ICS143A: Principles of Operating Systems

Lecture 12: Starting other CPUs

Anton Burtsev
February, 2017

Starting other CPUs

We're back to main()

```
main(void)
{
...
  startothers(); // start other
                    processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
  userinit(); // first user process
  mpmain();
}
```

Starting other CPUs

- Copy start code in a good location
  - 0x7000 (remember same as the one used by boot loader)
- Pass start parameters on the stack
  - Stack for a high-address kernel
    - Each CPU allocates a page from a physical allocator
  - Entry point (mpenter())
  - Two entry page table
    - To do the low to high address switch

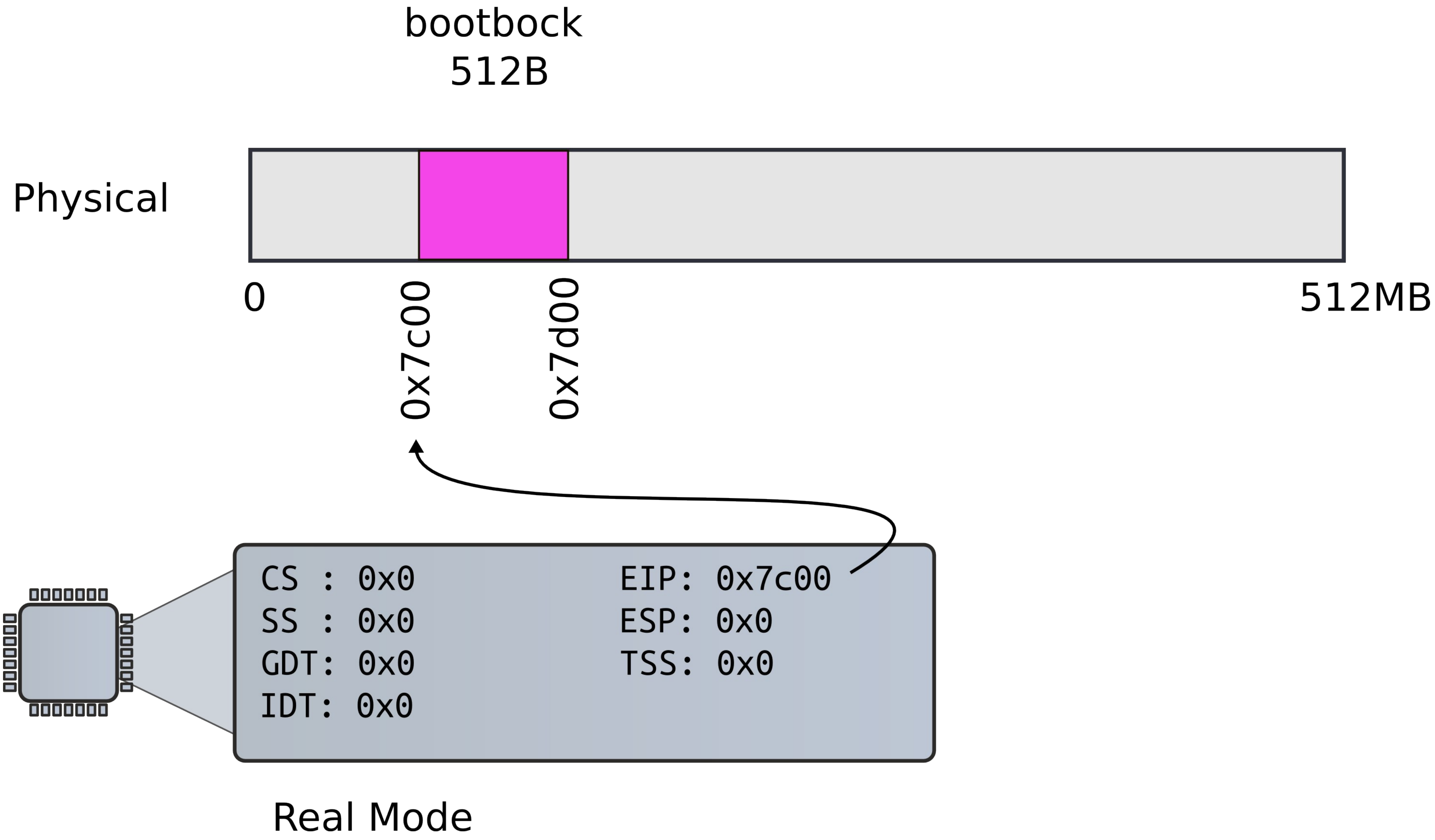
Start other CPUs

```
startothers(void)
{
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start,
          (uint)_binary_entryother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum()) // We've started already.
      continue;
...
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void**)(code-8) = mpenter;
    *(int**)(code-12) = (void *) v2p(entrypgdir);

    lapicstartap(c->id, v2p(code));
```

We could choose any address, but 0x7c00 is convenient



```
CS : 0x0            EIP: 0x7c00
SS : 0x0            ESP: 0x0
GDT: 0x0            TSS: 0x0
IDT: 0x0
```

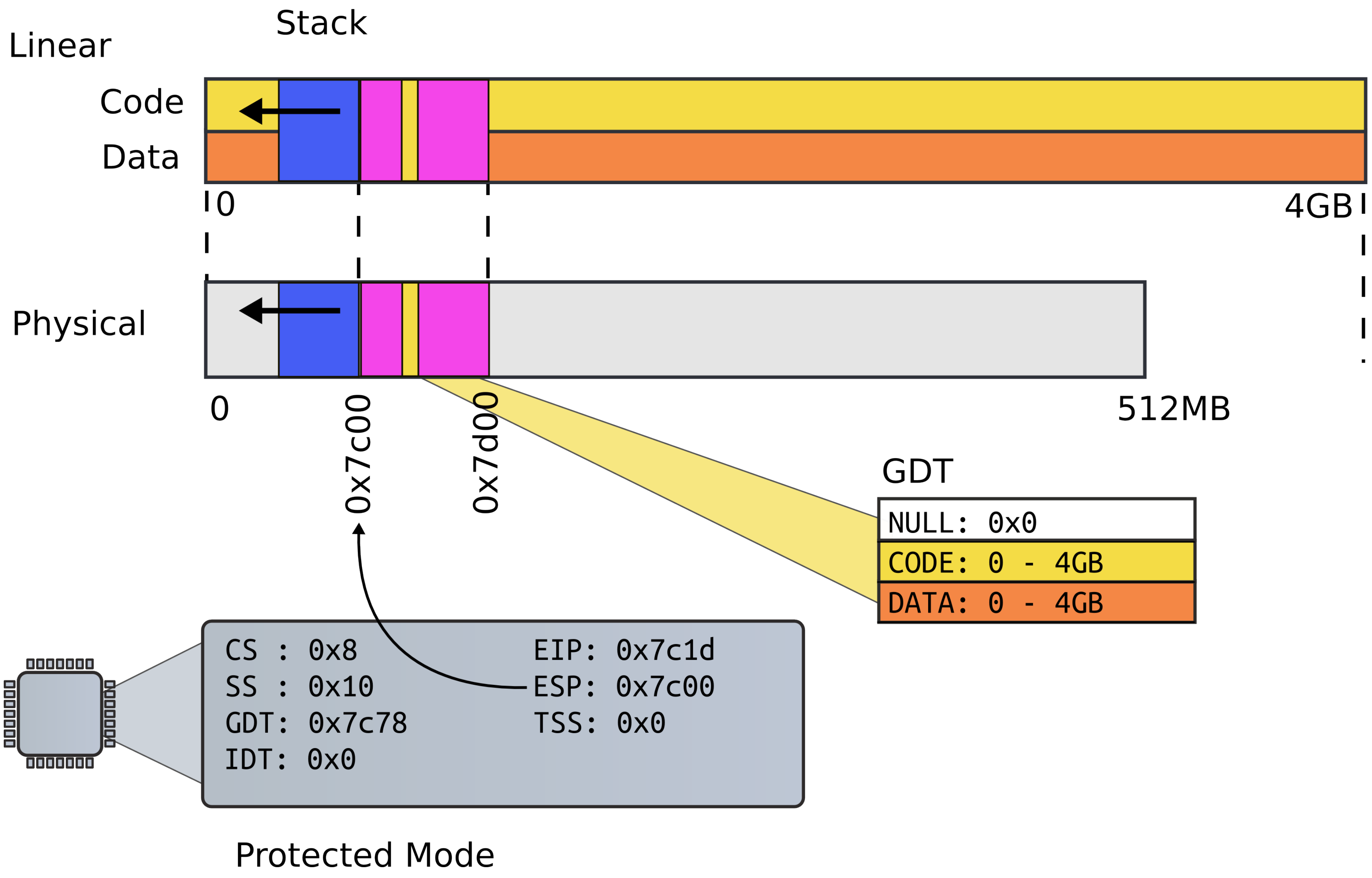
Start other CPUs

```
startothers(void)
{
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start,
         (uint)_binary_entryother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum()) // We've started already.
      continue;
...
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void**)(code-8) = mpenter;
    *(int**)(code-12) = (void *) v2p(entrypgdir);

    lapicstartap(c->id, v2p(code));
```

Recap(): First stack



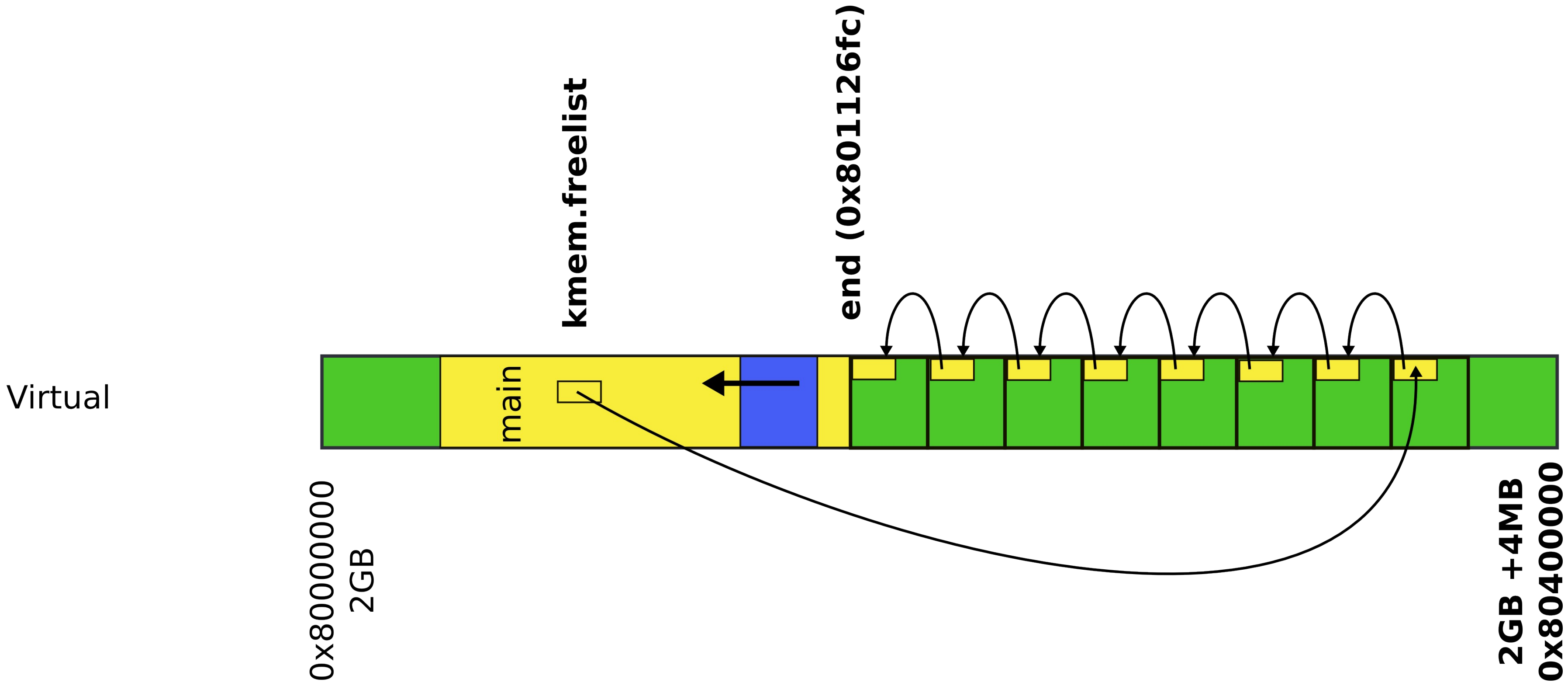
GDT

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB

```
CS : 0x8          EIP: 0x7c1d
SS : 0x10         ESP: 0x7c00
GDT: 0x7c78       TSS: 0x0
IDT: 0x0
```

Protected Mode

Recap: kalloc() – allocate page



```
CS : 0x8          EIP: main
SS : 0x10         ESP: stack
GDT: 0x7c78       TSS: 0x0
IDT: 0x0          CR3: entrypgdir
```

Protected Mode

Start other CPUs

```
startothers(void)
{
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start,
          (uint)_binary_entryother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum()) // We've started already.
      continue;
...
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void**)(code-8) = mpenter;
    *(int**)(code-12) = (void *) v2p(entrypgdir);

    lapicstartap(c->id, v2p(code));
```

Start other CPUs

```
startothers(void)
{
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start,
          (uint)_binary_entryother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum()) // We've started already.
      continue;
...
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void**)(code-8) = mpenter;
    *(int**)(code-12) = (void *) v2p(entrypgdir);

    lapicstartap(c->id, v2p(code));
```

entryother.S

```
.code16
.globl start
start:
cli

xorw %ax,%ax
movw %ax,%ds
movw %ax,%es
movw %ax,%ss

```

- Disable interrupts
- Init segments with 0

entryother.S

```
lgdt gdtdesc
movl %cr0, %eax
orl $CR0_PE, %eax
movl %eax, %cr0
ljmpl $(SEG_KCODE<<3), $(start32)

.code32
start32:
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
movw %ax, %ss
movw $0, %ax
movw %ax, %fs
movw %ax, %gs
```

- Load GDT
- Switch to 32bit mode
  - Long jump to start32
- Load segments

```
# Turn on page size extension for 4Mbyte pages
movl %cr4, %eax
orl $(CR4_PSE), %eax
movl %eax, %cr4
# Use enterpgdir as our initial page table
movl (start-12), %eax
movl %eax, %cr3
# Turn on paging.
movl %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl %eax, %cr0

# Switch to the stack allocated by startothers()
movl (start-4), %esp
# Call mpenter()
call *(start-8)
```

entryother.S

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

Init segments

```
seginit(void)
{
  struct cpu *c;
...
  c = &cpus[cpunum()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0x8000000, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);

  // Map cpu, and curproc
  c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);

  lgdt(c->gdt, sizeof(c->gdt));
  loadgs(SEG_KCPU << 3);

  // Initialize cpu-local storage.
  cpu = c;
  proc = 0;
}
```

Per-CPU variables

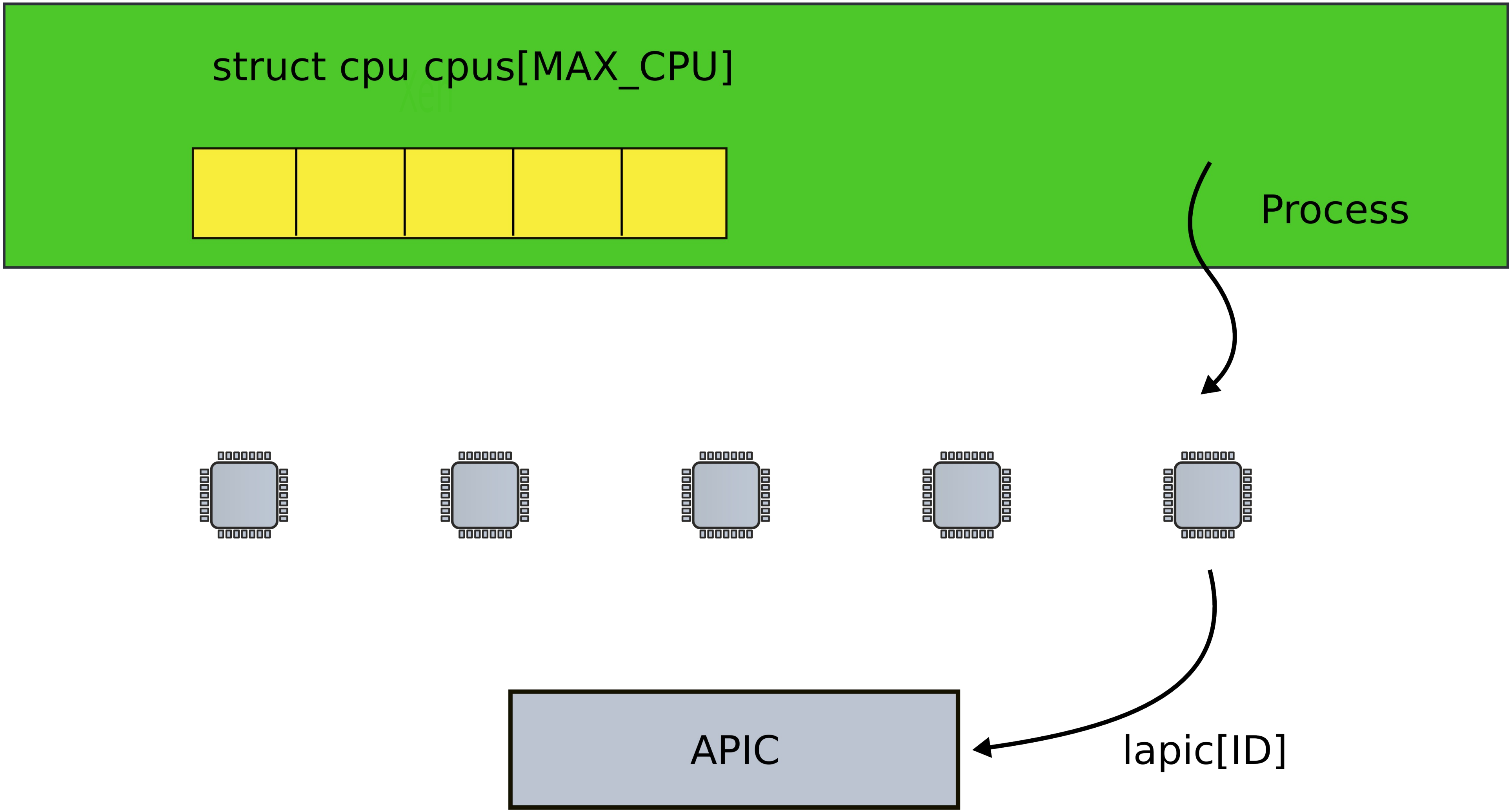
- Variables private to each CPU

Per-CPU variables

- Variables private to each CPU
  - Current running process
  - Kernel stack for interrupts
    - Hence, TSS that stores that stack

```
extern struct cpu cpus[NCPU];
```

One catch: lapic[id] is slow

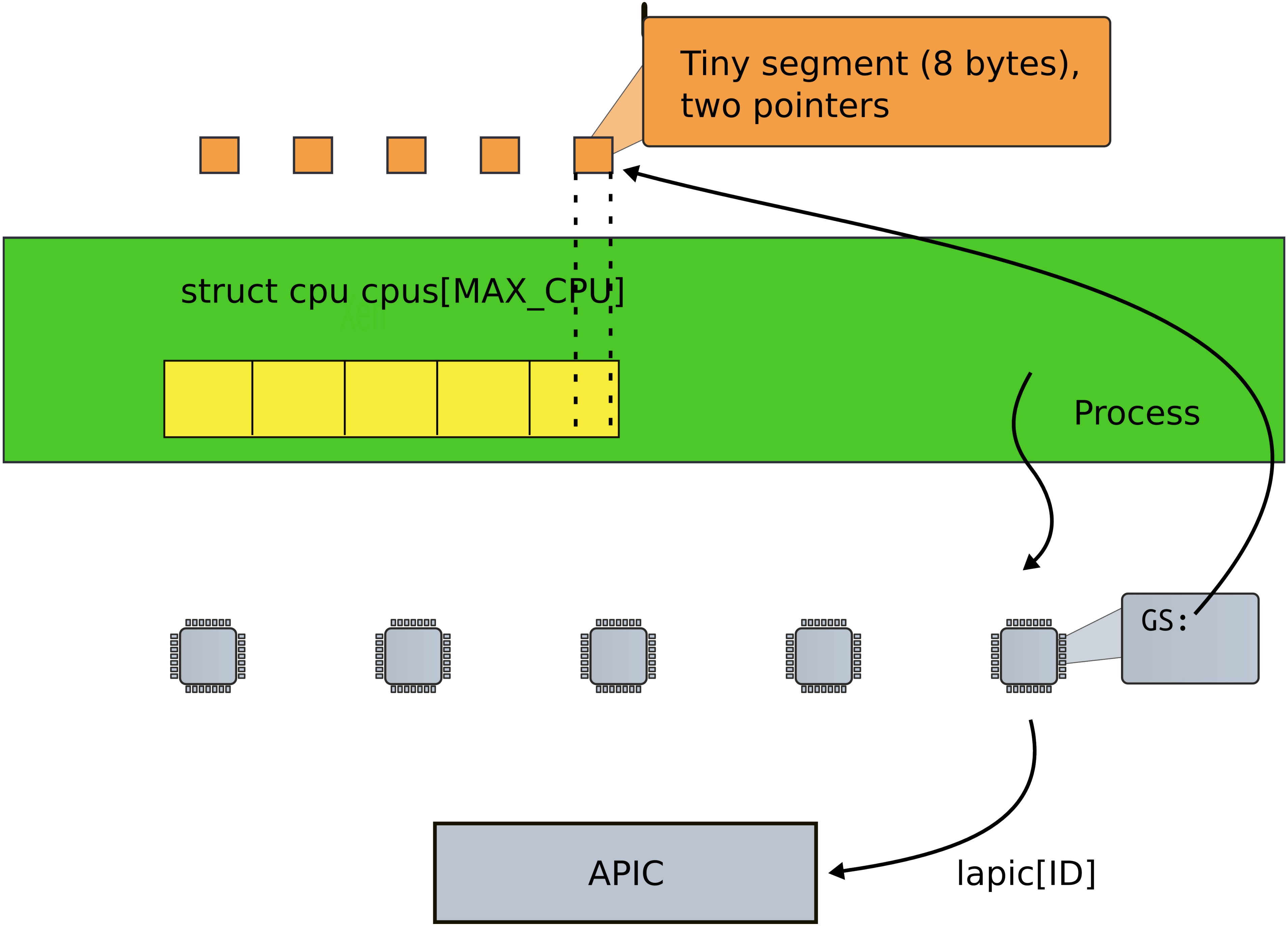


We need to save id of a CPU on each CPU

- We can use a register ...

We need to save id of a CPU on each CPU

- We can use a register …
    - But it's wasteful



Tiny segment (8 bytes), two pointers

struct cpu cpus[MAX_CPU]

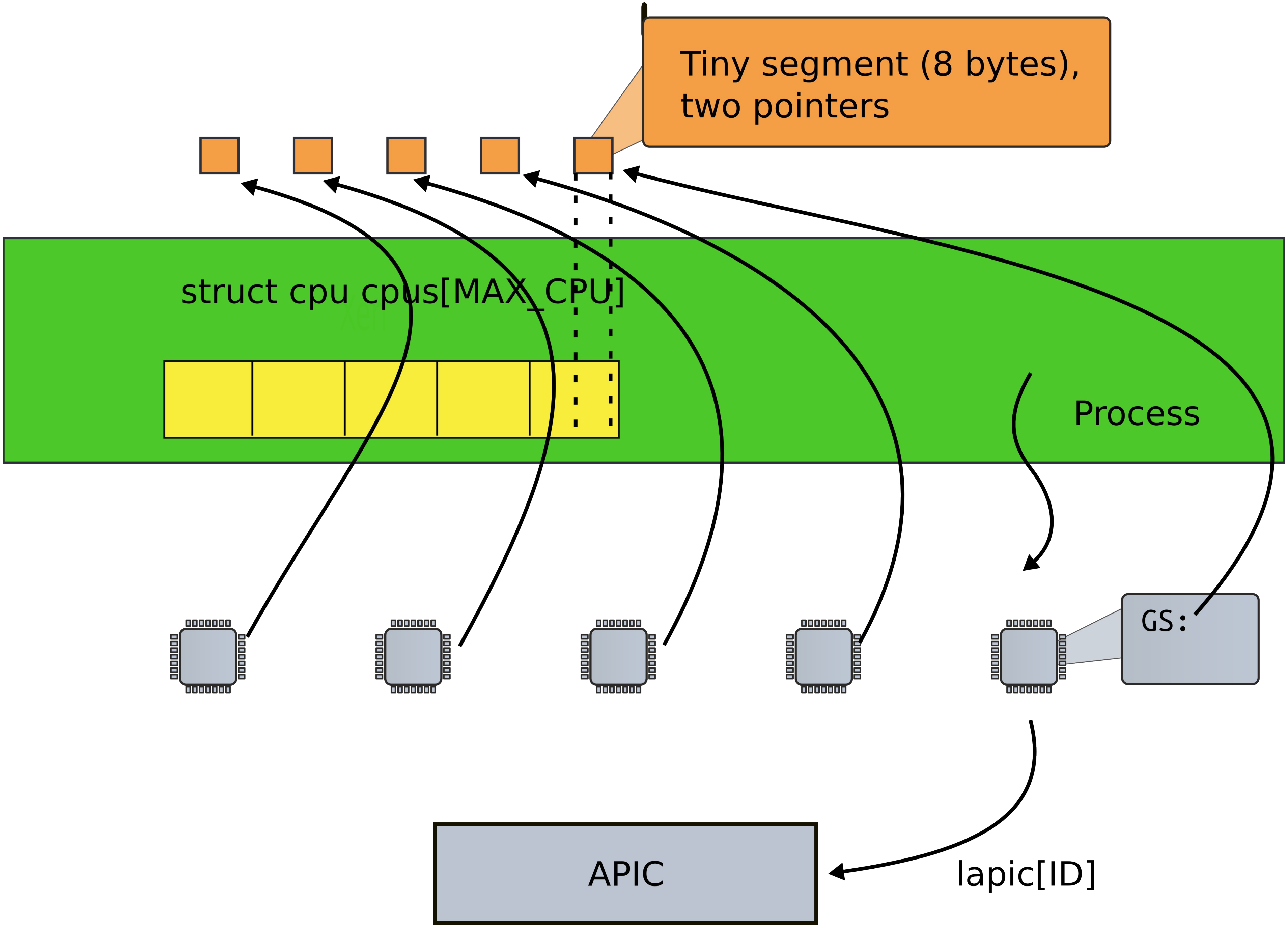
Xen

Process

GS:

APIC

lapic[ID]



Tiny segment (8 bytes), two pointers

struct cpu cpus[MAX_CPU]

Process

GS:

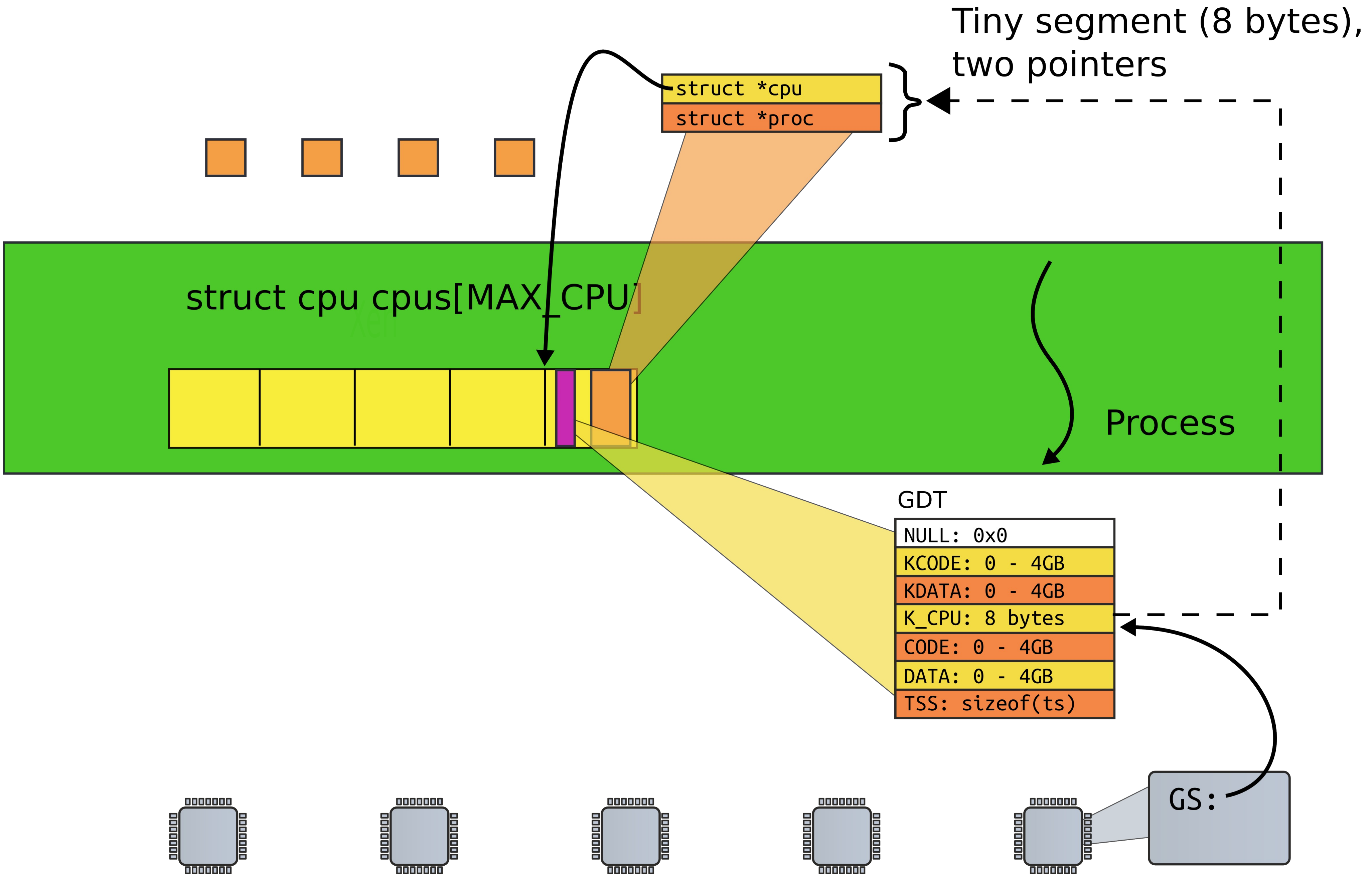
APIC

lapic[ID]

```c
// Per-CPU state
struct cpu {
  uchar apicid; // Local APIC ID
  struct context *scheduler; // swtch() here to enter scheduler
  struct taskstate ts; // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS]; // x86 global descriptor table
  volatile uint started; // Has the CPU started?
  int ncli; // Depth of pushcli nesting.
  int intena; // Were interrupts enabled before pushcli?

  // Cpu-local storage variables; see below
  struct cpu *cpu;
  struct proc *proc; // The currently-running process.
};
```

```
// Per-CPU state
struct cpu {
  uchar apicid; // Local APIC ID
  struct context *scheduler;
  struct taskstate ts;
  struct segdesc gdt[NSEGS]; // x86 global descriptor table
...
  // Cpu-local storage variables; see below
  struct cpu *cpu;
  struct proc *proc; // The currently-running process.
};
...
extern struct cpu *cpu asm("%gs:0"); // &cpus[cpunum()]
extern struct proc *proc asm("%gs:4"); // cpus[cpunum()].proc
```



Tiny segment (8 bytes),
two pointers

struct *cpu
struct *proc

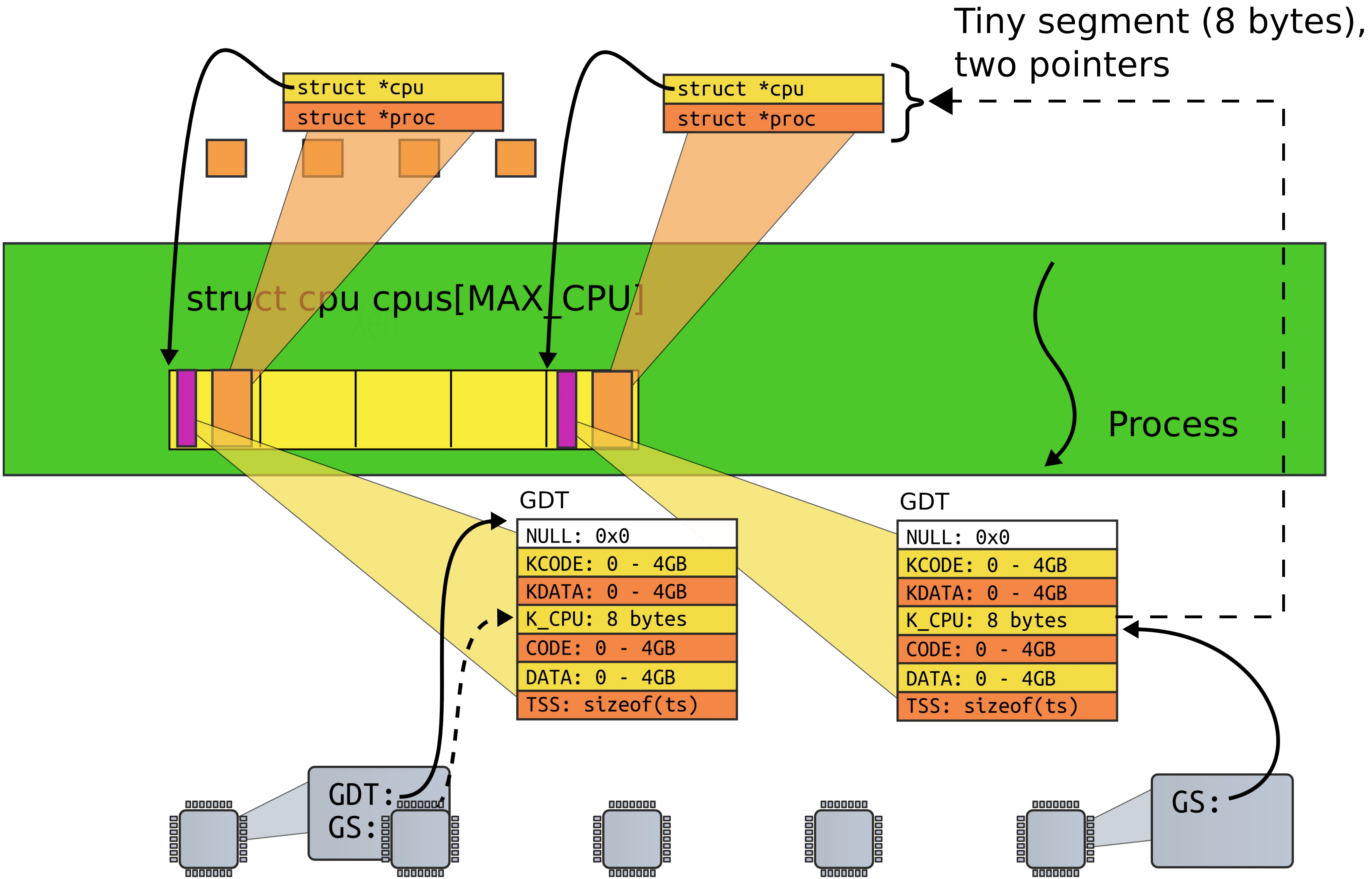
struct cpu cpus[MAX_CPU]

Process

GDT

NULL: 0x0
KCODE: 0 - 4GB
KDATA: 0 - 4GB
K_CPU: 8 bytes
CODE: 0 - 4GB
DATA: 0 - 4GB
TSS: sizeof(ts)

GS:



Tiny segment (8 bytes), two pointers

struct *cpu
struct *proc

struct *cpu
struct *proc

struct cpu cpus[MAX_CPU]

Process

GDT

NULL: 0x0
KCODE: 0 - 4GB
KDATA: 0 - 4GB
K_CPU: 8 bytes
CODE: 0 - 4GB
DATA: 0 - 4GB
TSS: sizeof(ts)

GDT

NULL: 0x0
KCODE: 0 - 4GB
KDATA: 0 - 4GB
K_CPU: 8 bytes
CODE: 0 - 4GB
DATA: 0 - 4GB
TSS: sizeof(ts)

GDT:
GS:

GS:

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

```
// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting\n", cpu->id);
  idtinit(); // load idt register
  xchg(&cpu->started, 1);
  scheduler(); // start running processes
}
```

Thank you