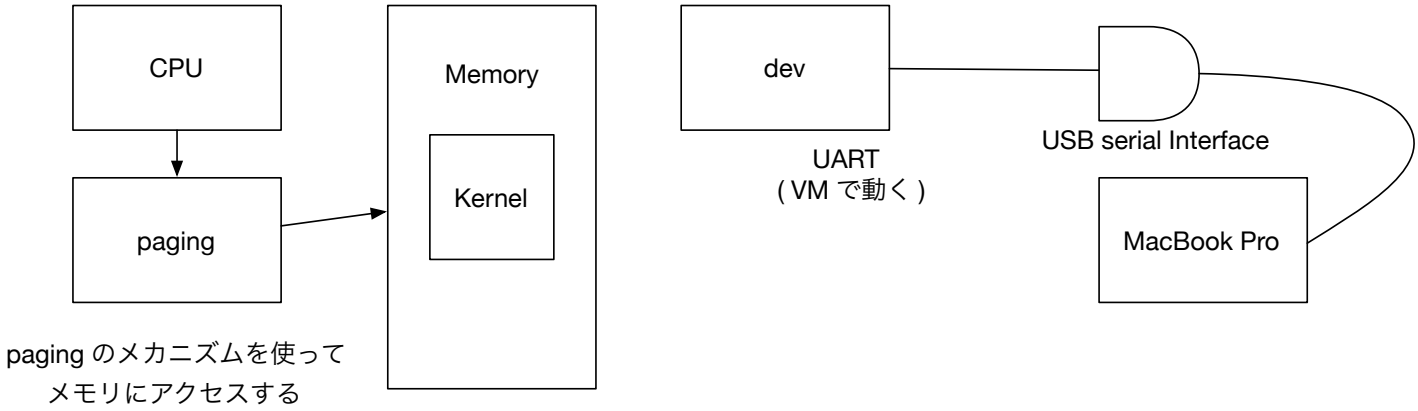
X.v6読み会

Boot Sequence

Boot

```
        Kernel load
Page    Table (Boot用のPagetableの設定)
initialize (memory / IO / process / file )
ini process
```



- Kernel load
    - x86ならEFIがある
    - ARMだとVersatile(firmware)
- paging を設定しなければ動かない
- Boot用のpagetableの設定ができた後一連の初期化を行う
    - memory
    - I/O
    - process
    - file
- 初期化が終わった後
    - init process(process番号1番)
    - init process は linux だと rc.d/~ の下にある

ls 時の User と System

```
User        Sys
fork
exec        ls  (lsを探す)
            open    ls
            load    ls
            (loadの後にmemory空間の初期化を行う)
            pase table reset
            goto User Mode
opendir
            system call (file IO) (VM)
            (filesystemを読んでまたUserに値を返す)
```

- 10行目 **VM** の説明
    - lsは自分で仮装メモリにアクセスしていくので memory allocation をやる
    - malloc は User library だが memory の要求は System でやらなきゃいけない
    - break とい System call で memory を増やす
    - memory を増やすと OS は最初に ls のバイナリを生成
    - すると、break で取った領域が別にできる
    - breakで 取った memory が全部 リアル memory に割り当てられるわけではない
    - いくつかは仮想メモリに行く
    - memory にアクセスした時に仮装メモリだったらtrapしてmemoryを割り当てる
    - memoryが割り当てられなければ、他の実メモリを追い出して書き換える
    - VM関係の一連の処理がある
- fork
    - forkすると process structure ができる
    - process自体は active や waitの状態を持っている
    - 複数のactiveがあると順に実行していく(scheduler)
- kernel の要素
    - process manegement
    - scheduler
    - file IO
    - Virtual Memory

この4つの Kernel の要素を読んでいく

X.v6 の trace

arm 用の gdb で kernel.elf を開く

dalmoreに入って

```
cd /mnt/dalmore-home/one/src/xv6-rpi/src
/net/open/Linux/arm/gcc-arm-none-eabi-7-2017-q4-major/bin/arm-none-eabi-gdb kernel
.elf
```

qemuを立ち上げる 見たいのは Boot Sequence だから qemu-debug を使う

```
make clean; make -f makefile-armclang qemu-debug
```

makefile-armclang の qemu-debug 部分

```
qemu-debug : kernel.elf
        @clear
        @echo "Press Ctrl-A and then X to terminate QEMU session\n"
        export QEMU_AUDIO_DRV=none ; $(QEMU)  -M versatilepb -m 128 -cpu ${QEMUCPU
} -nographic -singlestep -d exec,cpu,guest_errors -D qemu.log -kernel kernel.elf -
s -S
```

- -M versatilepb
  - 仮想メモリにこの firmwere を使うという指示
  - versarilepb は Raspi(ARM) のfirmwereの一種
- -m 128
  - memory の量
- -cpu ${QEMUCPU}
  - cpu の種類
  - アーキテクチャによって命令違う
  - armのxv6を作る時のcompile時にcpuに教えるcpuの種類と合わせる必要がある
  - 名前がqemu側とcompile側のcpuの名前が違うので試行錯誤して合わせる
- -s -S
  - Boot時に debugger が接続するまで止めるようにする

gdb

```
(gdb)b _start

# remote 接続
```

```
(gdb)target remote :1234

# bt: stackを参照して、stack上のfanction callの履歴を調べる
(gdb)bt

(gdb)info registarts (初期のレジスタ)
r0 ~ r12 (ARMのレジスタは0~12)
sp (stack pointer)
lr (link regester)
pc (program counter)
cpsr (cpu statusのレジスタ)

(gdb)disass
# disassemblerの略

(gdb)x/20x $pc+48+4
0x100030
# entry.Sのr1,=edata_entryやend_entryが$pc+48に変わってる
# pcは次の命令を指している10030->10038だから48からさらに+4

(gdb)p (void*) edata_entry
$1 = (void *0) 0x10588 <edata_entry>
# 10588がレジスタのr1に入る

(gdb)stepi
(gdb)info registers
r1 0x10588 66952

(gdb)stepi
(gdb)info registers
r1  0x10588 66952
r2  0x19000 102400

(gdb)define ni
>x/1i $pc
>x/1i $pc
>end
# こうするとentry.Sを1命令ずつ実行できる


(gdb)si
0x0001000c in _start ()
=> 0x1000c <_start+12>: cmp r1,r2

(gdb)si
0x0001000c in _start ()
=> 0x1014c <_start+16>: cmp r1,{r3}
```

```
(gdb)disass
=>  0x00010014 <*20>: blt
    0x00010018 <*24>: msr
# blt でループしてることがわかる

(gdb)tb *0x00010018
(gdb)c
# bltのループを抜ける

(gdb)disass
(gbd)info registers
r1  0x19000 102400
r2  0x19000 102400
# r2の値になるまでr1をincrementすることでループを抜けてる事がわかる
# 19000までmemoryがある

# CPUのコントロール


(gdb)si
(gdb)info registers
cpsr    0x600001d3

# entry.Sの MSRの行を実行
(gdb)si
(gdb)info registers
cpsr    0xd3
# CPUのコントロールレジスタを指定している
# supervisorじゃないとアクセスできない

# entry.Sの LDR の行を実行
(gdb) si
(gdb) info registers
sp  0x12000
# 0x0 -> 0x12000
# stack ponterが設定されたのでサブルーチンコールが使えるようになる
# サブルーチンコール 戻り先をstackに覚えておいてそこに飛ぶ
# lr(link register)に覚えさせる事で1回だけアクセスせずに飛ぶ事ができる
# BL breach & link


(gdb)si
(gdb)info registers
lr  0x10024

(gdb)disass _start
    0x00010024 <+36>: b
```

```
# startまできたのでここ以降はstart.c
(gdb)l
# cなので  next が使える

(gdb)next
(gdb)next
(gdb)s
set_bootpgtbl(... lent=1048576 ...)
(gdb)p (void*) len
$2 = (void *) 0x100000
# 1048576 は16進数で100000

(gdb)disass
```

- memory に直接 load できる値は 長い値だとmemory一旦置かないといけない
- 一番最初に呼び出すのは C で書かれたファイルじゃなくてアセンブラ
    - entry.S で行なっている

entry.S

```
_start:
    LDR     r1, =edata_entry
    LDR     r2, =end_entry
    MOV     r3, #0x00

# BLT までをループ
1:
    CMP     r1, r2
    STMLTIA r1!, {r3}  # {}はレジスタのリスト r1にセーブする
    # !は セーブした数だけr1を進める(++とかと一緒)
    # Memclear とかと一緒

    BLT     1b



    # initialize stack pointers for svc modes
    # CPUのコントロールレジスタを指定
    # cpsr 0xd3
    MSR     CPSR_cxsf, #(SVC_MODE|NO_INT)
    LDR     sp, =svc_stktop

    BL      start
    B .
```

- B . まで行くとそこを永遠とループする

- ARM fault持ってるので Bではなくfaultにすべき

start.c

l.166~

```
void start (void)
{
        uint32  vectbl;
    _puts("starting xv6 for ARM...\n");

    // double map the low memory, required to enable paging
    // we do not map all the physical memory
    set_bootpgtbl(0, 0, INIT_KERNMAP, 0);
    set_bootpgtbl(KERNBASE, 0, INIT_KERNMAP, 0);
```

l.69~

```
    for (idx = 0; idx < len; idx++) {
        pde = (phy << PDE_SHIFT);

        if (!dev_mem) {
            // normal memory, make it kernel-only, cachable, bufferable
            pde |= (AP_KO << 10) | PE_CACHE | PE_BUF | KPDE_TYPE;
            // |(or) を使って足して行く
        } else {
            // device memory, make it non-cachable and non-bufferable
            pde |= (AP_KO << 10) | KPDE_TYPE;
        }
```

- ARMv6 page table entry https://developer.arm.com/docs/ddi0211/h/memory-management-unit/hardware-page-table-translation/armv6-page-table-translation-subpage-ap-bits-disabled