xv6© -rev10
(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)
Entering the Kernel, main

Carmi Merimovich

Tel-Aviv Academic College

September 18, 2017

Where are we?

1. POWERUP.
   - The Boot processor executes instructions. MMU inactive.
2. ROM code loades 512-bytes boot block to address 0x07C0 and up.
   - ROM terminates by JMPing to address 0x07C0.
3. Boot block code loads the kernel from ide0 into 0x00100000 and up.
   - Boot block code terminates by JMPing into the kernel's entry point.
   - The entry point address is found at a fixed location in the kernel ELF.
   - The entry point code is in Assembly and is labeled `entry`.

entry's aim

`entry`

sets up a temporary kernel programming model.

- MMU is activated with a table coding the following translation:

$$[0x80000000, 0x803FFFFF] \mapsto [0x00000000, 0x003FFFFF]$$

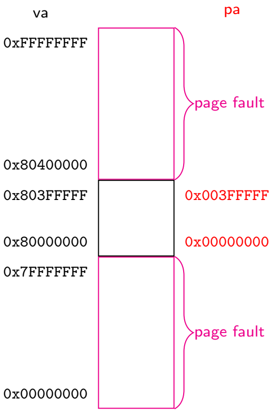
- `esp` is set to the end of a 4KB buffer.
- `entry` finishes by JMPing into `main`.

The REAL entry point on the primary processor is **_start/entry**

entry does the following:

- Builds a temporary page table which maps 4MB of virtual addresses begining at 0x80000000 to 4MB of physical address begining at 0x00000000.
- Note the kernel was loaded at physical address 0x0010000 and the programming model is the kernel begins at 0x80100000.
- Starts the paging unit with the above constructed table.
- Points register **esp** to a 4KB stack area.
- (This would be the kernel stack of the primary processor.)
- Jumps to **main**.

Memory after **entry** execution



Recall

- Virtual address in the eye of the paging unit:

31 … 12	11 … 0
vp#	offset

- Physical address in the eye of the paging unit:

31 … 12	11 … 0
pp#	offset

- Page table entry:

31 … 12	11 …	2	1	0
pp#		u	w	p

Setting up the temporary translation table

The 4MB translation

- 4MB range : 0x00000000-0x003FFFFF.
- In ranges of 1MB:

$$\mathtt{0x800xxxxx} \mapsto \mathtt{0x000xxxxx}$$
$$\mathtt{0x801xxxxx} \mapsto \mathtt{0x001xxxxx}$$
$$\mathtt{0x802xxxxx} \mapsto \mathtt{0x002xxxxx}$$
$$\mathtt{0x803xxxxx} \mapsto \mathtt{0x003xxxxx}$$

- In ranges of 4KB:

$$\mathtt{0x80000xxx} \mapsto \mathtt{0x00000xxx}$$
$$\mathtt{0x80001xxx} \mapsto \mathtt{0x00001xxx}$$
$$\mathtt{0x80002xxx} \mapsto \mathtt{0x00002xxx}$$
$$\mathtt{0x80003xxx} \mapsto \mathtt{0x00003xxx}$$
$$\vdots$$

The rule: 0x800xxxxx ↦ 0x000xxxxx

- Translating to pages we have: 0x800xx ↦ 0x000xx.
- In table form:

vp#		pp#	
dec	hex	hex	dec
524288	80000	00000	0
524289	80001	00001	1
⋮	⋮	⋮	⋮
524541	800FE	000FE	254
524542	800FF	000FF	255

- Coding it (for an **hypothetical** linear page table):

```
for (i = 0; i < 256; i++)
 pgtbl[0x80000+i] = (i << 12) | 3;
```

The rule: $\texttt{0x801xxxxx} \mapsto \texttt{0x001xxxxx}$

- The rule for pages is: $\texttt{0x801xx} \mapsto \texttt{0x001xx}$.
- In table form:

vp#		pp#	
dec	hex	hex	dec
524544	80100	00100	256
524545	80101	00101	257
⋮	⋮	⋮	⋮
524798	801FE	001FE	510
524799	801FF	001FF	511

- Coding it (for linear page table):

```
for (i = 0x100; i < 0x200; i++)
 pgtbl[0x80000+i] = (i << 12) | 3;
```

Hypothetical code for the four rules

```
for (i = 0x000; i < 0x100; i++)
  pgtbl[0x80000+i] = (i << 12) | 3;
for (i = 0x100; i < 0x200; i++)
  pgtbl[0x80000+i] = (i << 12) | 3;
for (i = 0x200; i < 0x300; i++)
  pgtbl[0x80000+i] = (i << 12) | 3;
for (i = 0x300; i < 0x400; i++)
  pgtbl[0x80000+i] = (i << 12) | 3;
```

Which is equivalent to the following.

```
for (i = 0; i < 1024; i++)
  pgtbl[0x80000+i] = (i << 12) | 3;
```

Real page table is hierarchical

- The real page table is broken into pieces.

31 — 22	21 — 12	11 — 0
$\mathtt{i}_0$	$\mathtt{i}_1$	offset

virtual address as viewd by the paging unit

- For each index $\mathtt{0x80000} + \mathtt{i}$ we have:
  - Piece number: $(\mathtt{0x80000} + \mathtt{i})/\mathtt{1024}$.
  - Index in the piece: $(\mathtt{0x80000} + \mathtt{i})\%\mathtt{1024}$.
- In computer lingo, for each index $\mathtt{0x80000} + \mathtt{i}$ we have:
  - Piece number: $(\mathtt{0x80000} + \mathtt{i}) >> \mathtt{10}$.
  - Index in the piece: $(\mathtt{0x80000} + \mathtt{i})\&\mathtt{1023}$.

Pieces in binary

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
8	0	0	x	x	x	x	x

31									22	21									12	11											0
1	0	0	0	0	0	0	0	0	0	0	0	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
8	0	1	x	x	x	x	x

31									22	21									12	11											0
1	0	0	0	0	0	0	0	0	0	0	1	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y

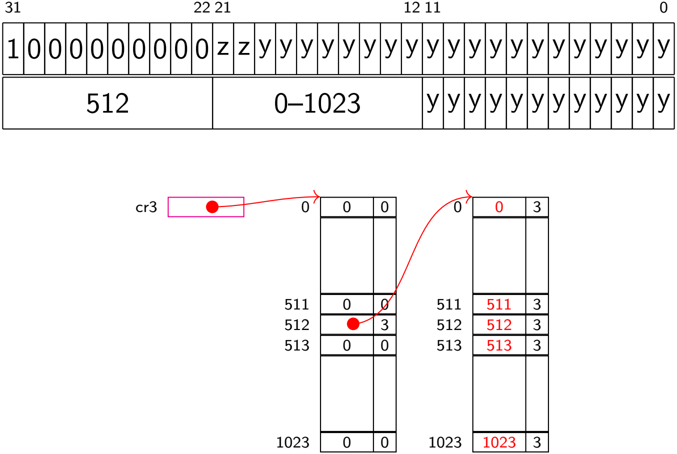
31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
8	0	2	x	x	x	x	x

31									22	21									12	11											0
1	0	0	0	0	0	0	0	0	0	1	0	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
8	0	3	x	x	x	x	x

31									22	21									12	11											0
1	0	0	0	0	0	0	0	0	0	1	1	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y

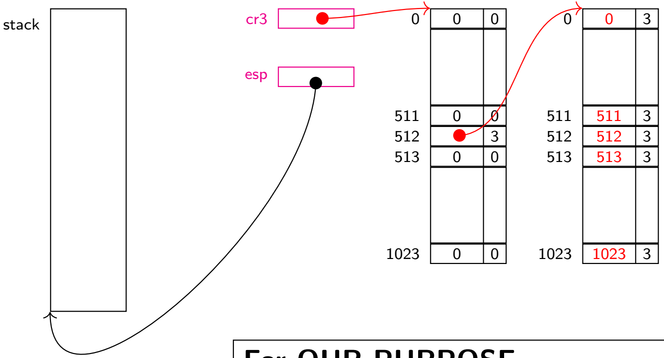
Pieces of 0x80000 + i (i < 1024)



Coding for real table

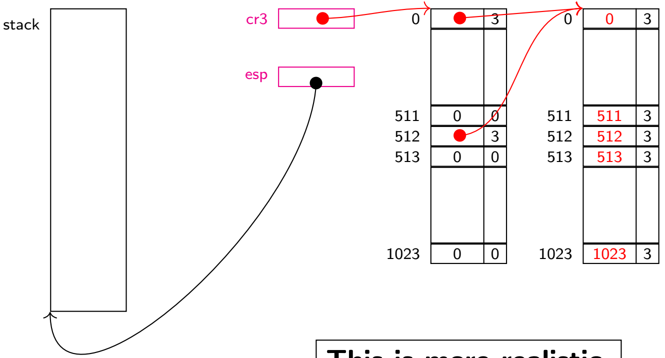
```
static int
    pgdir[1024] __attribute__ (aligned(4096)),
    pgtbl[1024] __attribute__ (aligned(4096));

memset(pgdir, 4096, 0);
pgdir[512] = pgtbl | 3;
for (i = 0; i < 1024; i++) {
 pgtbl[i] = (i <<12) | 3;
}
lcr3(pgdir);
lcr0(rcr0() | CR0_PG);
// Paging has gone active
(Spectacular crash)
```

state when entering `main`



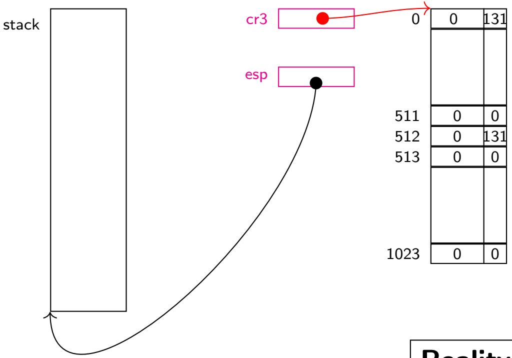
For OUR PURPOSE,
this is a good enough description,
eventhough it is not the reality.

state when entering `main` (elective)



This is more realistic.

state when entering `main` (elective)



Reality.

entry.s (elective)

```
.globl _start
_start = V2P_WO(entry)
.globl entry
entry:
 movl %cr4,%eax
 orl $CR4_PSE,%eax
 movl %eax, %cr4
 movl $V2P_WO(entrypgdir),%eax
 movl %eax, %cr3
 movl %cr0, %eax
 orl $CR0_PG|CR0_WP,%eax
 movl %eax,%cr0
```

```
movl $stack+KSTACKSIZE,%esp

mov $main,%eax
jmp *%eax

.comm stack,KSTACKSIZE
```

main

Since we got here from **entry**, this code is run by the primary processor.

- **main** calls routines initializing the different subsystems.
- We study each initialization together with its subsystem.
- Most subsystems need one initialization per (computer) system.
- Initialization which are needed per-cpu are done in **mpmain**.

The auxiliary processors begins at **entryother**, then proceed to **mpenter**.

xv6 kernel programming model

- Like for user mode, there is an **xv6** programing model.
- For user mode, the kernel is the maintainer of the model.
- The kernel is the model maintainer for the kernel programming.
- The kernel is trustable, so the model can include a “don’t do” rules

The xv6 kernel programming model

- The kernel is linked into virtual address 0x80100000 and up.
- The code can freely use the general purpose registers: **eax**, **ecx**, **edx**, **ebx**, **esi**, **edi**, **ebp**.
- There is a 4KB stack for the kernel to use. No messing up with it!
- Registers **cs**, **ds**, **ss**, **es** should not be modified.
- (the base and limit are assumed to be zero and max in the above registers).

main

```c
int main(void) {
  kinit1(end,
        P2V(4*1024*1024));
  kvmalloc();
  mpinit();
  lapicinit();
  seginit();
  picinit();
  ioapicinit();
  consoleinit();
  uartinit();
  pinit();
  tvinit();
  binit();
  fileinit();
  iinit();
  ideinit();
  if (!ismp)timerinit();
  startothers();
  kinit2(P2V(4*1024*1024),
              P2V(PHYSTOP));
  userinit();
  mpmain();
}
```

main

Since we got here from **entry**, this code is run by the Boot processor.

- **main** calls routines initializing the different subsystems.
- We study each initialization together with its subsystem.
- Most subsystems need one initialization per (computer) system.
- Initialization which are needed per-cpu are done in **mpmain**.

The application processors begins at **entryother**, then proceed to **mpenter**.

Starting auxiliary processors

In **startothers()**, for each application processor the boot processor executes the following:

1. Copies the **entryother** code into address 0x80007000.
    - **entryother** is a replacement of the **entry** routine.
2. Through the lapic instructs the AP to start executing at 0x0700.
3. Waits for the AP be done with initialization.

Application processor initialization

1. `entryother` sets up a temporary kernel programming model.
   - MMU is activated with a table coding the following translation:

$$[\mathtt{0x80000000}, \mathtt{0x803FFFFF}] \mapsto [\mathtt{0x00000000}, \mathtt{0x003FFFFF}]$$

   - `esp` is set to the end of a 4KB buffer.
   - `entryother` finishes by JMPing into `mpenter`.

```
static void mpenter(void) {
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
```

The initializations we study now

```
kinit1(end, P2V(4*1024*1024)); // phys page alloca
kvmalloc();          // kernel page table
   ⋮
seginit();           // set up segments
   ⋮
pinit();             // process table
   ⋮

   ⋮
kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must co
userinit();          // first user process
mpmain();
```