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

Carmi Merimovich

Tel-Aviv Academic College

September 18, 2017

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

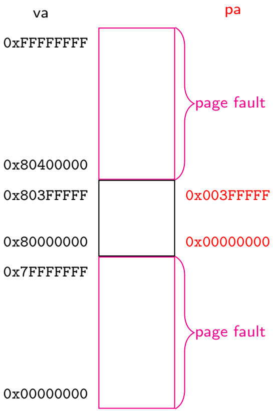
- The kernel is linked into virtual address 0x80100000 and up.
- The code can freely use the general purpose registers: **eax**, **ecx**, **edx**, **ebx**, **esi**, **edi**, **ebp**.
- There is a 4KB stack for the kernel to use. No messing up with it!
- Registers **cs**, **ds**, **ss**, **es** should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

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

entry does the following:

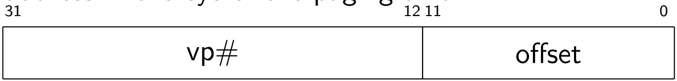
- Builds a temporary page table which maps 4MB of virtual addresses begining at 0x80000000 to 4MB of physical address begining at 0x00000000.
- Note the kernel was loaded at physical address 0x0010000 and the programming model is the kernel begins at 0x80100000.
- Starts the paging unit with the above 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



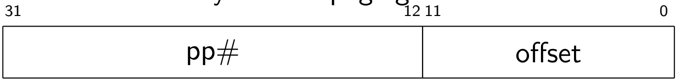
Recall

- Virtual address in the eye of the paging unit:



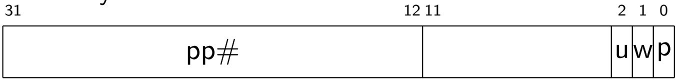
31 … 12	11 … 0
vp#	offset

- Physical address in the eye of the paging unit:



31 … 12	11 … 0
pp#	offset

- Page table entry:



31 … 12	11 … 3	2	1	0
pp#		u	w	p

Setting up the temporary translation table

The 4MB translation

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

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

- In ranges of 4KB:

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

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: 0x801xxxxx ↦ 0x001xxxxx

- The rule for pages is: 0x801xx ↦ 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

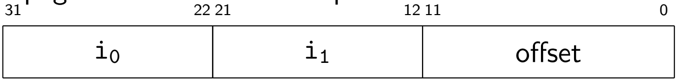
```
for (i = 0x000; i < 0x100; i++)
 pgtbl[0x80000+i] = (i << 12) | 3;
for (i = 0x100; i < 0x200; i++)
 pgtbl[0x80000+i] = (i << 12) | 3;
for (i = 0x200; i < 0x300; i++)
 pgtbl[0x80000+i] = (i << 12) | 3;
for (i = 0x300; i < 0x400; i++)
 pgtbl[0x80000+i] = (i << 12) | 3;
```

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

$\texttt{i}_0$ $\texttt{i}_1$ offset

virtual address as viewd by the paging unit

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

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

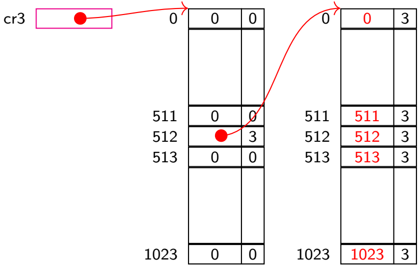
31									22	21									12	11											0
1	0	0	0	0	0	0	0	0	0	1	0	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y

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)

31									22	21									12	11											0
1	0	0	0	0	0	0	0	0	0	z	z	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
512										0–1023										y	y	y	y	y	y	y	y	y	y	y	y

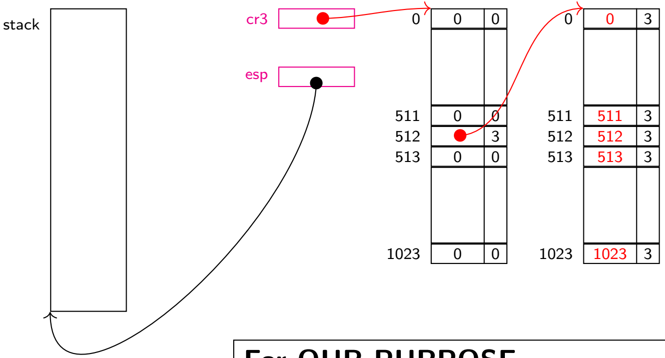


Coding for real table

```
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`

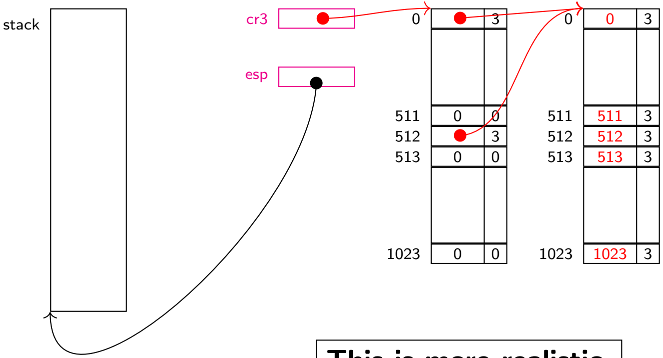


0	0	0
511	0	0
512	●	3
513	0	0
1023	0	0

0	0	3
511	511	3
512	512	3
513	513	3
1023	1023	3

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

state when entering `main` (elective)

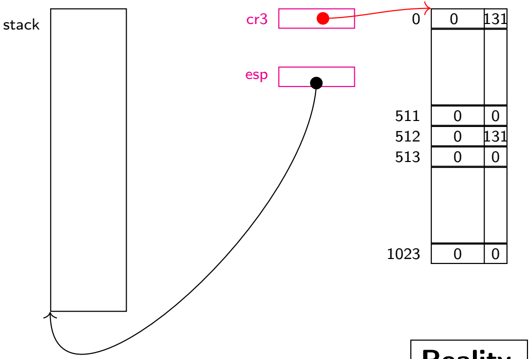


0	●	3
511	0	0
512	●	3
513	0	0
1023	0	0

0	0	3
511	511	3
512	512	3
513	513	3
1023	1023	3

This is more realistic.

state when entering `main` (elective)



0	0	131
511	0	0
512	0	131
513	0	0
1023	0	0

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**.

main

```
1216 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();
}
```

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();
```