-unknown-elf-gcc-ar        riscv64-unknown-elf-readelf
riscv64-unknown-elf-gcc-nm        riscv64-unknown-elf-run
riscv64-unknown-elf-gcc-ranlib    riscv64-unknown-elf-size
riscv64-unknown-elf-gcov          riscv64-unknown-elf-strings
riscv64-unknown-elf-gcov-dump     riscv64-unknown-elf-strip
```

ビルド用ツールのインストール

- macOS
  - Homebrew(パッケージシステム)を使うのが簡単

```
$ brew tap riscv/riscv
$ brew install riscv-tools
$ brew install qemu
```

- Linux（aptを使うディストリビューション）

```
$ sudo apt-get install git build-essential gdb-
multiarch qemu-system-misc gcc-riscv64-linux-gnu
binutils-riscv64-linux-gnu
```

ビルド用ツールの確認

- 必要なコマンドがシェルの実行パスに入っていることを確認する.

```
$ which riscv64-unkown-elf-gcc↵
/usr/local/bin/riscv64-unkown-elf-gcc
$ which qemu-system-riscv64↵
/usr/local/bin/qemu-system-riscv64
```

- 上記のように出力されなかった場合は, 次のスライドにしたがって実行パスの設定を行うこと.

xv6のソースコードの取得

- 以下のいずれかのやりかたで自分のホームディレクトリにxv6のソースコードを展開する

gitを使う場合

```
$ cd↵
$ git clone https://github.com/titech-os/xv6-riscv.git↵
```

ssh経由でgitを使う場合

```
$ cd↵
$ git clone git@github.com:titech-os/xv6-riscv.git↵
```

GitHubで自分のアカウント内にforkしてからcloneしてもよい

xv6のビルドと起動

- xv6をビルドする

```
$ cd xv6-riscv↵
$ make↵
```

- xv6を起動する

```
$ make qemu↵
```

起動の様子

```
$ make qemu↵
...
qemu-system-riscv64 -machine virt -kernel kernel/kernel -m 3G -smp 3
-nographic -drive file=fs.img,if=none,format=raw,id=x0 -device
virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
...
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
```

xv6上で起動したシェルのプロンプト

xv6上でのlsコマンドの実行例

```
$ ls↵
.               1 1 1024
..              1 1 1024
README          2 2 1982
cat             2 3 22536
echo            2 4 21368
forktest        2 5 11704
grep            2 6 25880
init            2 7 22120
kill            2 8 21344
ln              2 9 21288
ls              2 10 24768
mkdir           2 11 21448
rm              2 12 21432
sh              2 13 40320
stressfs        2 14 22440
usertests       2 15 110520
wc              2 16 23680
zombie          2 17 20840
console         3 18 0
$
```

その他のコマンド

```
$ echo Hello↵
Hello
$ grep xv6 README↵
xv6 is a re-implementation of Dennis Ritchie's and Ken
Thompson's Unix
...
$ wc README↵
43 286 1982 README
$ mkdir tmp↵
$ cd tmp↵
$ ../ls↵
.                 1 19 32
..                1 1 1024
```

他に，テストプログラム forktest, usertests 等も実行してみること（少し時間がかかる）.

xv6(QEMU)の停止

- xv6を実行しているターミナルでctrl-aをタイプし，続けてxをタイプする.

makeの引数

- make
  - xv6のカーネルをビルドする.
- make clean
  - ビルド結果を消去する.
- make qemu
  - ユーザプログラムをビルドし,
  - QEMUを使ってxv6を起動する.
- make qemu-gdb
  - デバッガgdbを接続可能な状態でxv6を起動する.

ソースファイル

- kernel/　カーネルのソースファイル
  - *.c, *.h, *.S: Cソースとヘッダ，アセンブリソース
  - kernel.ld: ローダスクリプト
- user/　ユーザプログラムのソースファイル
  - *.c, *.h, *.S: Cソースとヘッダ，アセンブリソース
  - usys.pl: システムコールのエントリ作成用Perlスクリプト
- Makefile
  - makeによるビルド方法の記述
- doc/ ドキュメント
- README, LICENSE

ビルドにより作成されるもの

- kernel/kernel
  - xv6カーネル（実行形式）
- fs.img
  - ユーザファイルシステム用ディスクのイメージ
    - _cat, _echo, _forktest, ..., _zombie
      - Xv6用実行可能ファイル（fs.imgに含まれる）
- その他
  - *.o : オブジェクトファイル
  - *.sym : シンボルファイル(ld)
  - *.d : 依存関係記述ファイル（makeが用いる）

xv6上で動作するプログラムの作成

エディタで以下の内容のCソースファイルを作り，~/xv6/user にhello.cという名前で保存する.

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main (void) {
    printf("Hello, World!\n");
    exit(0);
}
```

- exit()の呼び出しは省略できない

Makefileを編集して以下のように実行形式の名前を
UPROGSに追加する.

```
UPROGS=\
        $U/_cat\
        $U/_echo\
        ...
        $U/_wc\
        $U/_zombie\
        $U/_hello
```

ファイルを作り直して実行する.

```
$ make qemu↵
```

デバッガ(gdb)を使って実行するには

以下のようにして起動する.

```
$ make qemu-gdb↵
```

xv6は起動途中で停止する.

ターミナルの別ウィンドウを開いて以下のようにする.

（Linuxの場合はgdb-multiarchを実行する）

```
$ cd ~/xv6-riscv↵
$ riscv64-unknown-elf-gdb↵
...
GNU gdb (GDB) 8.3.0.20190516-git
...
The target architecture is assumed to be riscv
...
0x0000000000001000 in ?? ()
(gdb)
```

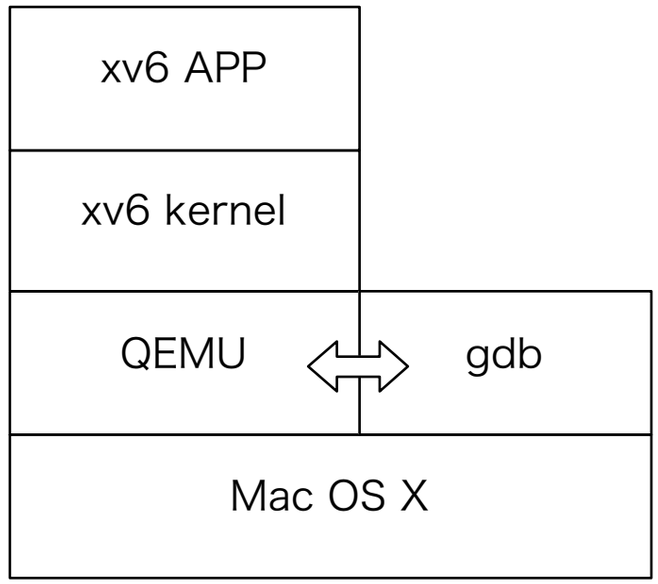
GDBを使う上での注意

- ~/.gdbinit に以下の内容を書いておかないと起動しない

```
add-auto-load-safe-path /Users/takuo/xv6-riscv/.gdbinit
```

  - /Usrs/takuo/xv6-riscv は各自のxv6-riscvへのフルパスにすること

gdbの使い方



- continue, c
  - 停止中の実行を再開する.
- ^C , ctrl-C
  - デバッグ中のプログラムを停止する.
- break 関数名, b 関数名
  - 指定された関数が呼び出された時点で実行を中断する.
- print 変数名, p 変数名
  - 変数の値を出力する.
- list
  - 停止している近辺のソースを表示する.

make qemu-gdbで実行するとブート時点で停止しているので,
とりあえず実行を再開させる.

```
(gdb) continue↵
Continuing.
```

gdbでctrl-Cをタイプして実行を中断させる.

```
^C
Thread 1 received signal SIGINT, Interrupt.
intr_off () at kernel/riscv.h:60
60    asm volatile("csrw sstatus, %0" : : "r" (x));
(gdb)
```

関数execにブレークポイントを設定して実行を再開する.

```
(gdb) break exec
Breakpoint 1 at 0x800048cc: file kernel/exec.c, line 14.
(gdb) continue
Continuing.
```

QEMUのコンソールウィンドウでlsを実行してみると，以下のように実行が中断されてgdbのプロンプトが現れる.

```
path=path@entry=0x3fffff9f00 "ls",
    argv=argv@entry=0x3fffff9e00) at kernel/exec.c:14
14 {
(gdb)
```

変数の値の出力

```
(gdb) print path↵
$1 = 0x3fffff9f00 "ls"
```

1ステップ実行（関数呼び出しの中に入らない）

```
(gdb) next
22    struct proc *p = myproc();
(gdb)
```

stepコマンドはnextと似ているが，関数呼び出しがあるとその中にはいっていく.