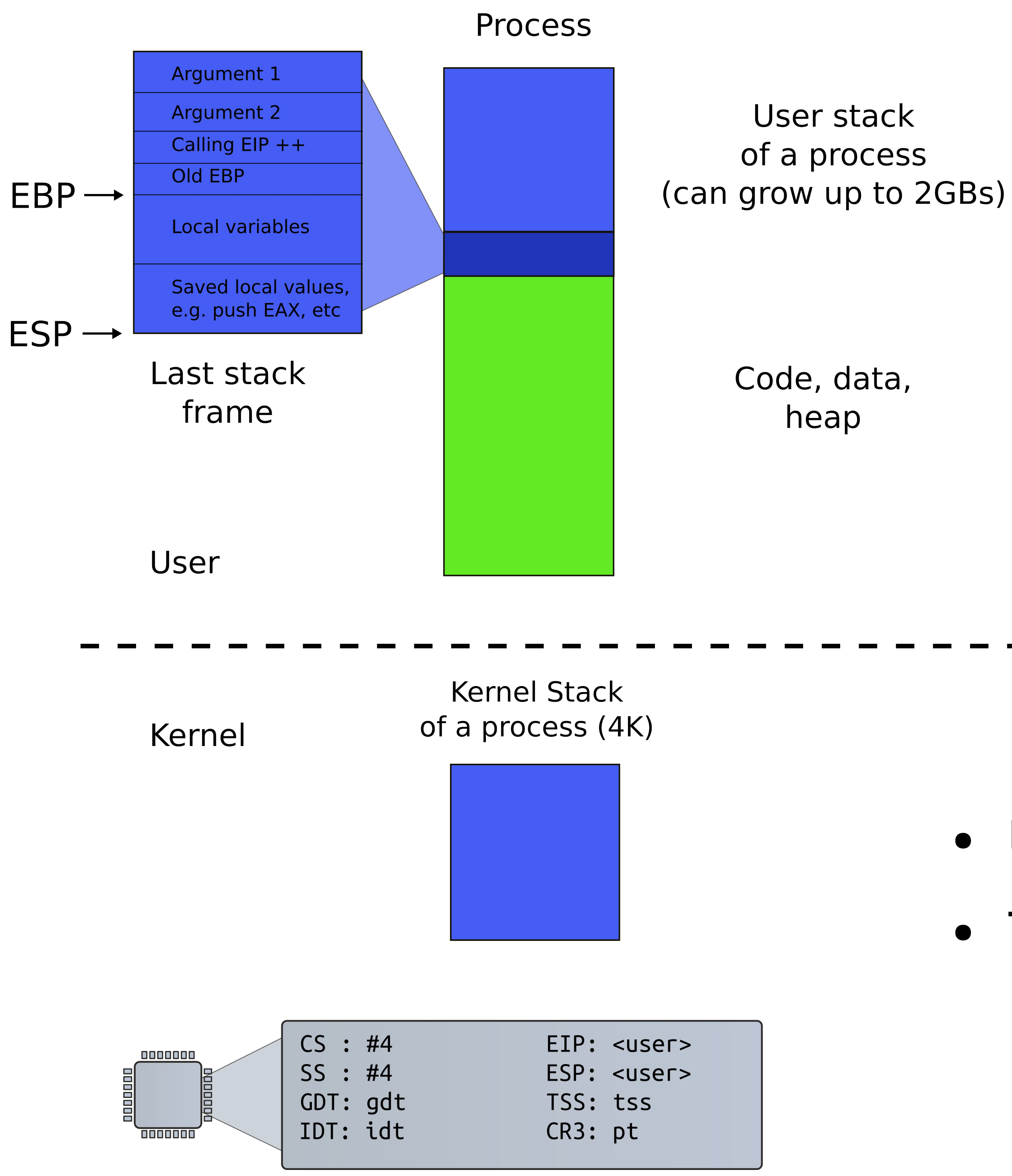
ICS143A: Principles of Operating Systems

Lecture 13: Context switch

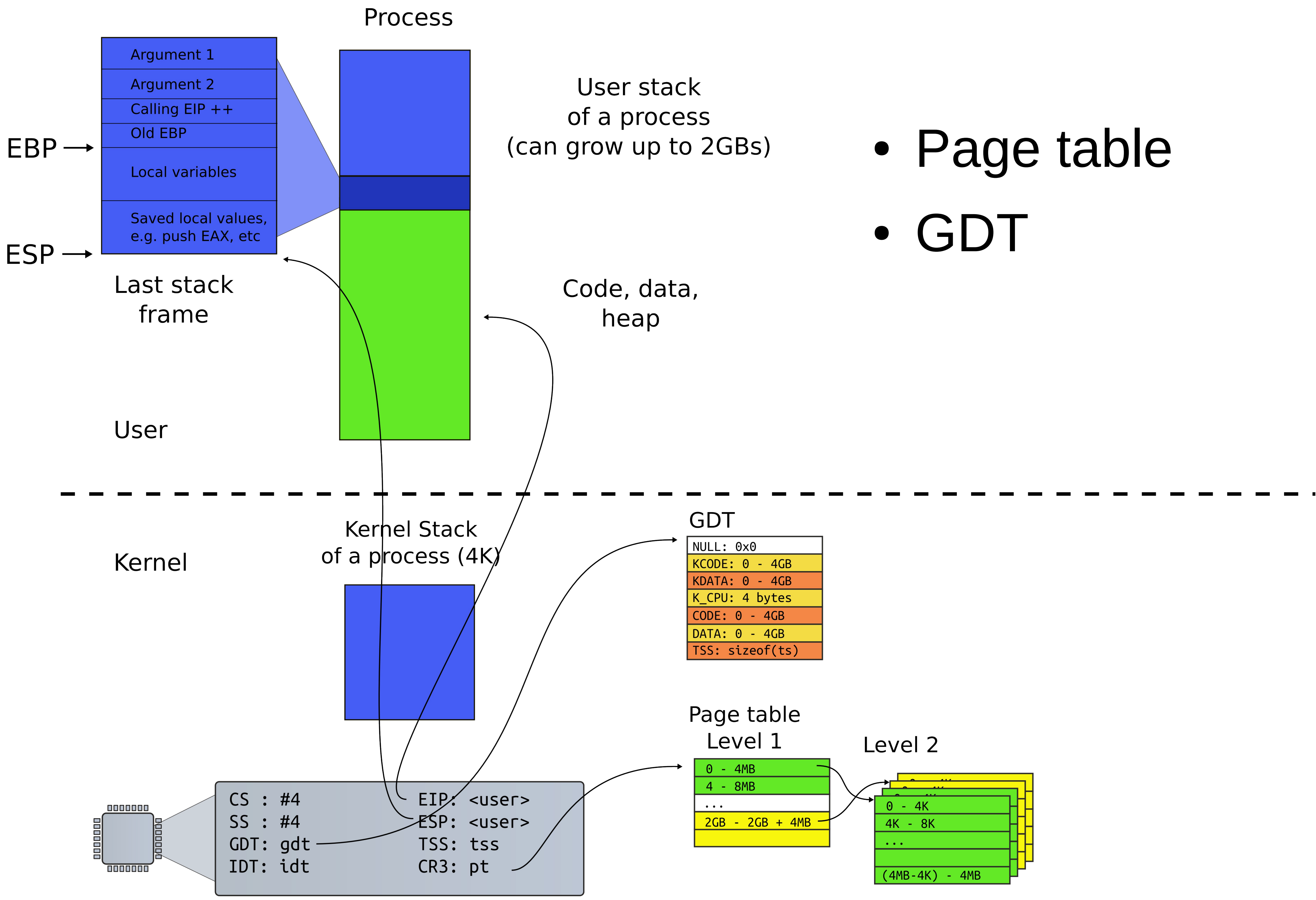
Anton Burtsev

November, 2017


Process
Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc
EBP
ESP
Last stack frame
User stack of a process (can grow up to 2GBs)
Code, data, heap
User
Kernel
Kernel Stack of a process (4K)

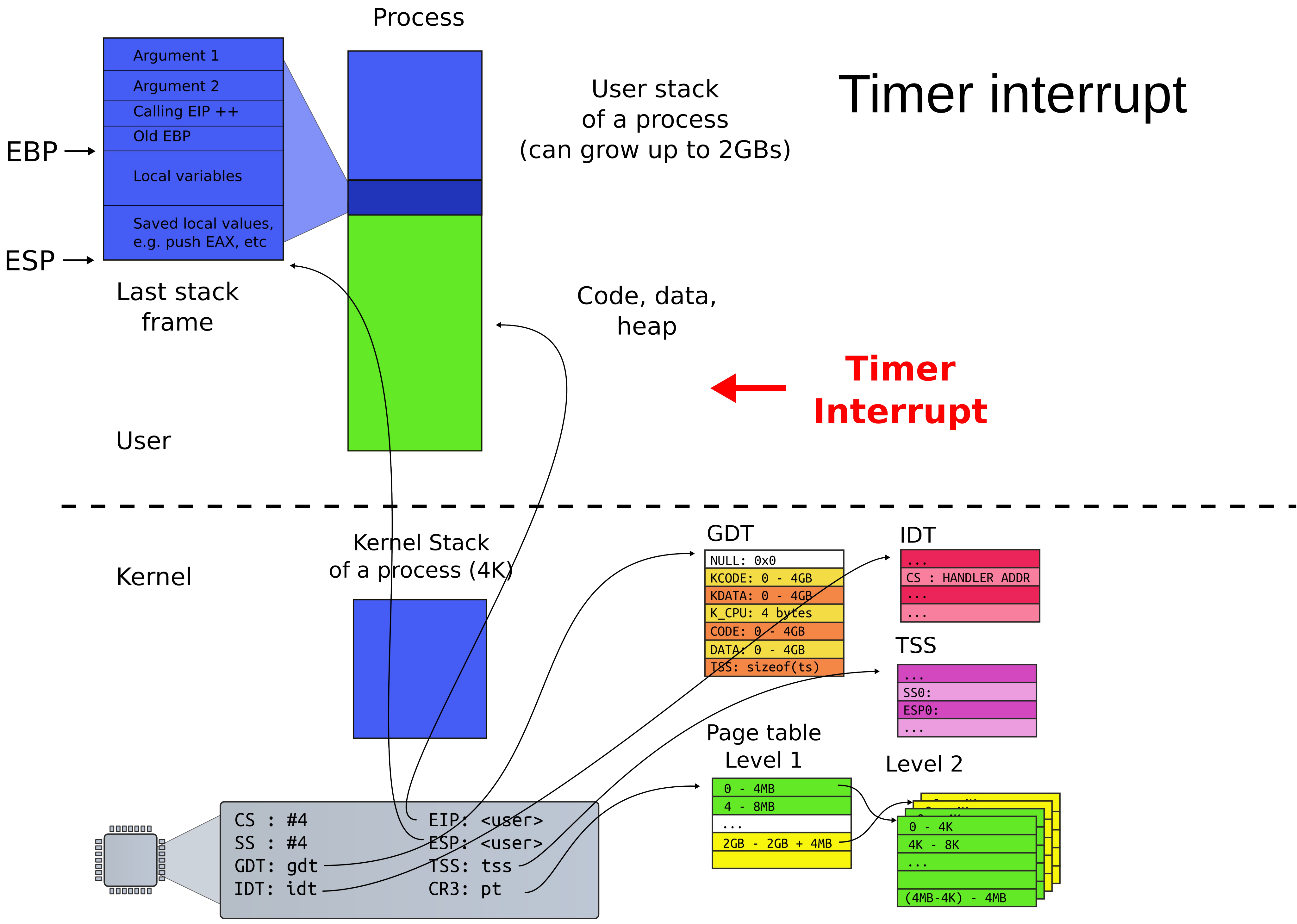
```
CS : #4          EIP: <user>
SS : #4          ESP: <user>
GDT: gdt         TSS: tss
IDT: idt         CR3: pt
```

- User mode
- Two stacks
  - Kernel and user
  - Kernel stack is empty

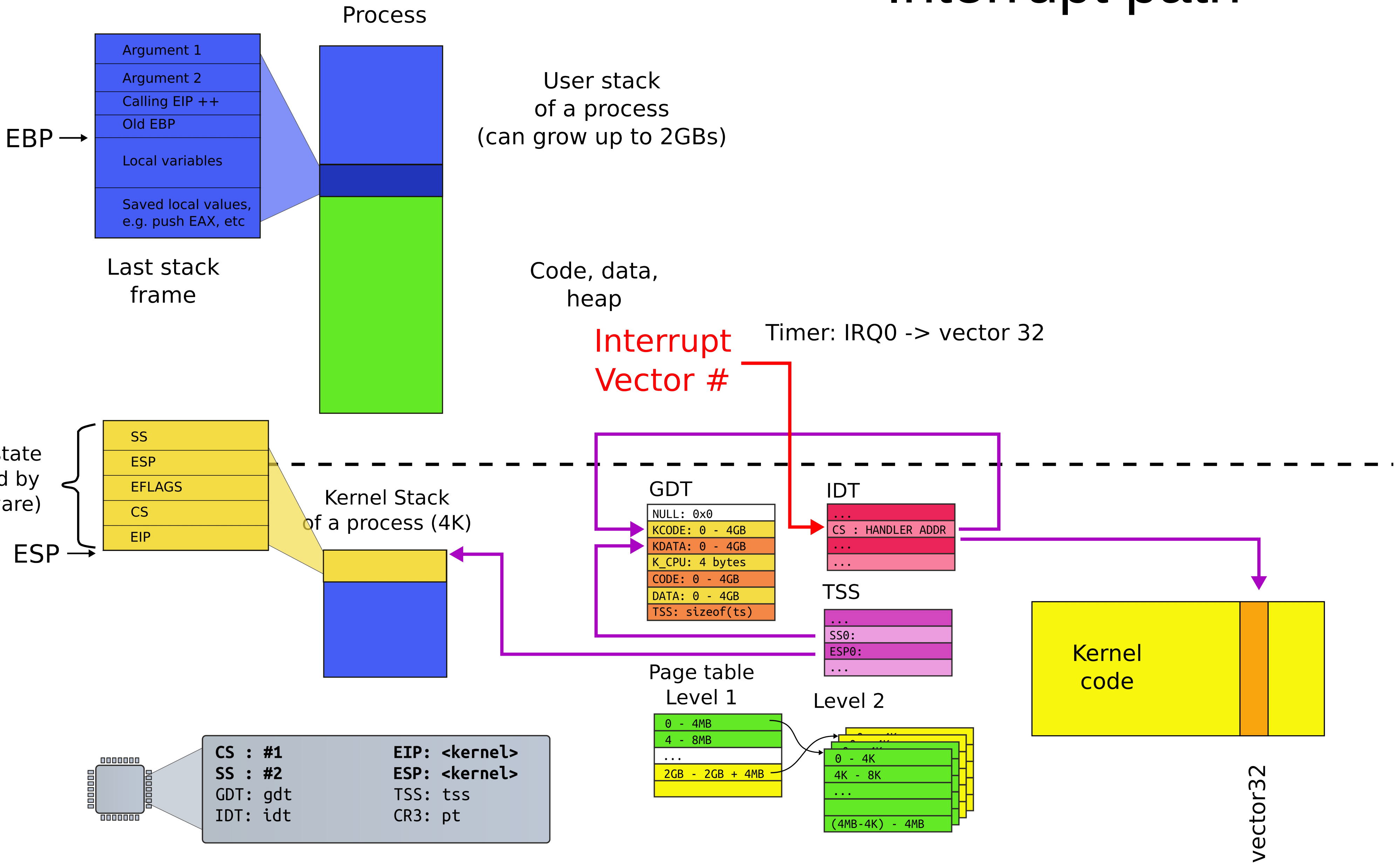


- Page table
- GDT

Timer interrupt



Interrupt path

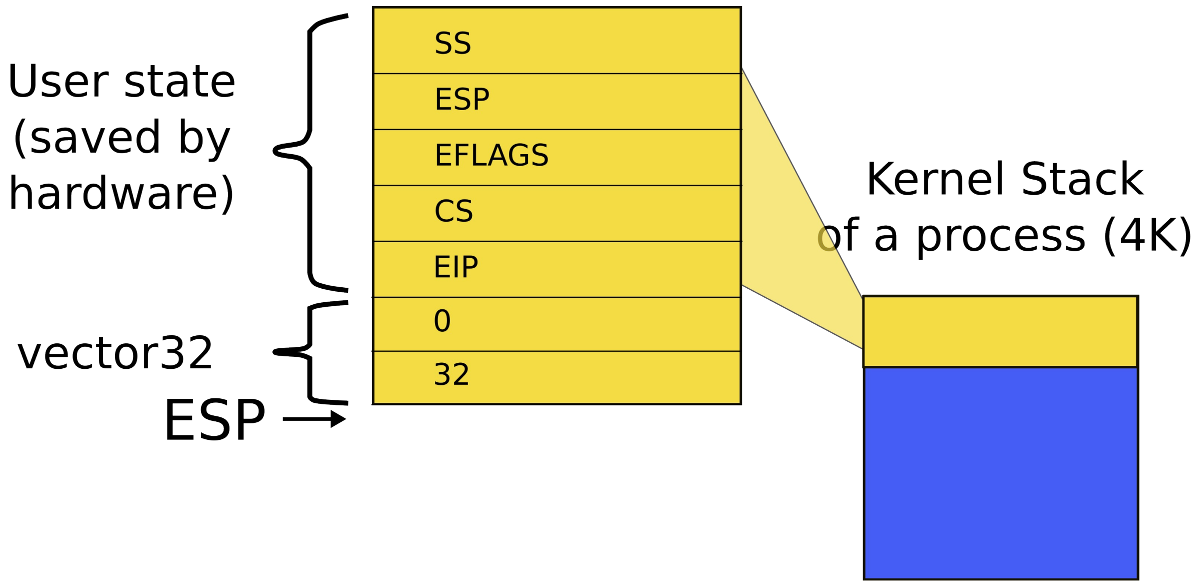


Where does IDT (entry 32) point to?

```
vector32:
  pushl $0     // error code
  pushl $32    // vector #
  jmp alltraps
```

- Automatically generated
- From vectors.pl
  - vector.S

Kernel stack after interrupt

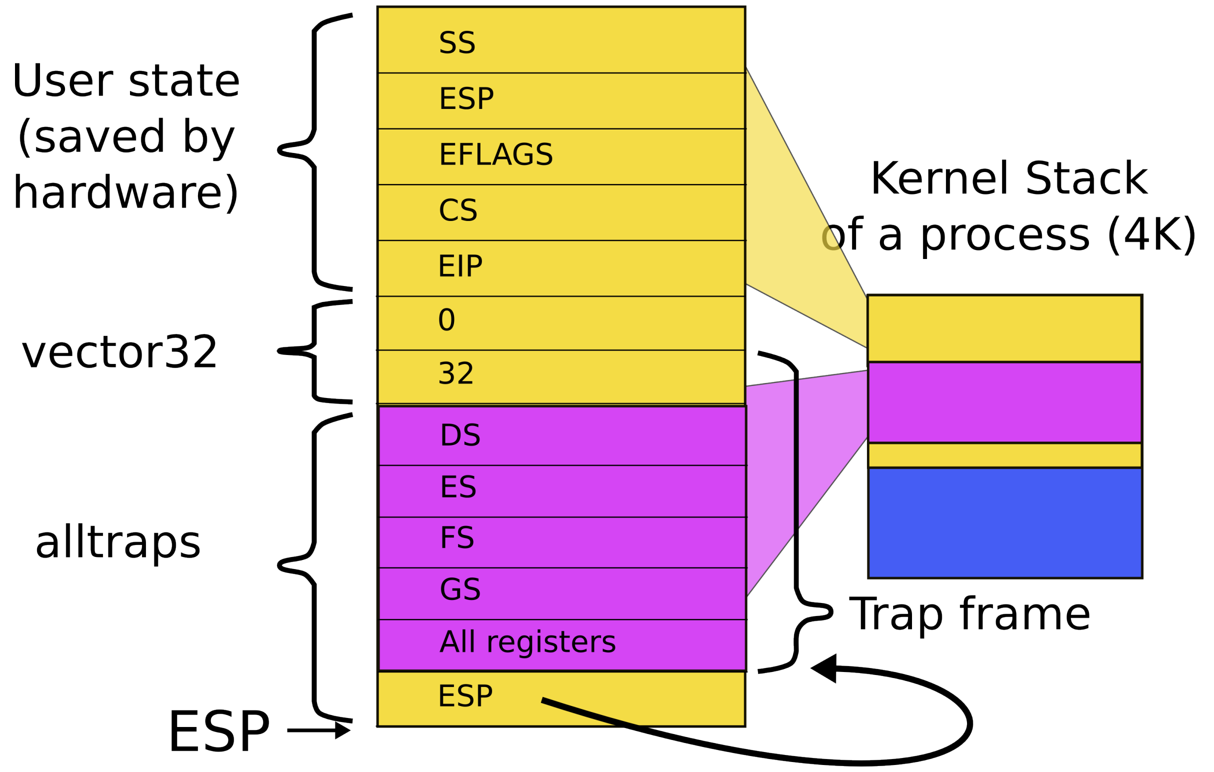


Call stack: vector32()

alltraps()

```
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
```

Kernel stack after interrupt



Call stack: vector32()
alltraps()

alltraps()

```
alltraps:
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal

# Set up data and per-cpu segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
movw $(SEG_KCPU<<3), %ax
movw %ax, %fs
movw %ax, %gs

# Call trap(tf), where tf=%esp
pushl %esp
call trap
```

trap()

```
trap(struct trapframe *tf)
{
...
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  break;
...
  if(proc && proc->state == RUNNING
       && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
```

trap()

```
trap(struct trapframe *tf)
{
...
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  break;
...
  if(proc && proc->state == RUNNING
     && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
```

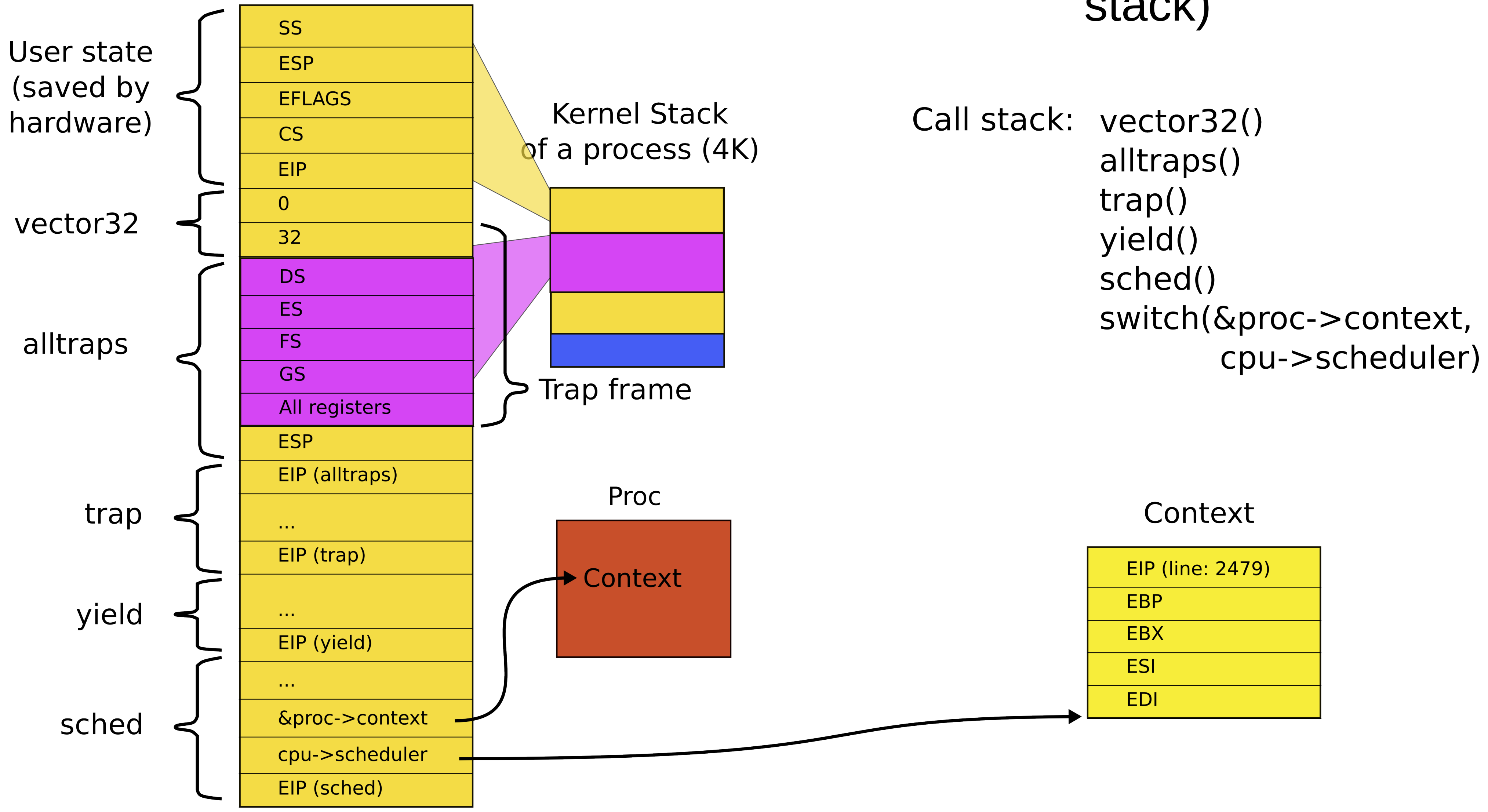
Invoke the scheduler

```
yield(void)
{
    acquire(&ptable.lock);
    proc->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```

Start the context switch

```
sched(void)
{
...
    swtch(&proc->context,
              cpu->scheduler);
...
}
```

Stack inside swtch() and its two arguments (passed on the stack)



Call stack:
vector32()
alltraps()
trap()
yield()
sched()
switch(&proc->context,
cpu->scheduler)

swtch()

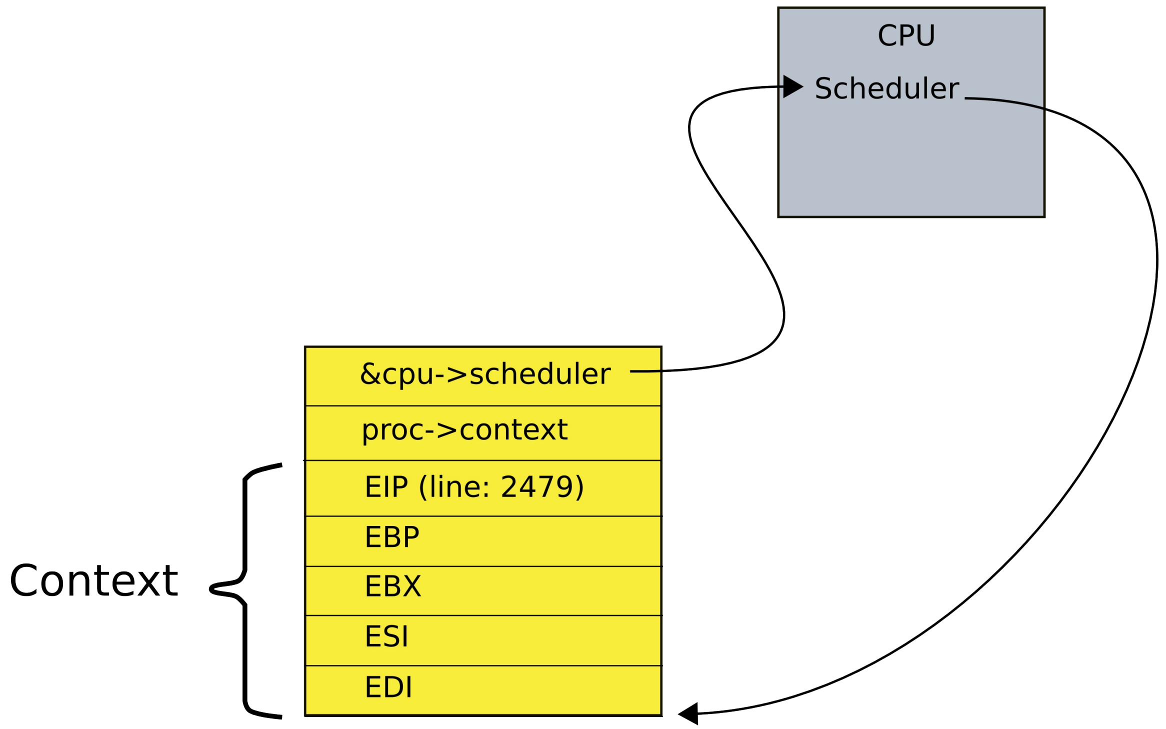
```
swtch:
movl 4(%esp), %eax  # **old
movl 8(%esp), %edx  # *new

# Save old callee-save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacksh
movl %esp, (%eax) # *old = %esp
movl %edx, %esp   # %esp = new

# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

```
void swtch(struct context **old,
                struct context *new);
```

- First argument:
  - A pointer to a pointer to a context
  - That we're going to save
- Second argument:
  - A pointer to a context
  - We're going to restore

Context data structure

```
2093 struct context {
2094   uint edi;
2095   uint esi;
2096   uint ebx;
2097   uint ebp;
2098   uint eip;
2099 };
```

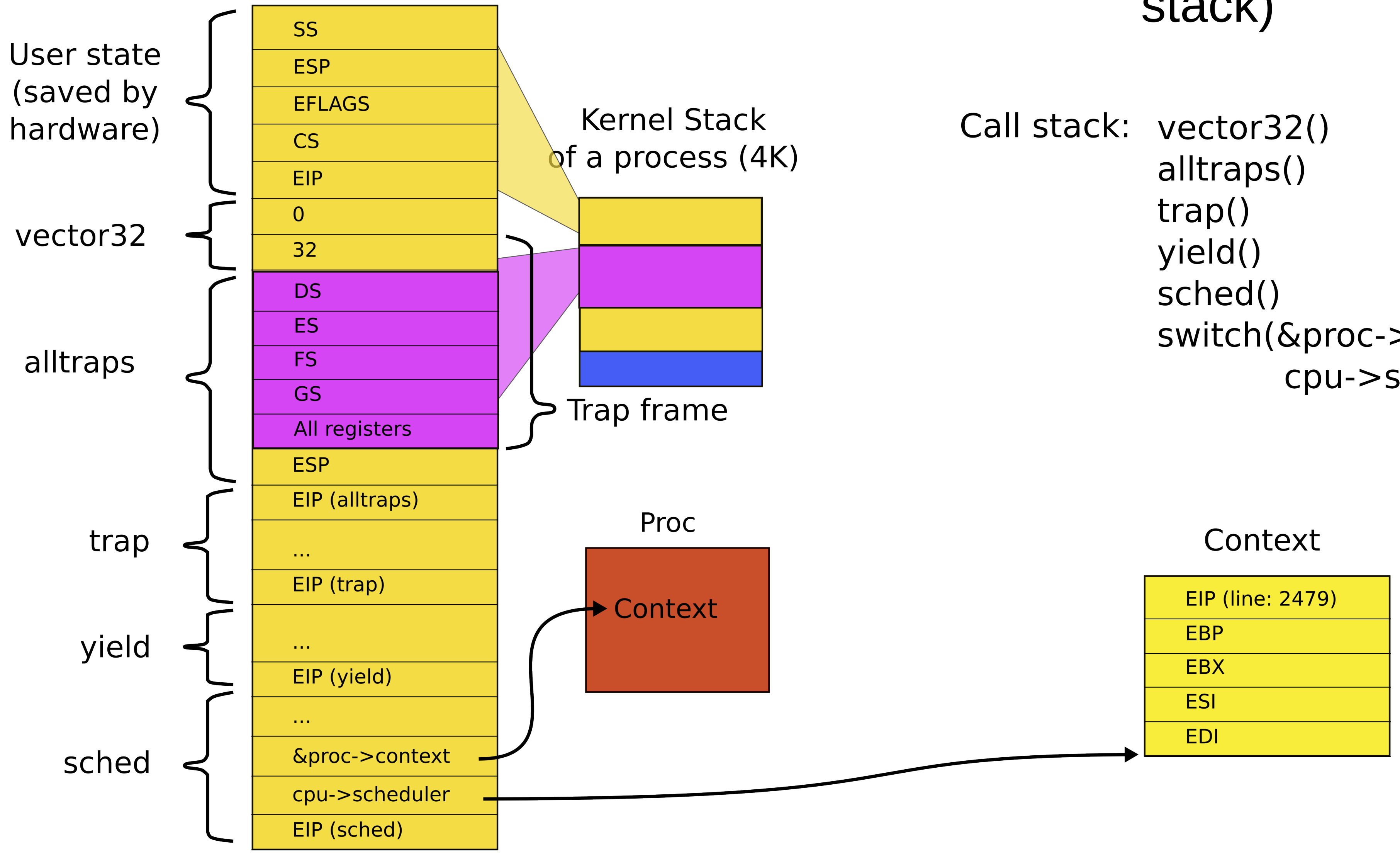


swtch()

```
swtch:
movl 4(%esp), %eax
movl 8(%esp), %edx

# Save old callee-save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacksh
movl %esp, (%eax)
movl %edx, %esp

# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

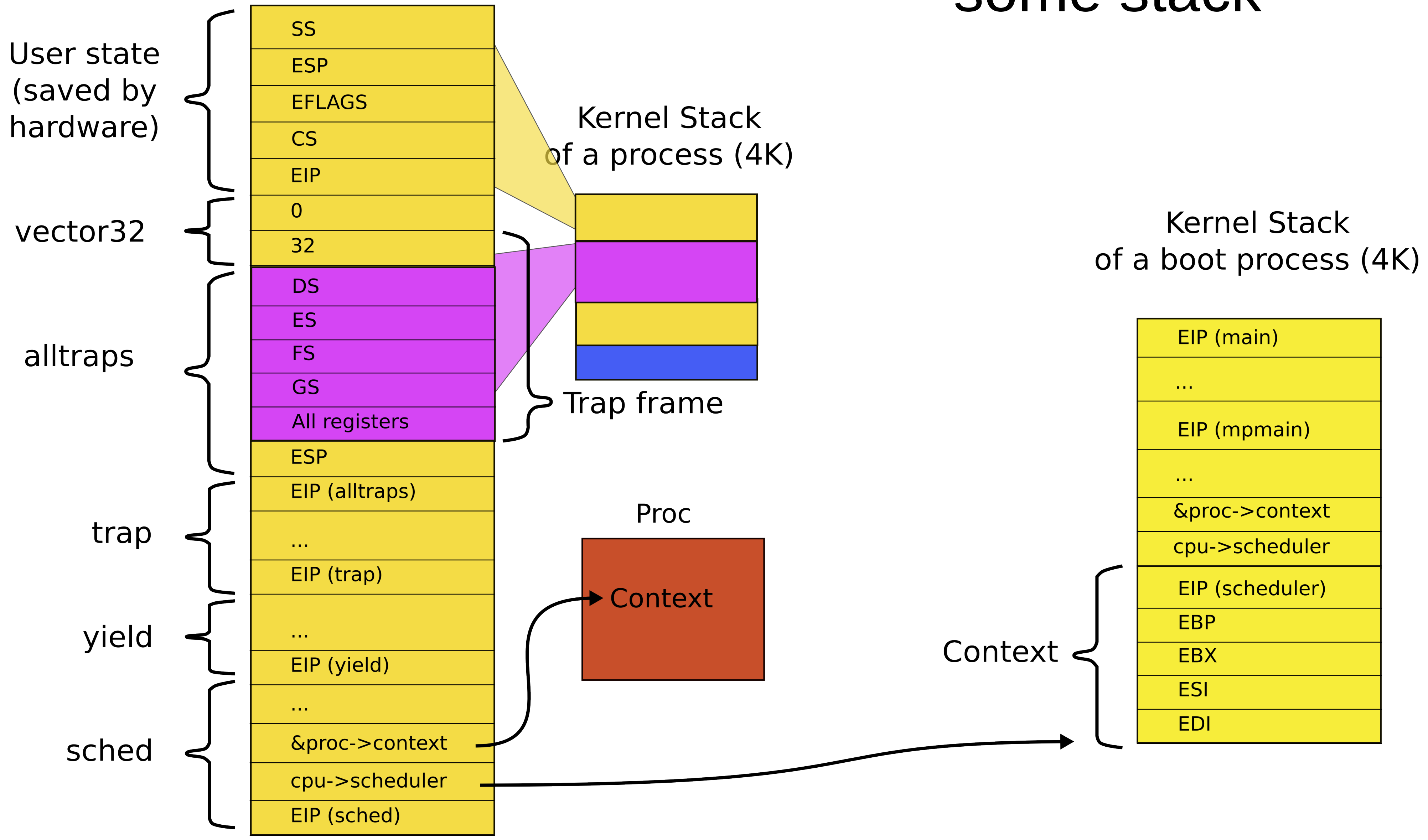
```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Stack inside swtch() and its two arguments (passed on the stack)



Call stack:
vector32()
alltraps()
trap()
yield()
sched()
switch(&proc->context,
cpu->scheduler)

Context is always top of some stack



Context is always top of some stack, why?

- How does initialization of each CPU end?

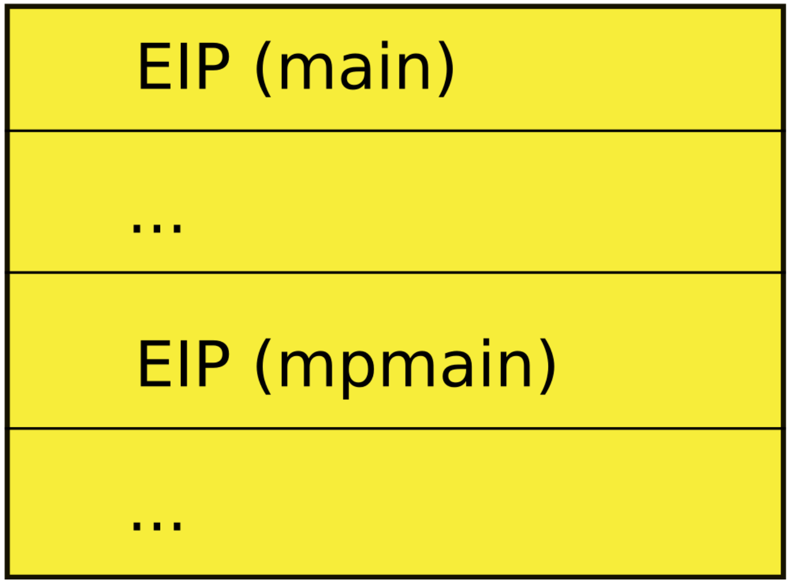
```
static void
mpenter(void)
{
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
```

```
// 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
}
```

We ended boot by starting a scheduler

Remember the stack of the boot process?

Kernel Stack
of a boot process (4K)



How does scheduler start?

```
scheduler(void)
{
  for(;;){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      proc = 0;
    }
  }
}
```

- Chooses next process to run
- Switches to it
  - From the current context

- So when the scheduler context switched the first time

```
2478 swtch(&cpu->scheduler,
                proc->context);
```

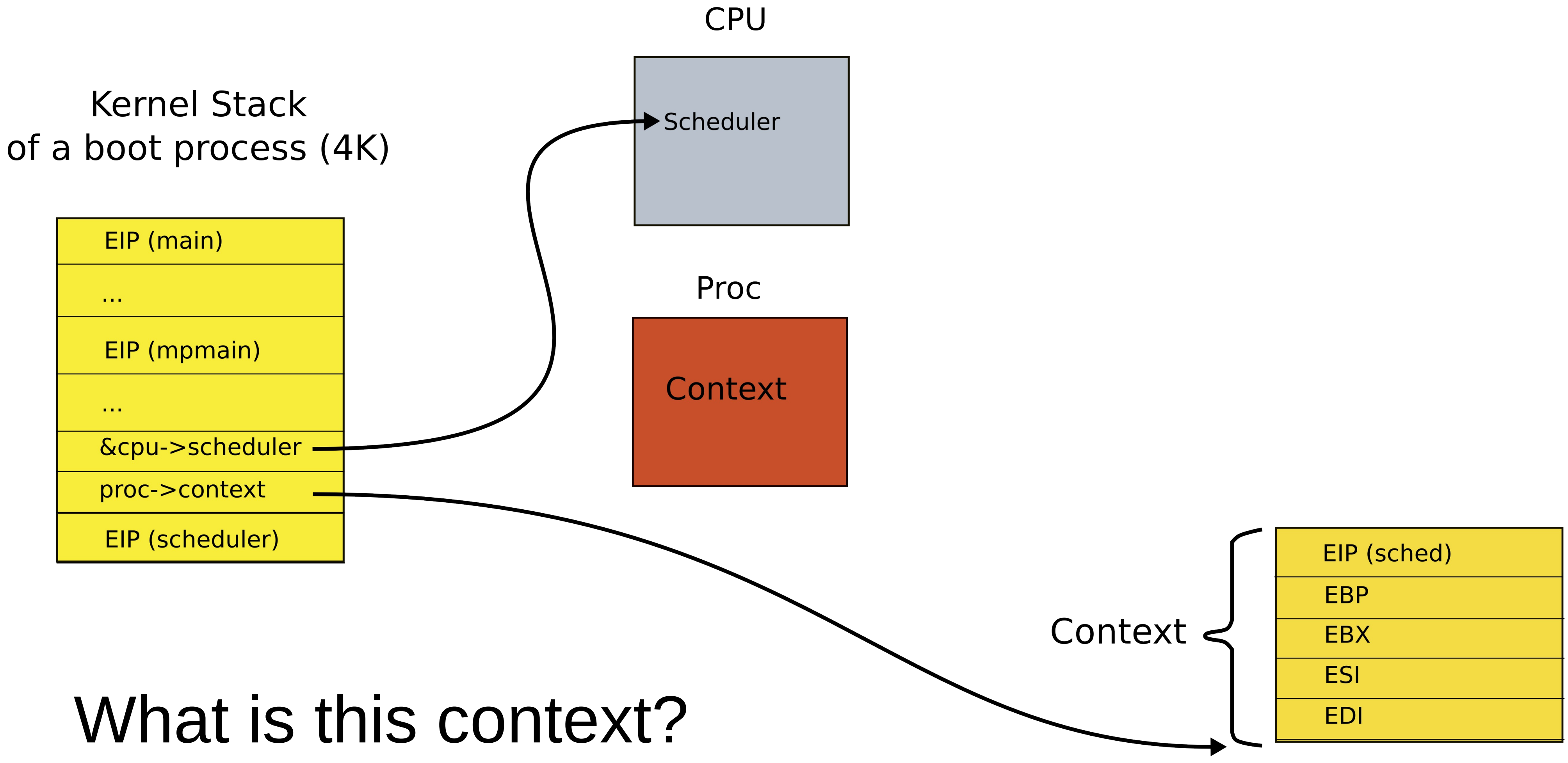
- It saved its own context

```
    &cpu->scheduler
```

- And restored a context of the first process

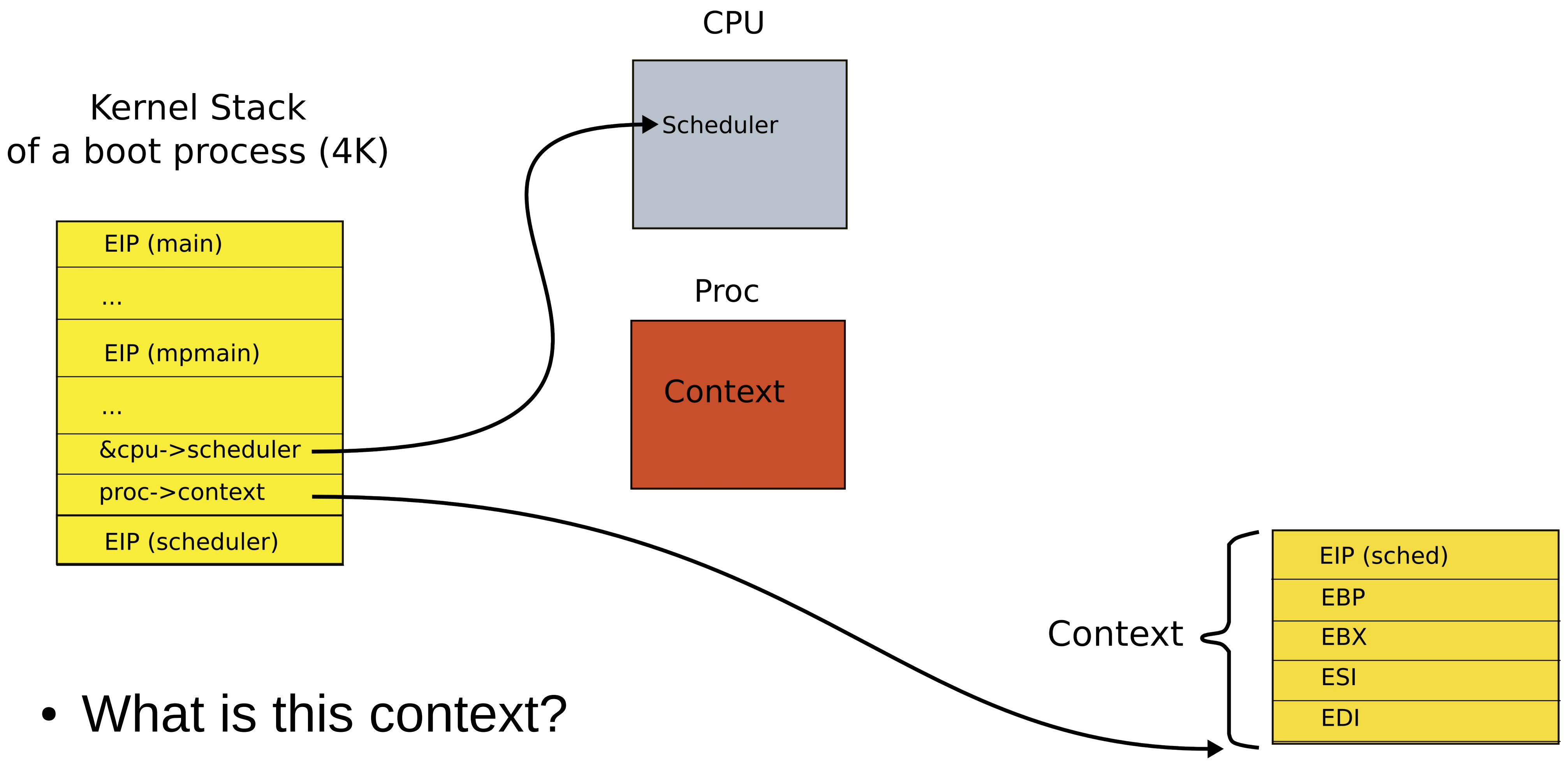
```
    proc->context
```

This is how stack looked like when scheduler() invoked swtch() for the first time



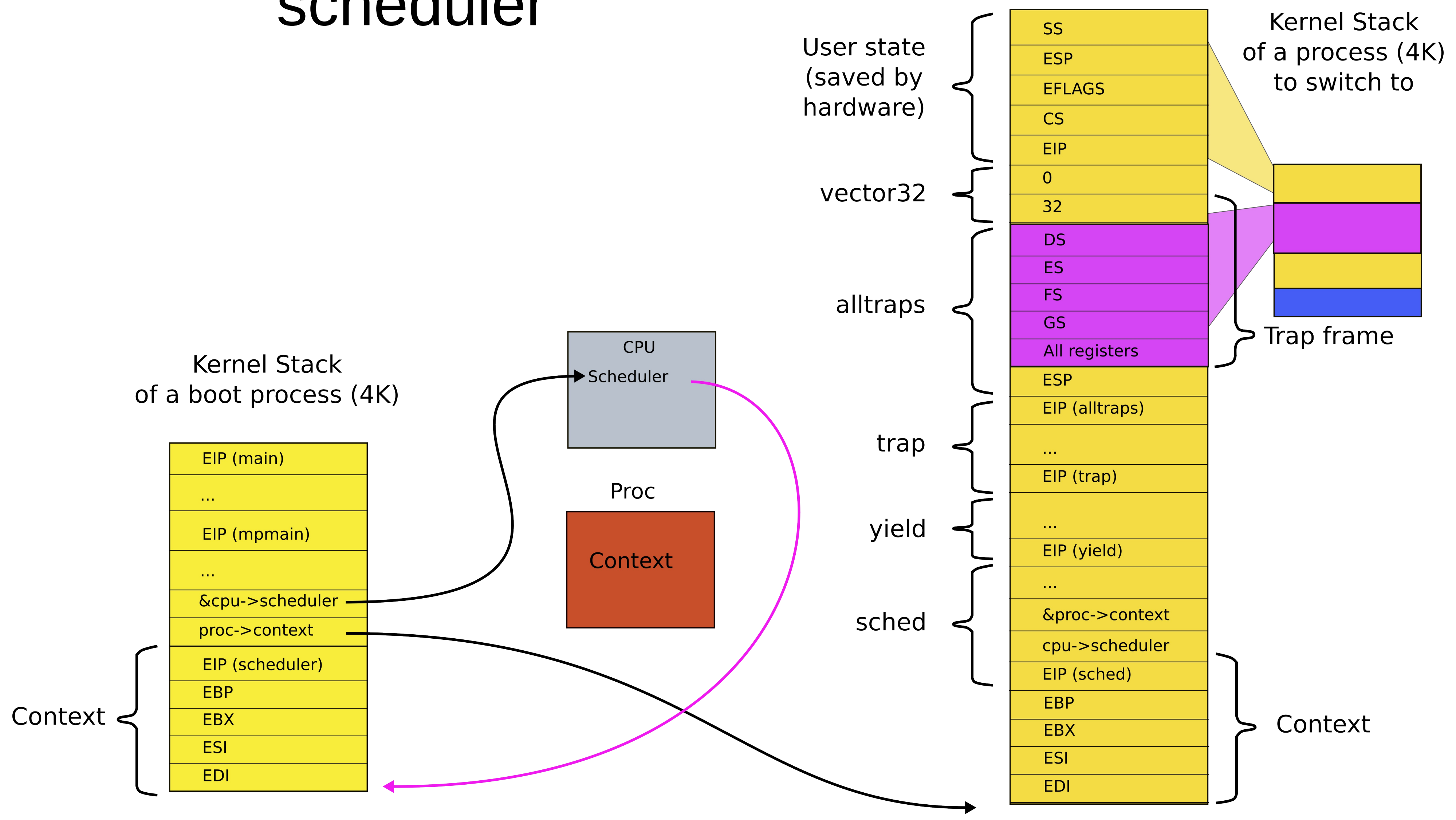
What is this context?

This is how stack looked like when scheduler() invoked swtch() for the first time



- What is this context?
- It's the context of the first process scheduler decides to run

Save context of the scheduler



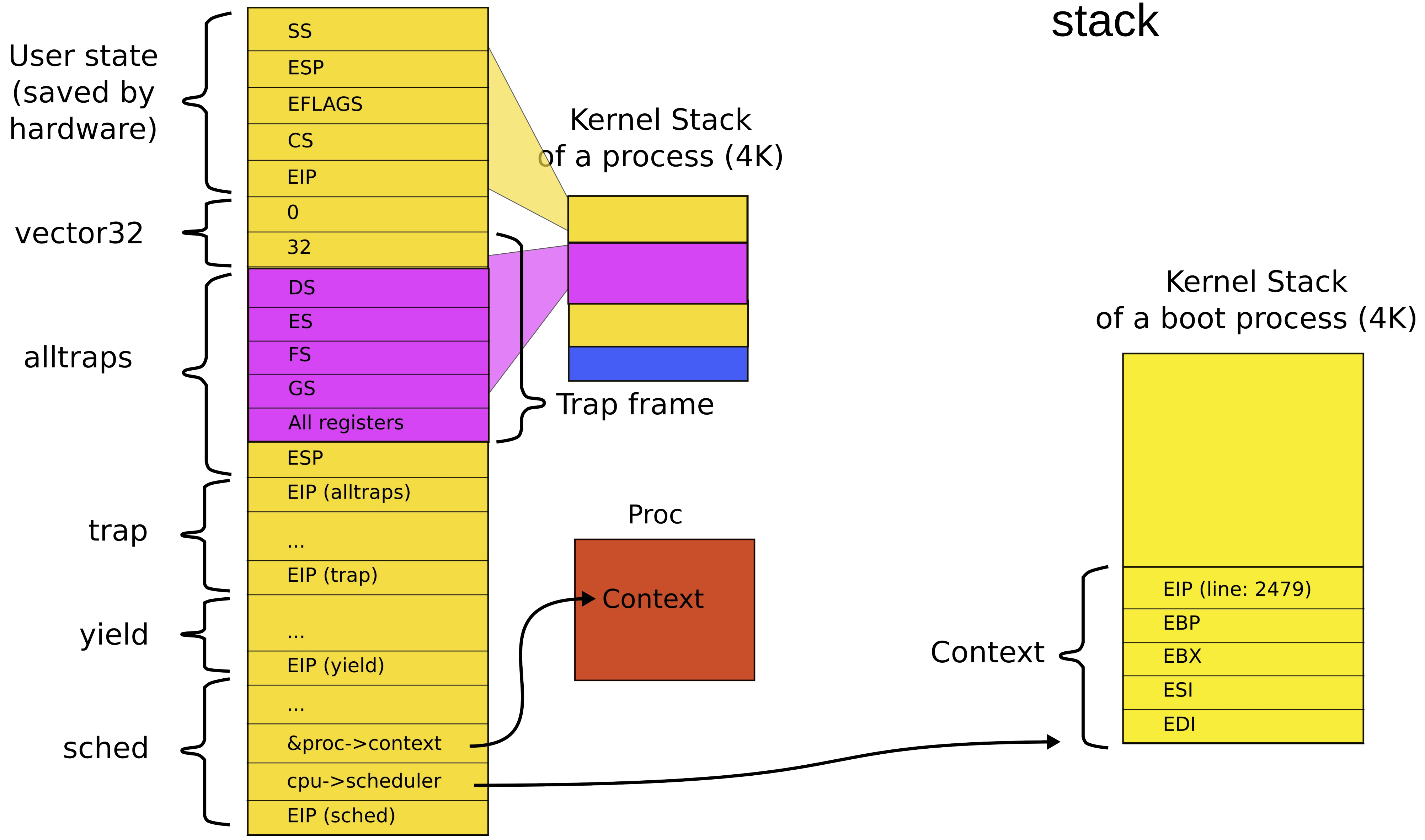
swtch()

```
swtch:
movl 4(%esp), %eax     // struct context **old
movl 8(%esp), %edx     // struct context *new

# Save old callee-save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacksh
movl %esp, (%eax)  // load current context (top of current stack) into
                   //  the memory location pointed by *old
movl %edx, %esp    // set stack to be equal to *new (the top of the new context)

# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

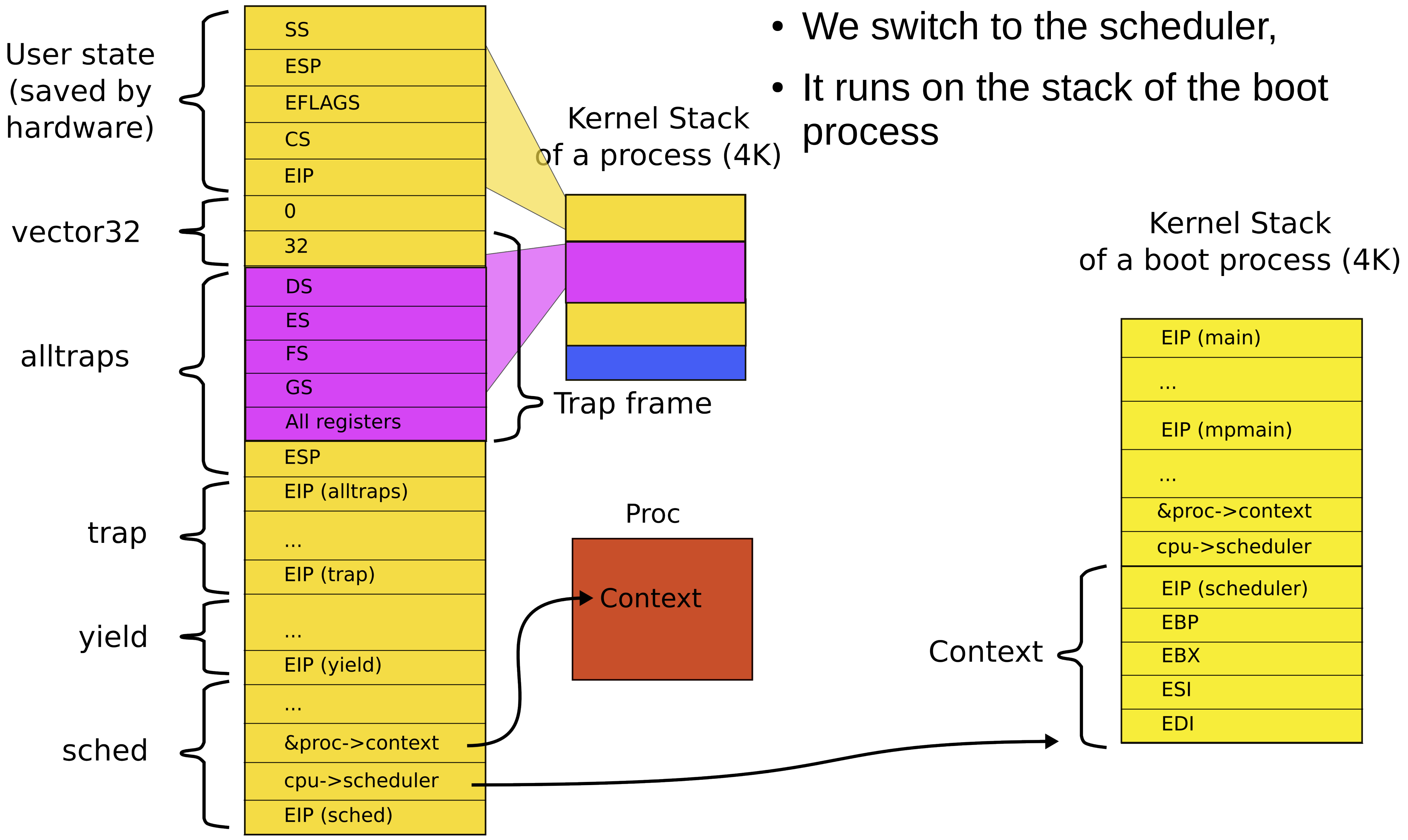
This is why the context is the top of some stack

- Initially it was the stack of mpenter()
  - On which scheduler started
- Then first process...
  - Then scheduler again
  - And the next process...

Back to main context switch: so context is always top of some stack



- Currently the ***new** context is the stack of the scheduler
- We switch to the scheduler,
- It runs on the stack of the boot process



swtch()

```
swtch:
movl 4(%esp), %eax      // struct context **old
movl 8(%esp), %edx      // struct context *new

# Save old callee-save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacks
movl %esp, (%eax)  // load current context (top of current stack) into
                   //  the memory location pointed by *old
movl %edx, %esp    // set stack to be equal to *new (the top of the new context)

# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

And now: exit from swtch()

```
swtch:
movl 4(%esp), %eax
movl 8(%esp), %edx

# Save old callee-save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacks
movl %esp, (%eax)
movl %edx, %esp

# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

Where does this swtch() return?

Where does this swtch() return?

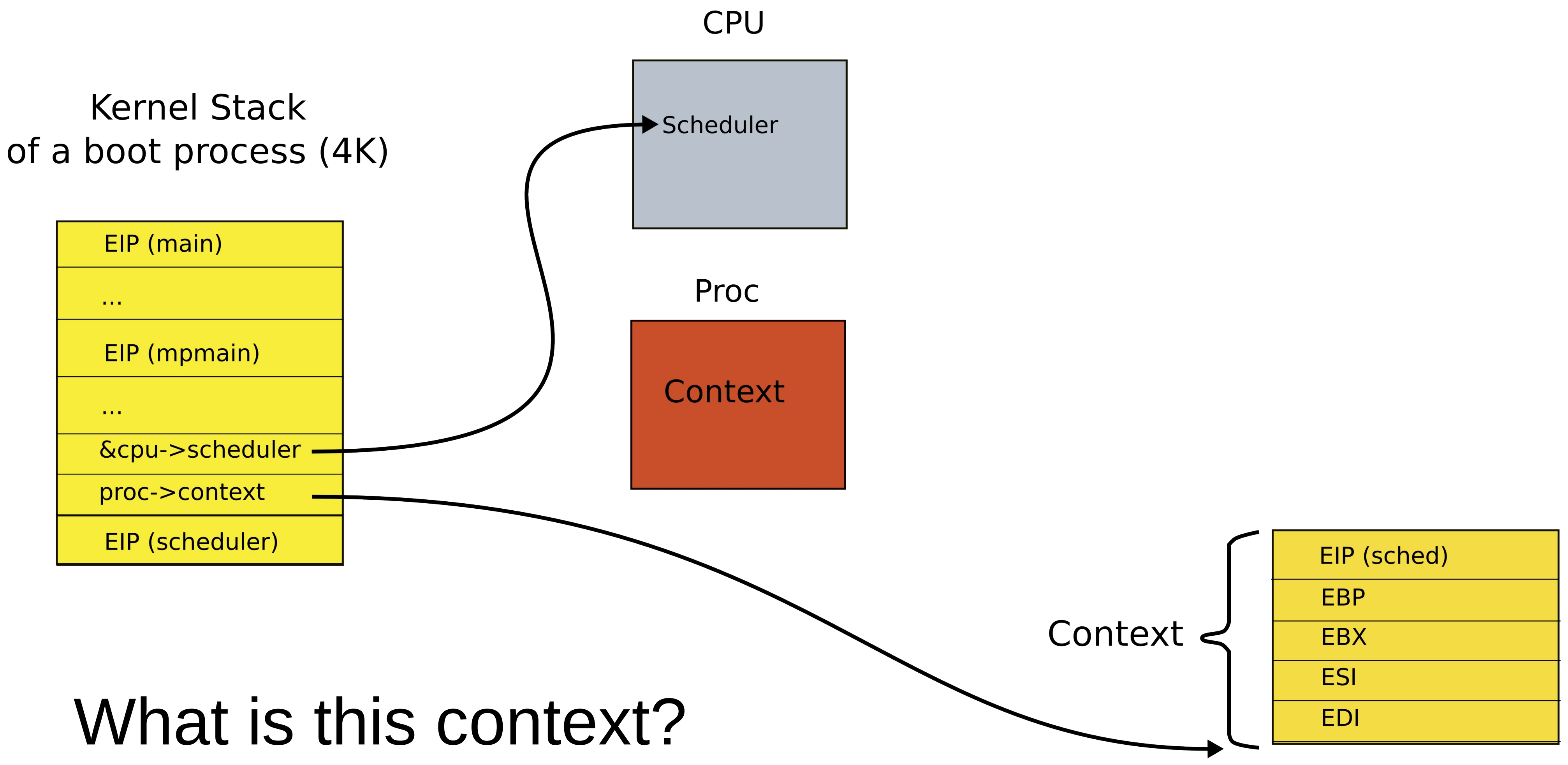
- Scheduler
- After all remember
  - We started with timer interrupt
  - Entered the kernel
  - Entered schedule()
  - Entered switch
- And are currently on our way from the process into the scheduler

What does scheduler do?

```
scheduler(void)
{
  for(;;){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      proc = 0;
    }
  }
}
```

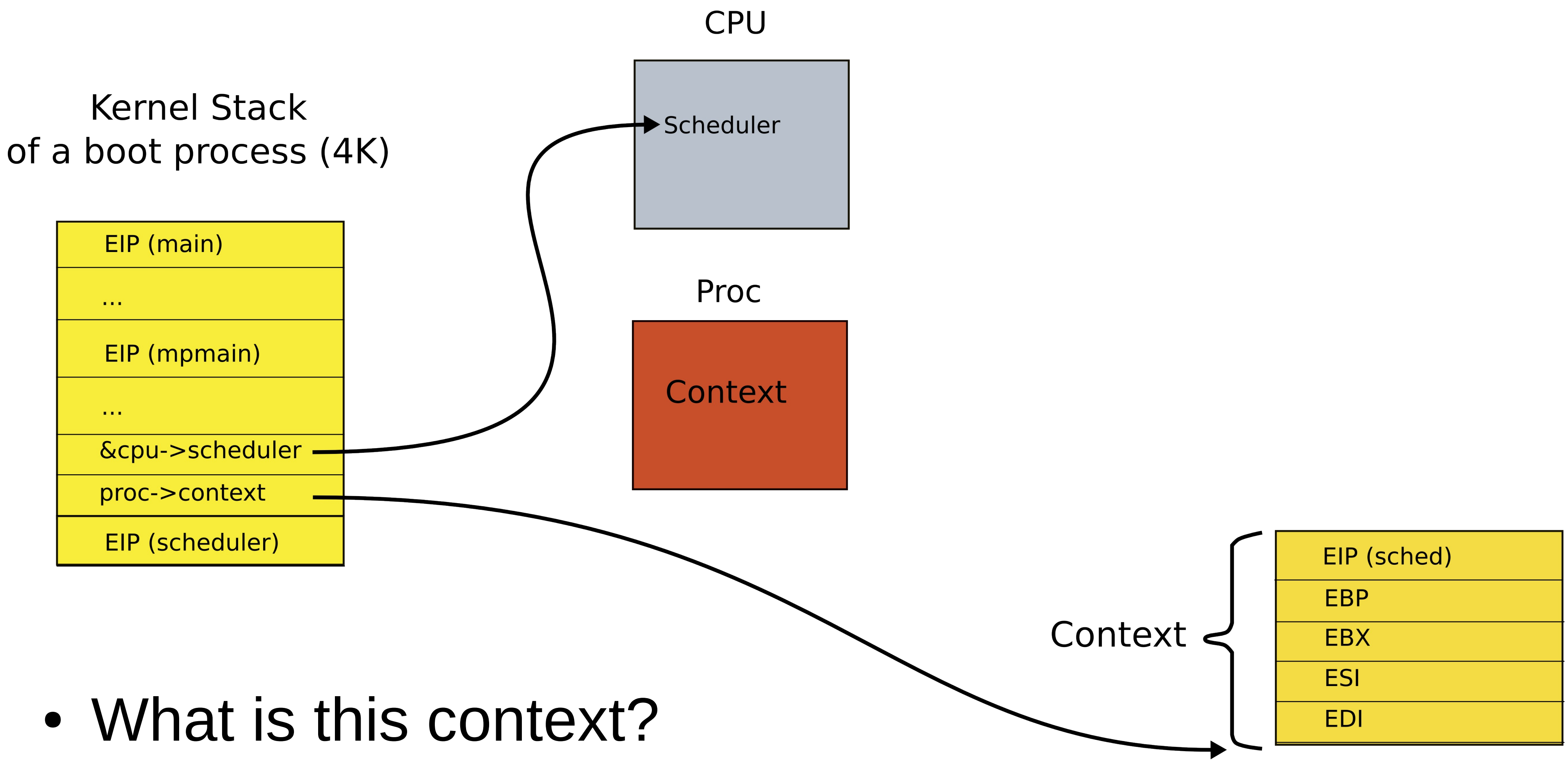
- Chooses next process to run
- Switches to it

What does stack look like when scheduler() invokes swtch()?



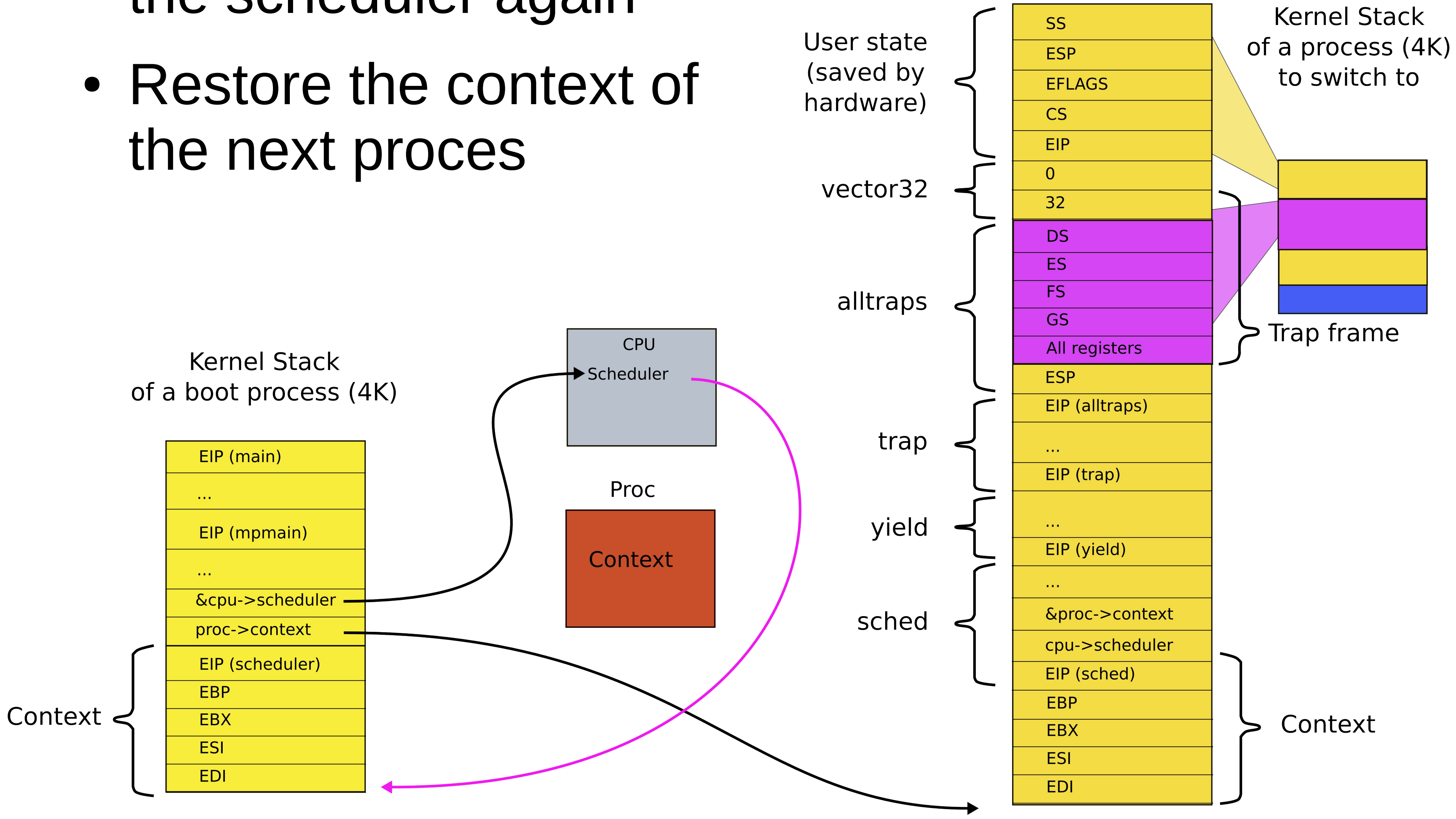
What is this context?

What does stack look like when scheduler() invokes swtch()?



- What is this context?
  - Right the context of the next process to run

- We save the context of the scheduler again
- Restore the context of the next proces



- Remember, from inside the scheduler we invoked swtch() as

```
swtch(&cpu->scheduler,
           proc->context);
```

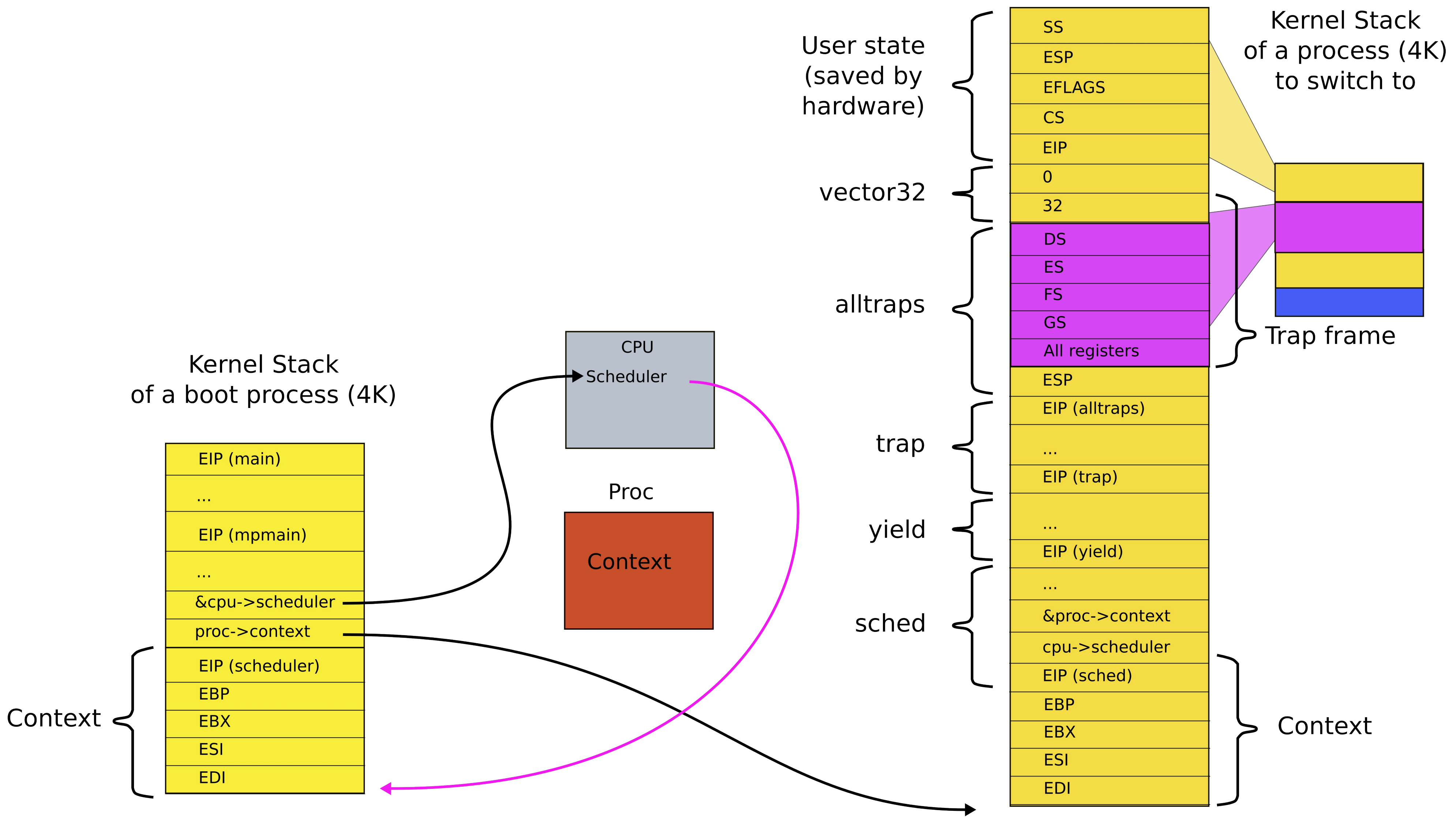
- Hence, we save context of the scheduler into

```
&cpu->scheduler
```

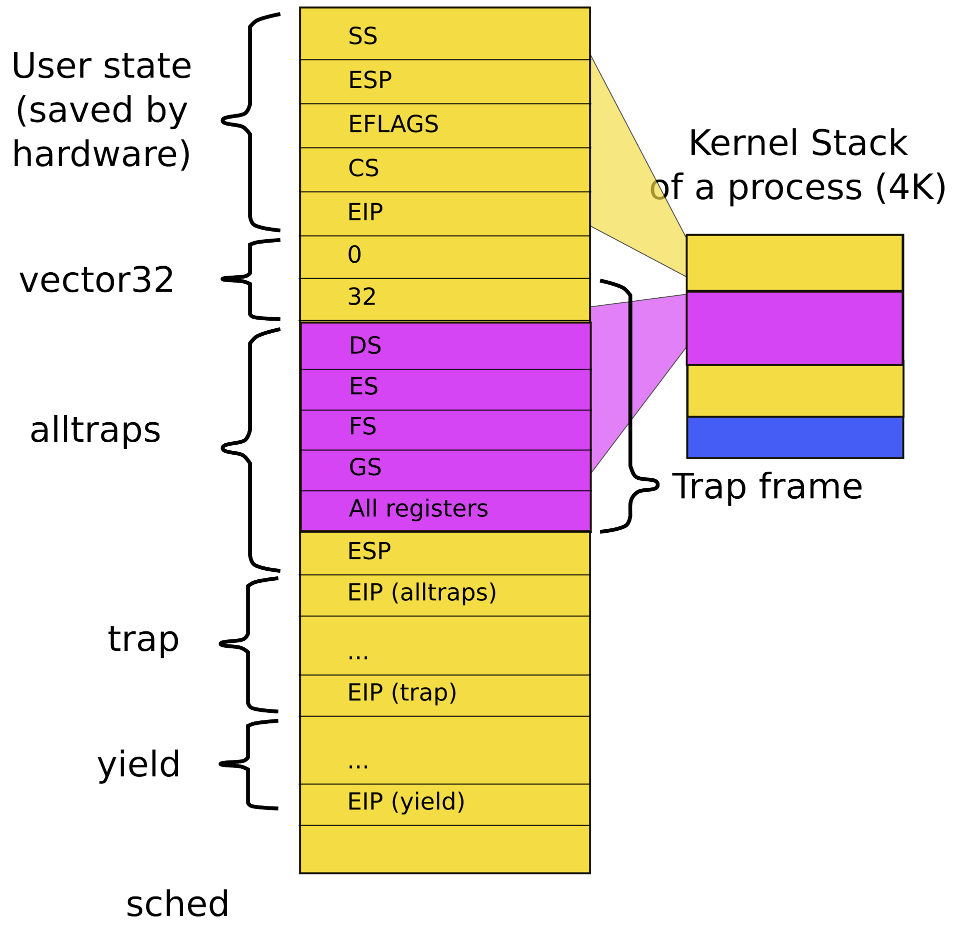
- And restore

```
proc->context
```

Stacks and context inside swtch()



Exiting back to user-level

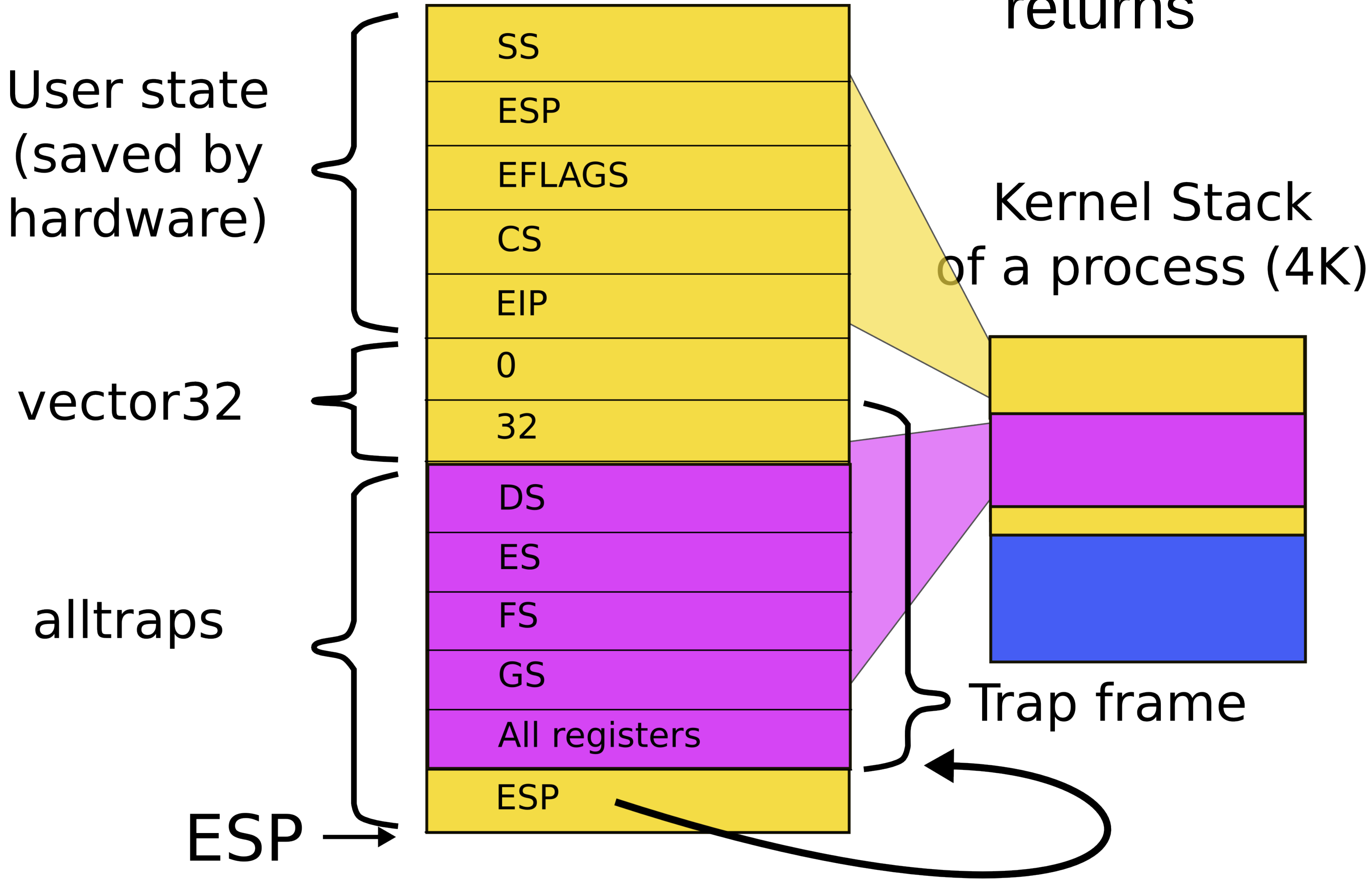


- Stack inside sched()
  - Normal returns until back to alltrap()

alltraps(): Exiting back into user level process

```
alltraps:
...
# Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp

# Return falls through to trapret...
.globl trapret
trapret:
popal
popl %gs
popl %fs
popl %es
popl %ds
addl $0x8, %esp # trapno and errcode
iret
```

Stack after trap() returns

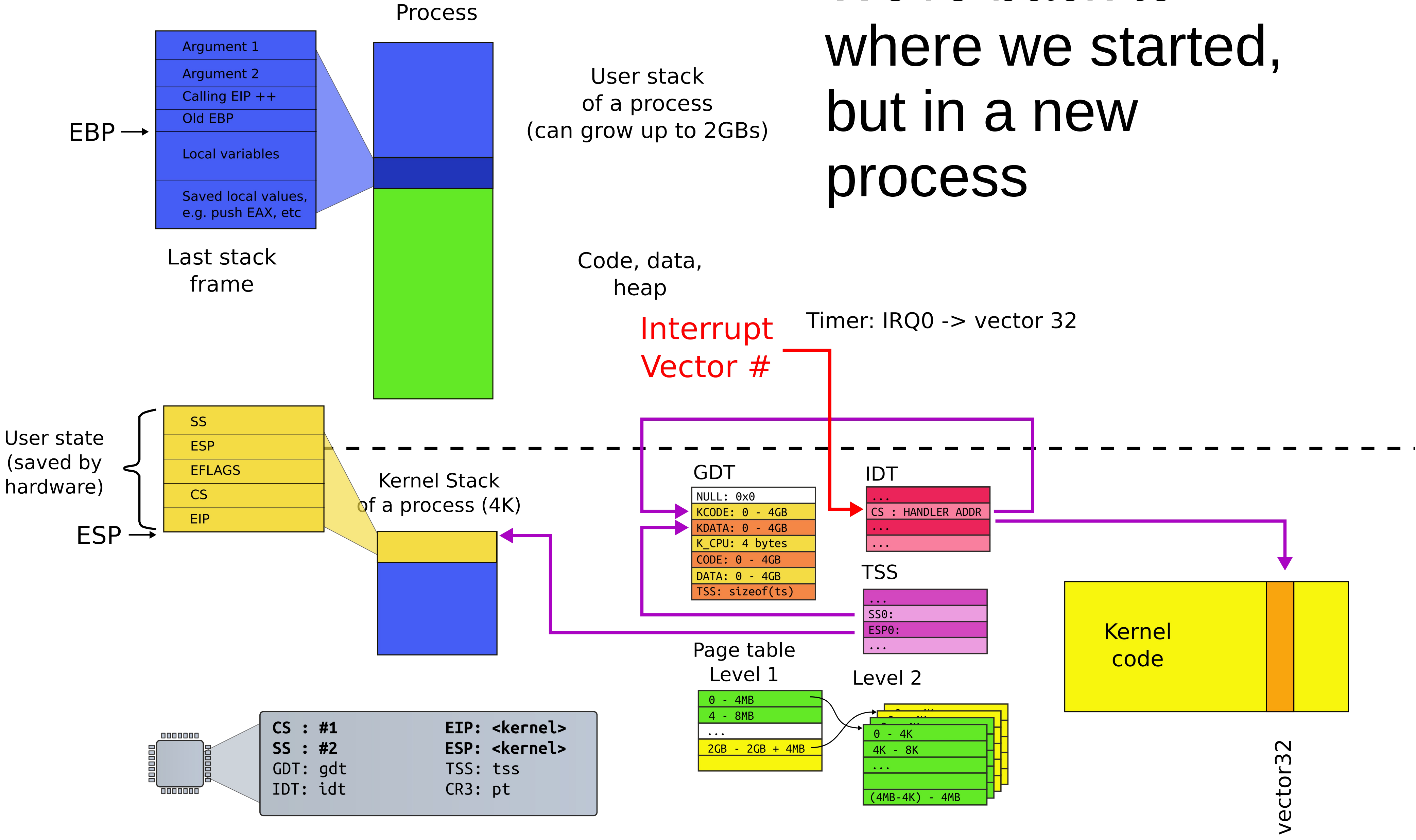


alltraps(): exiting

```
alltraps:
...
# Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp

# Return falls through to trapret...
.globl trapret
trapret:
popal
popl %gs
popl %fs
popl %es
popl %ds
addl $0x8, %esp # trapno and errcode
iret
```

- Restore all registers
- Exit into user
  - iret

We're back to where we started, but in a new process



```
CS : #1          EIP: <kernel>
SS : #2          ESP: <kernel>
GDT: gdt         TSS: tss
IDT: idt         CR3: pt
```

Summary

- We switch between processes now

Thank you