238P: Operating Systems

Lecture 13: Context switch

Anton Burtsev

November, 2019

Creating the first process

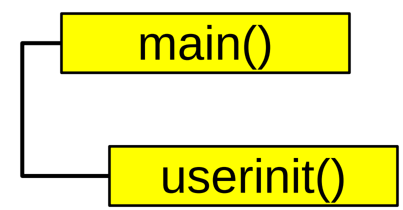
```
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
...
  seginit(); // segment descriptors
...
  tvinit(); // trap vectors
...
  userinit(); // first user process
  mpmain(); // finish this processor's setup
}
```

main()

Userinit() – create first process

- Allocate process structure
  - Information about the process

```
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[],
              _binary_initcode_size[];
...
  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start,
          (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
...
}
```

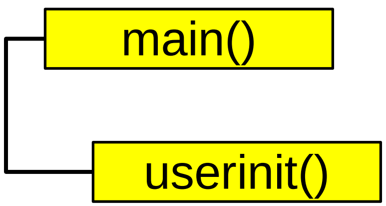


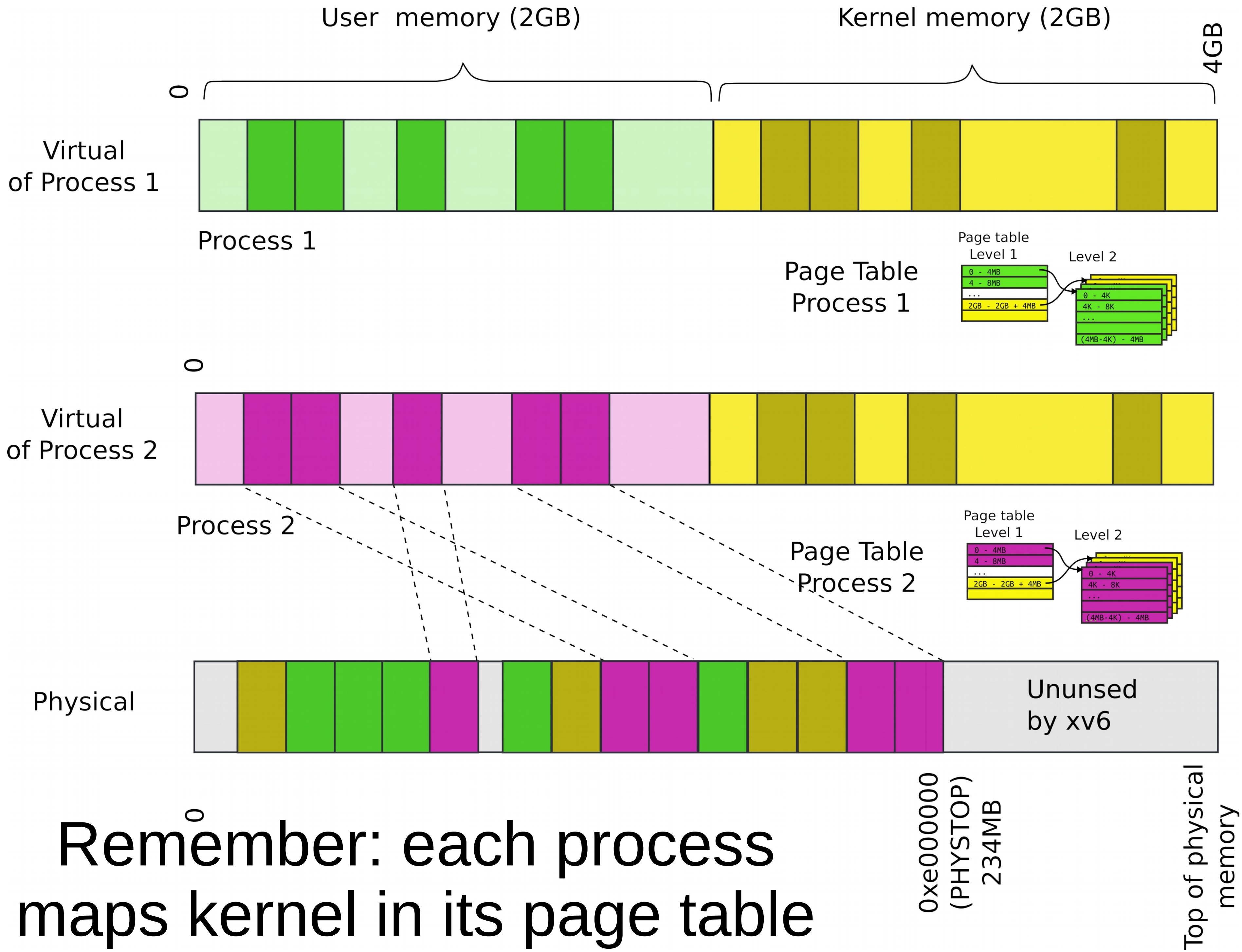
```c
struct proc {
  uint sz; // Size of process memory (bytes)
  pde_t* pgdir; // Page table
  char *kstack; // Bottom of kernel stack for this process
  enum procstate state; // Process state
  volatile int pid; // Process ID
  struct proc *parent; // Parent process
  struct trapframe *tf; // Trap frame for current syscall
  struct context *context; // swtch() here to run
  void *chan; // If non-zero, sleeping on chan
  int killed; // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd; // Current directory
  char name[16]; // Process name (debugging)
};
```

Userinit() – create first process

- Allocate process structure
  - Information about the process
- **Create a page table**
  - **Map only kernel space**

```c
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[],
              _binary_initcode_size[];
...
  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start,
          (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
...
}
```



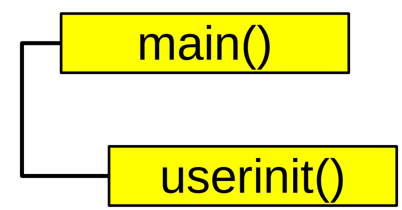


Remember: each process maps kernel in its page table

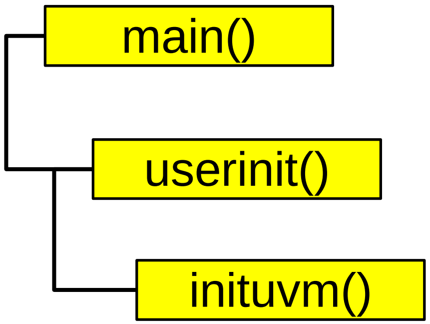
Userinit() – create first process

- Allocate process structure
  - Information about the process
- Create a page table
  - Map only kernel space
- **Allocate a page for the user init code**
  - **Map this page**

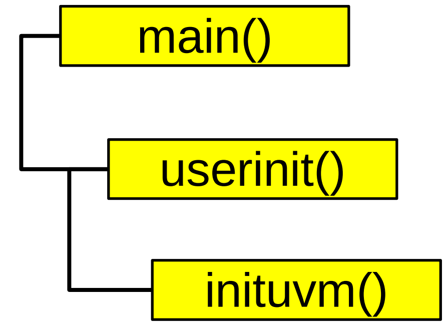
```
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[],
              _binary_initcode_size[];
...
  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start,
         (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
...
}
```



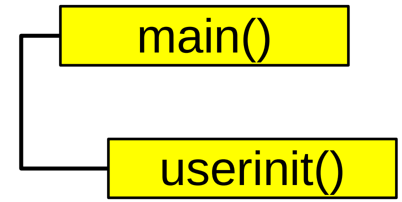
```
initvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem),
                    PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```



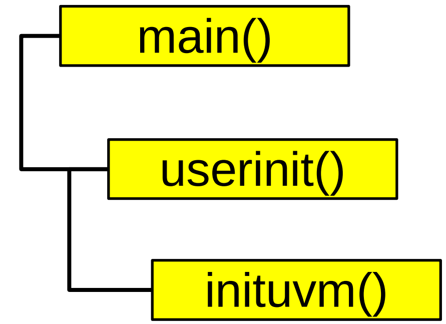
```c
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem),
           PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```



```
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[],
              _binary_initcode_size[];
...
  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start,
          (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
...
}
```



```c
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem),
           PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```



initcode.S: call exec(“/init”, argv);

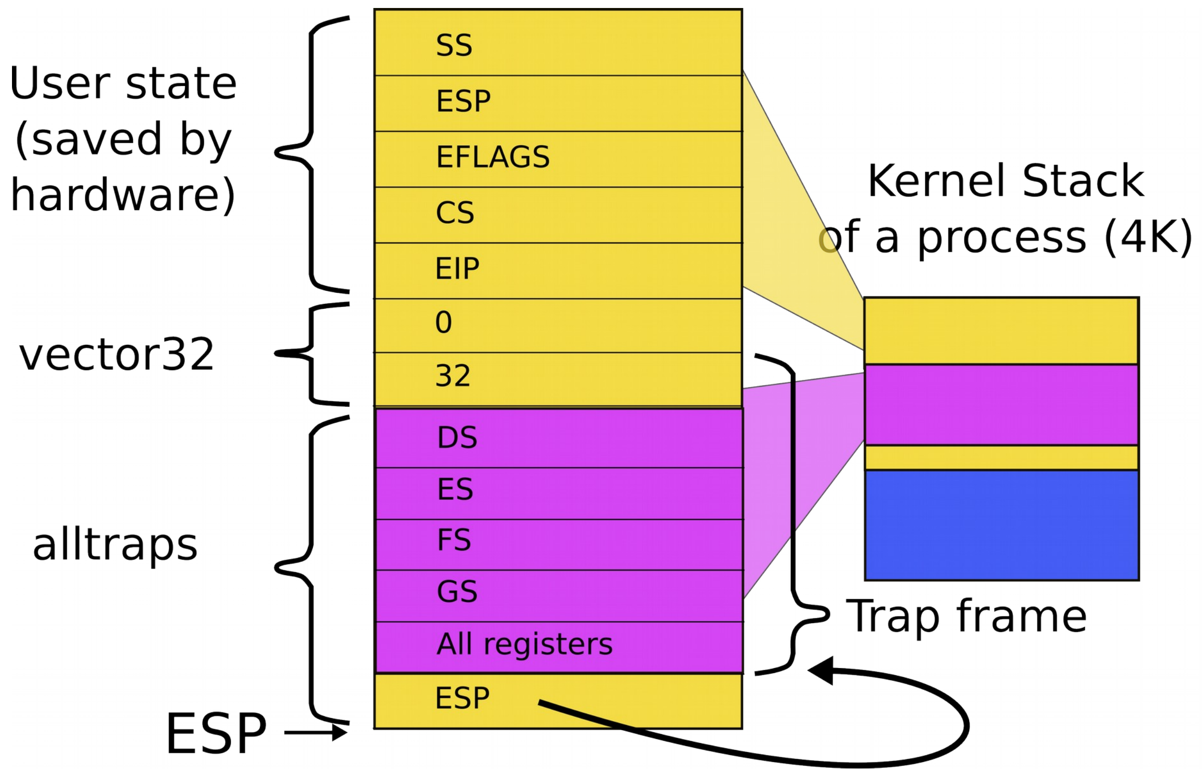
```
start:
  pushl $argv
  pushl $init
  pushl $0 // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

...
# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

userinit() – create first process

- Allocate process structure
  - Information about the process
- Create a page table
  - Map only kernel space
- Allocate a page for the user init code
  - Map this page
- **Configure trap frame for “iret”**

We need to configure the following kernel

- The stack of a process after interrupt/syscall



Call stack: vector32()
alltraps()

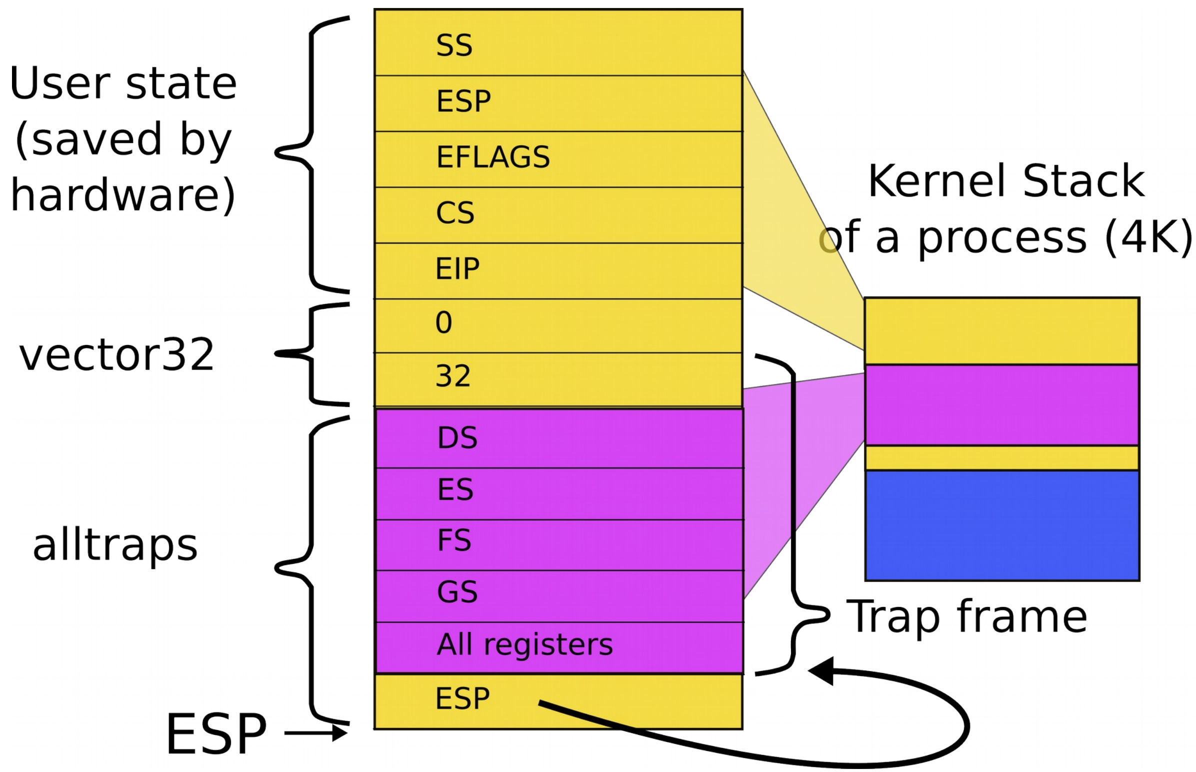
```
struct proc {
  uint sz; // Size of process memory (bytes)
  pde_t* pgdir; // Page table
  char *kstack; // Bottom of kernel stack for this process
  enum procstate state; // Process state
  volatile int pid; // Process ID
  struct proc *parent; // Parent process
  struct trapframe *tf; // Trap frame
  struct context *context; // swtch() here to run
  void *chan; // If non-zero, sleeping on chan
  int killed; // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd; // Current directory
  char name[16]; // Process name (debugging)
};
```

Trap frame is on the kernel stack of the process

```
2456 allocproc(void)
2457 {
...
2470   // Allocate kernel stack.
2471   if((p->kstack = kalloc()) == 0){
2472     p->state = UNUSED;
2473     return 0;
2474   }
2475   sp = p->kstack + KSTACKSIZE;
2476
2477   // Leave room for trap frame.
2478   sp -= sizeof *p->tf;
2479   p->tf = (struct trapframe*)sp;
2480
...
2492 }
```

```
userinit(void)
{
...
  inituvm(p->pgdir, _binary_initcode_start,
             (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0; // beginning of initcode.S
...
}
```

Kernel stack after interrupt/syscall



Call stack: vector32()
alltraps()

```
userinit(void)
{
...
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0; // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;
...
}
```

Wait, we mapped process memory, created trap frame, but it doesn't really run...

```
main(void)
{
...
  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0); // stdout
  dup(0); // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

- First process exec(“init”)
- /init starts /sh
  - fork() and exec(“sh”)

Summary

- We've finally learned how the first process came to life

  Also we know:

- How OS boots and initializes itself
- How each process is constructed (exec())
- How OS switches between processes

When OS context switches between processes?

When OS context switches between processes?

- Timer interrupt preempts the current process
- A process enters the kernel with a system call and has to wait on some resource
  - E.g., write to a pipe, but the pipe is full
- The process voluntarily yields CPU with the yield() system call

Lets look at timer interrupt

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()

```
3351 trap(struct trapframe *tf)
3352 {
...
3363   switch(tf->trapno){
3364   case T_IRQ0 + IRQ_TIMER:
3365     if(cpu->id == 0){
3366     acquire(&tickslock);
3367     ticks++;
3368     wakeup(&ticks);
3369     release(&tickslock);
3370   }
3372   break;
...
3423   if(proc && proc->state == RUNNING
          && tf->trapno == T_IRQ0+IRQ_TIMER)
3424     yield();
```

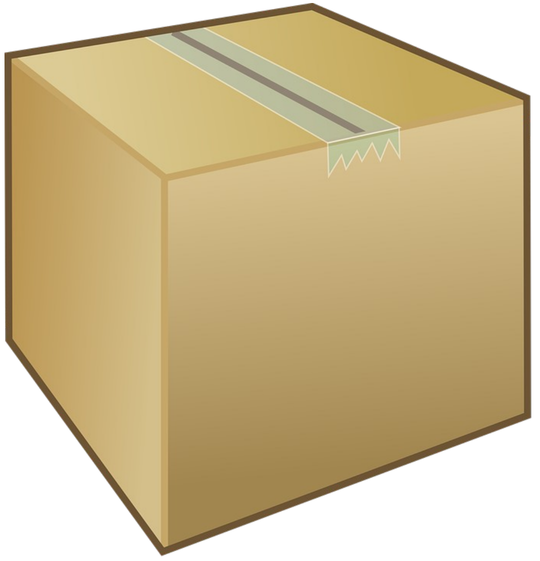
Invoke the scheduler

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

Start the context switch

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

But what do you think needs to happen inside switch()?



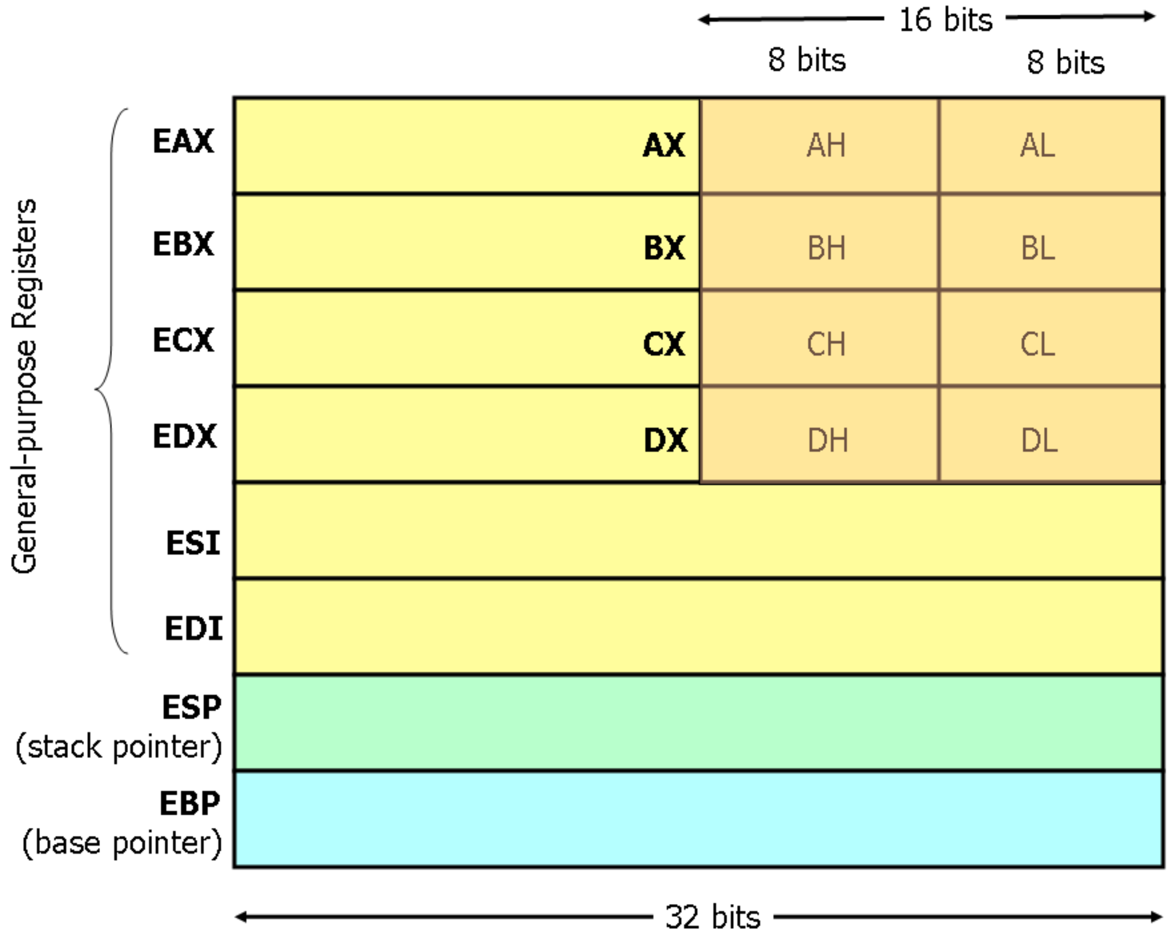
Process 1

Process 2

What should be in the box?
i.e., what is the state of the process?

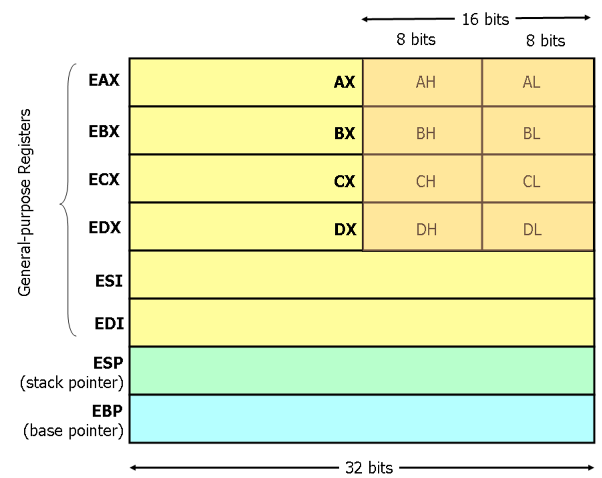
```
struct proc {
  uint sz; // Size of process memory (bytes)
  pde_t* pgdir; // Page table
  char *kstack; // Bottom of kernel stack for this process
  enum procstate state; // Process state
  volatile int pid; // Process ID
  struct proc *parent; // Parent process
  struct trapframe *tf; // Trap frame
  struct context *context; // swtch() here to run
  void *chan; // If non-zero, sleeping on chan
  int killed; // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd; // Current directory
  char name[16]; // Process name (debugging)
};
```

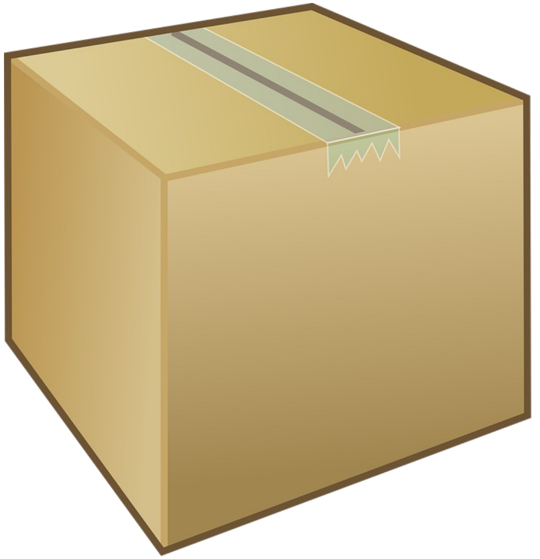
What about general registers?



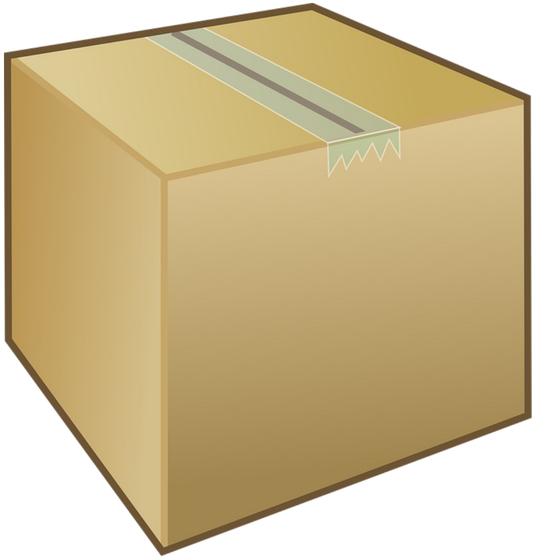
Save them on the stack

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



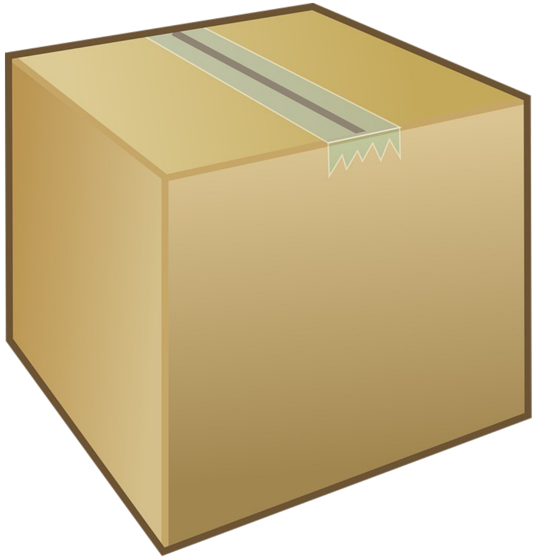


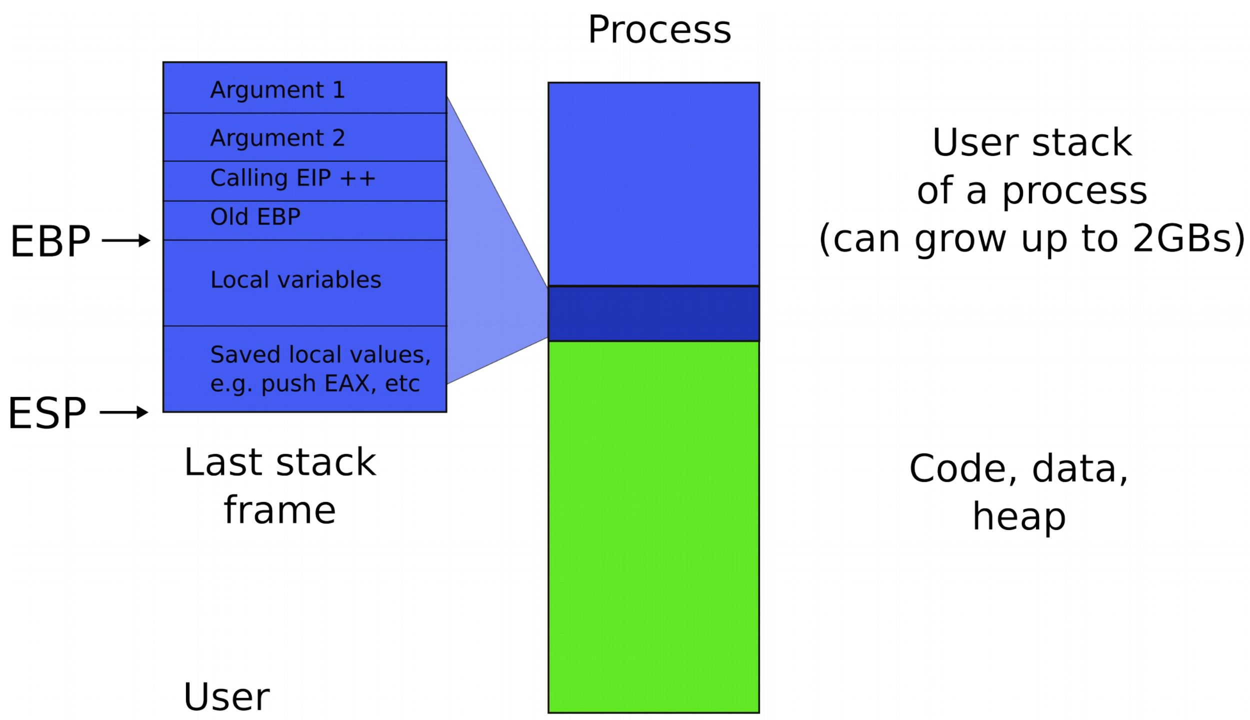
Process 1

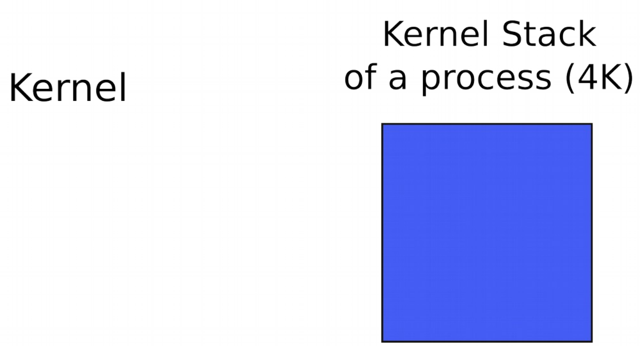


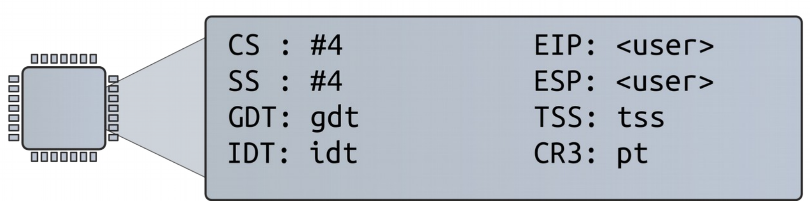
Process 2

Back to the timer interrupt path
(keep track of what happens to the stack, lets see how everything gets packed in a “box”!)



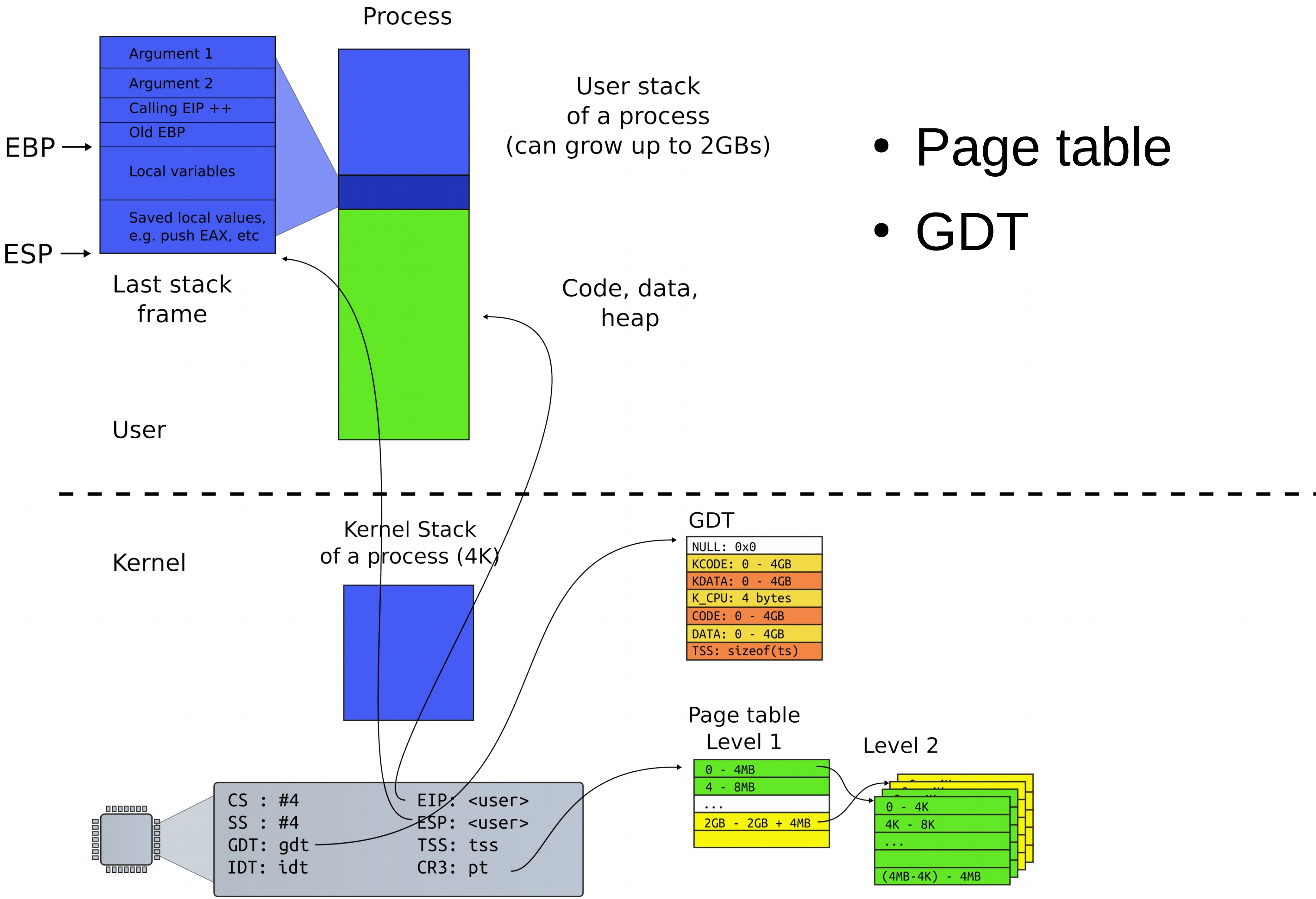


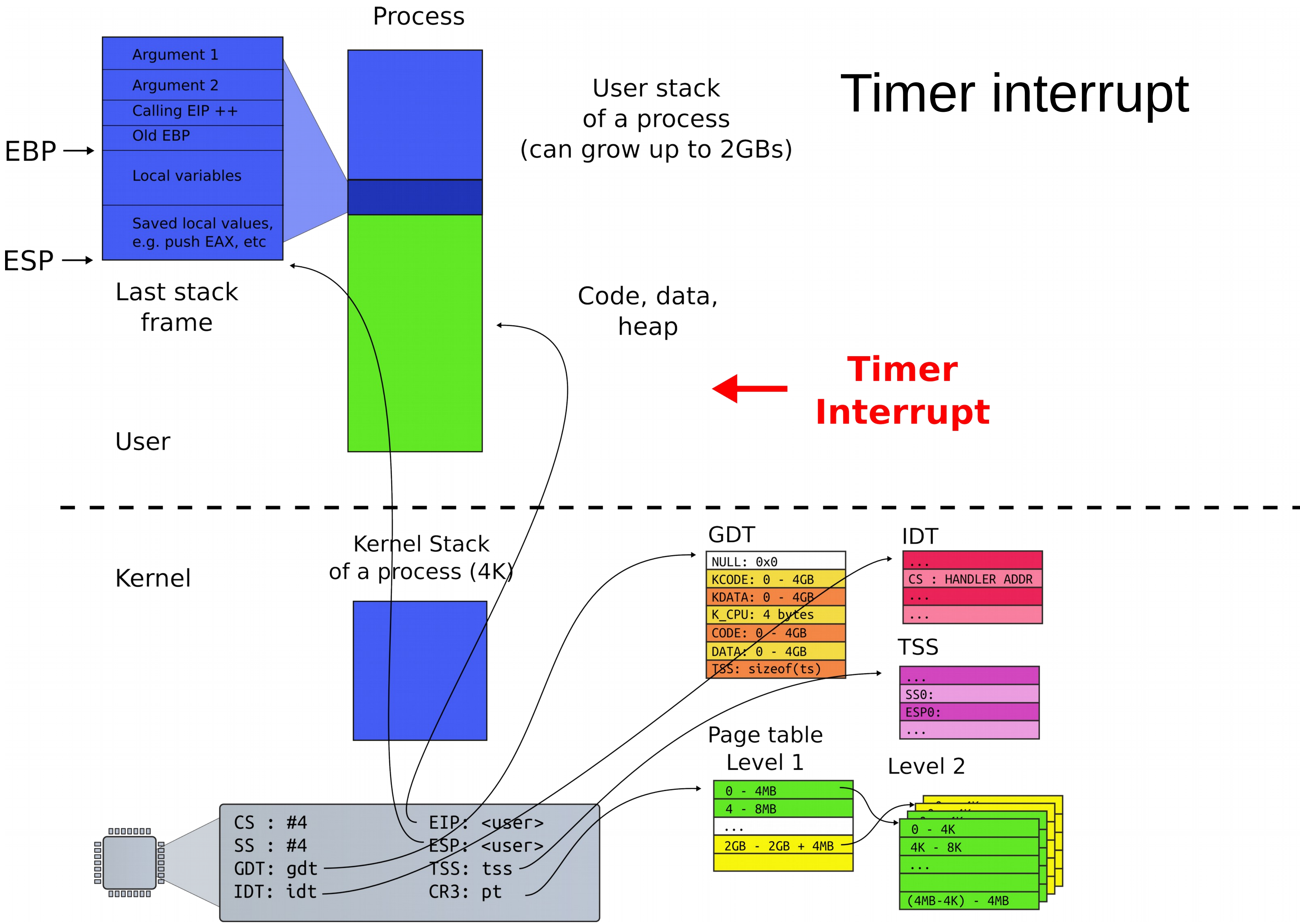




- User mode
- Two stacks
  - Kernel and user
  - Kernel stack is empty

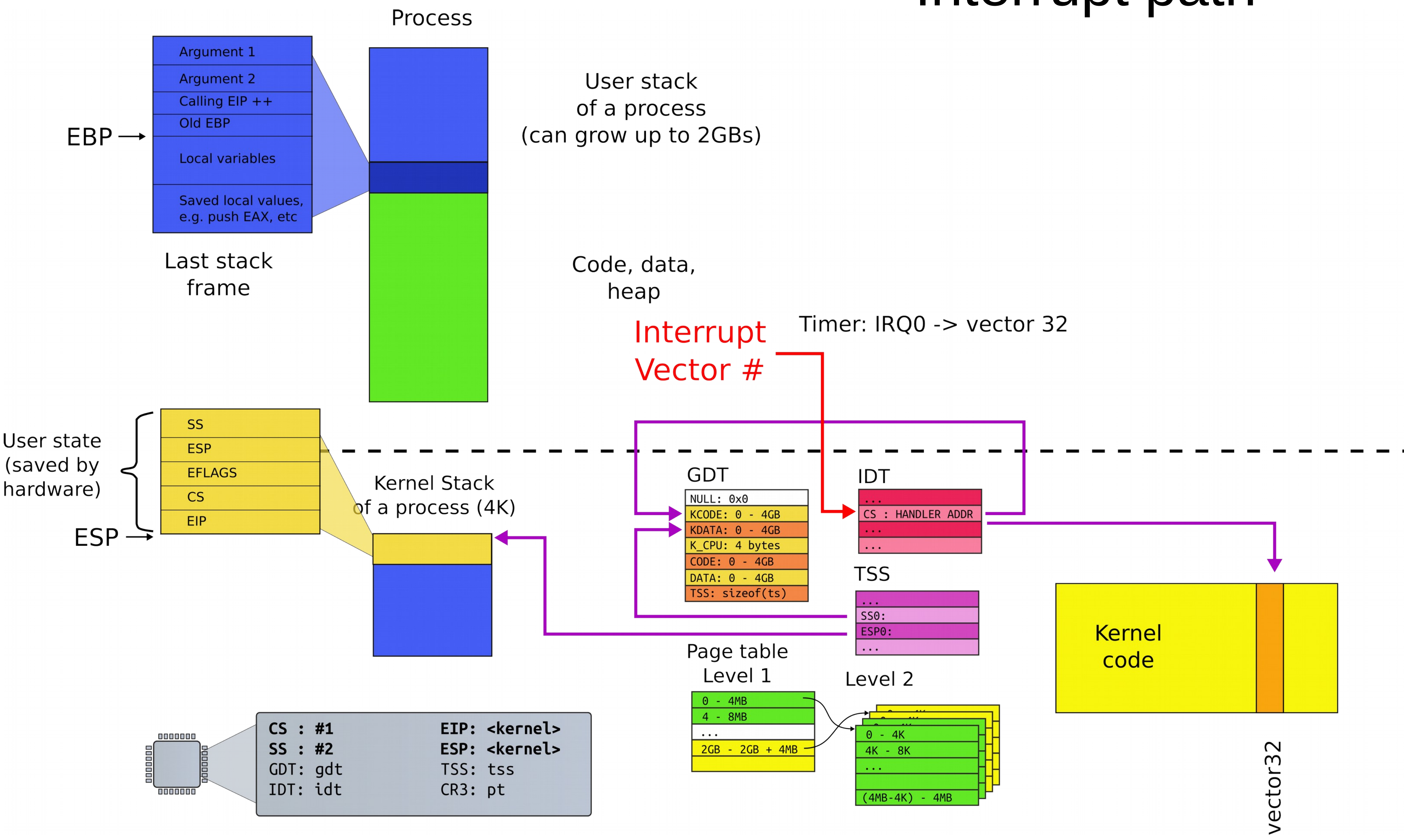
- Page table
- GDT





Timer interrupt
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
Timer Interrupt
User
Kernel
Kernel Stack of a process (4K)
GDT
NULL: 0x0
KCODE: 0 - 4GB
KDATA: 0 - 4GB
K_CPU: 4 bytes
CODE: 0 - 4GB
DATA: 0 - 4GB
TSS: sizeof(ts)
IDT
...
CS : HANDLER ADDR
...
...
TSS
...
SS0:
ESP0:
...
Page table Level 1
0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB
Level 2
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
CS : #4
SS : #4
GDT: gdt
IDT: idt
EIP: <user>
ESP: <user>
TSS: tss
CR3: pt

Interrupt path

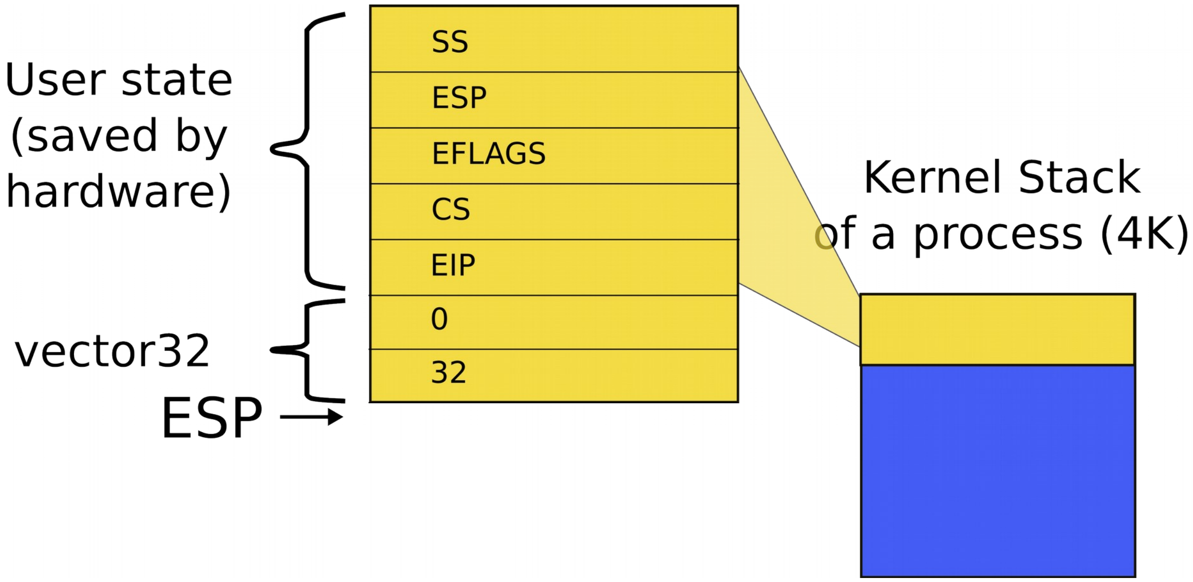


Where does IDT (entry 32) point to?

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

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

Kernel stack after interrupt

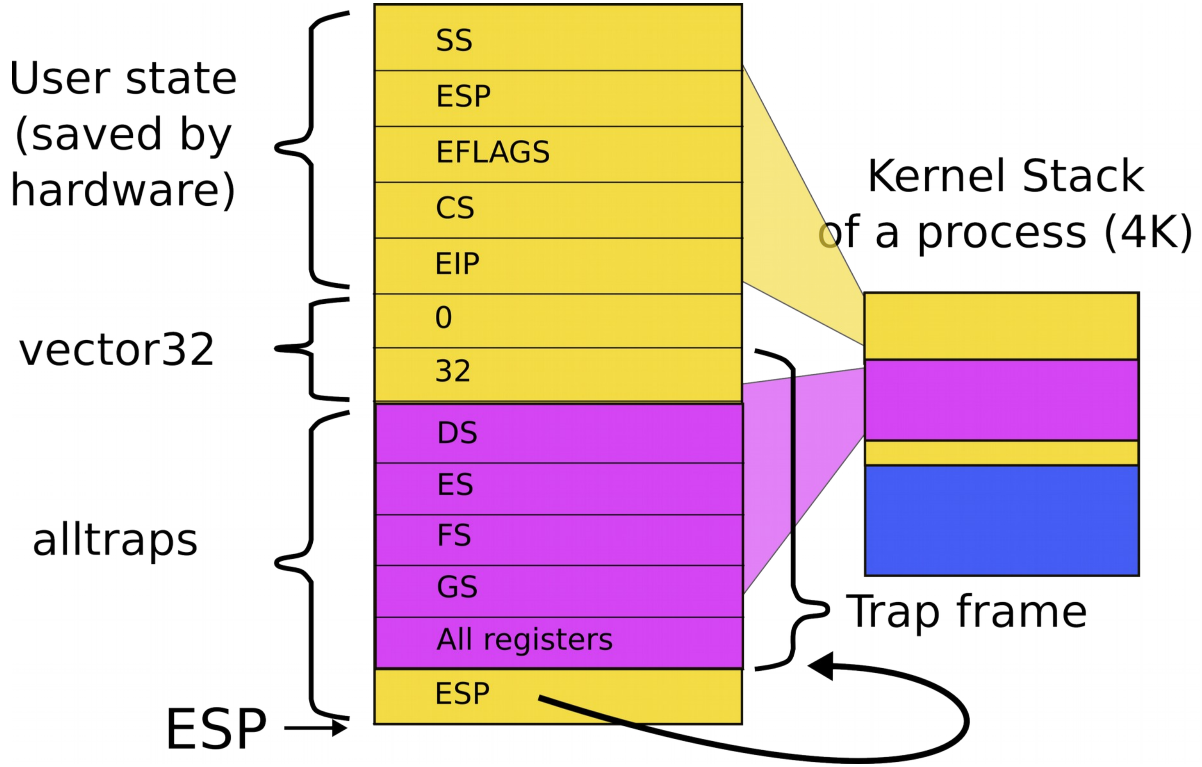


Call stack: vector32()

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

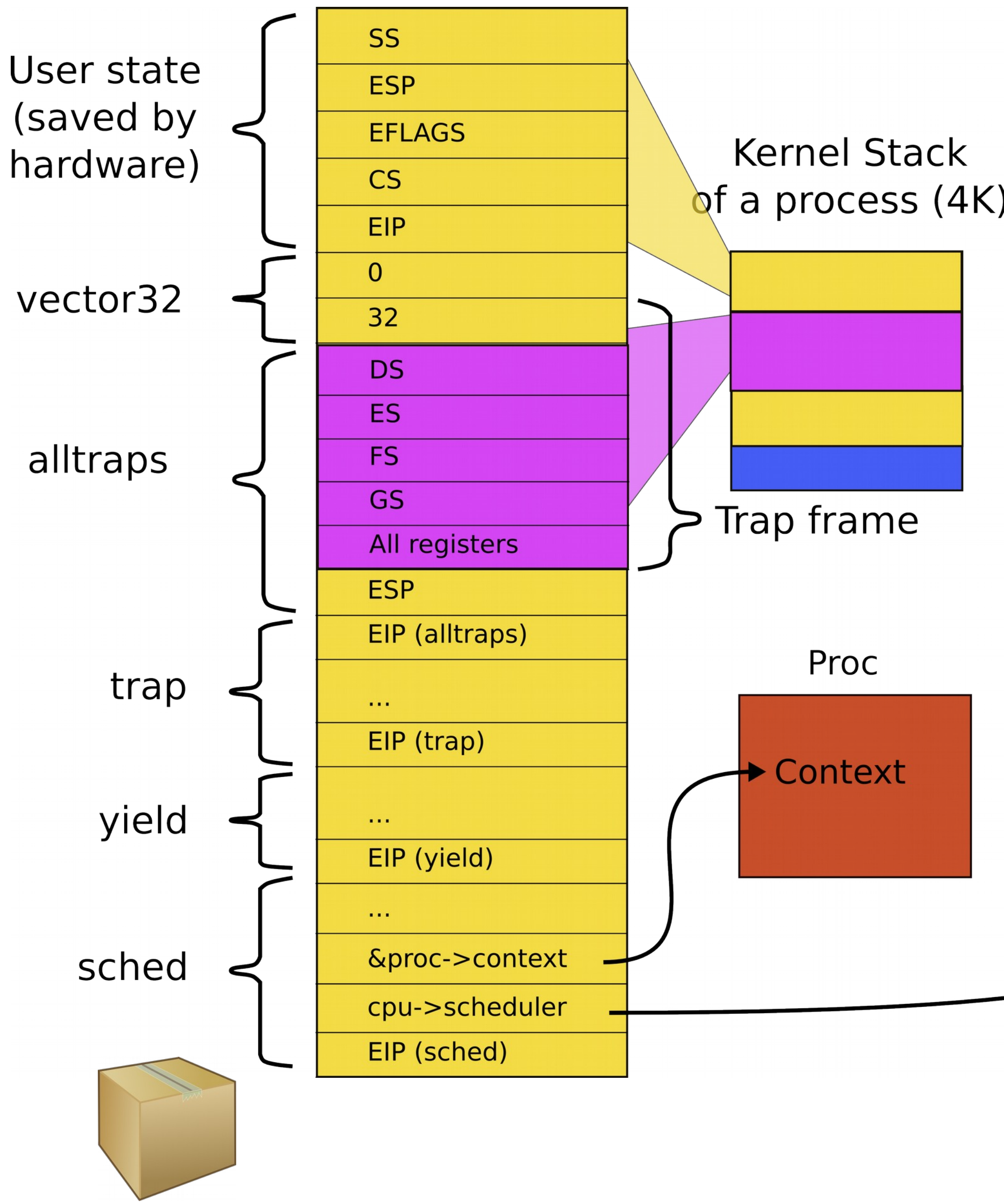
```
yield(void)
{
    acquire(&ptable.lock);
    proc->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```

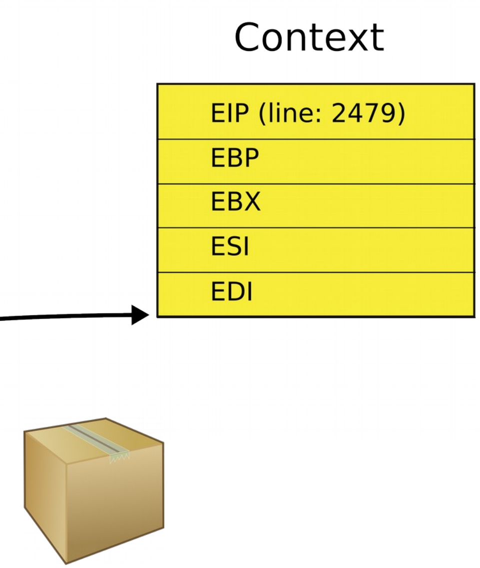
Start the context switch

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

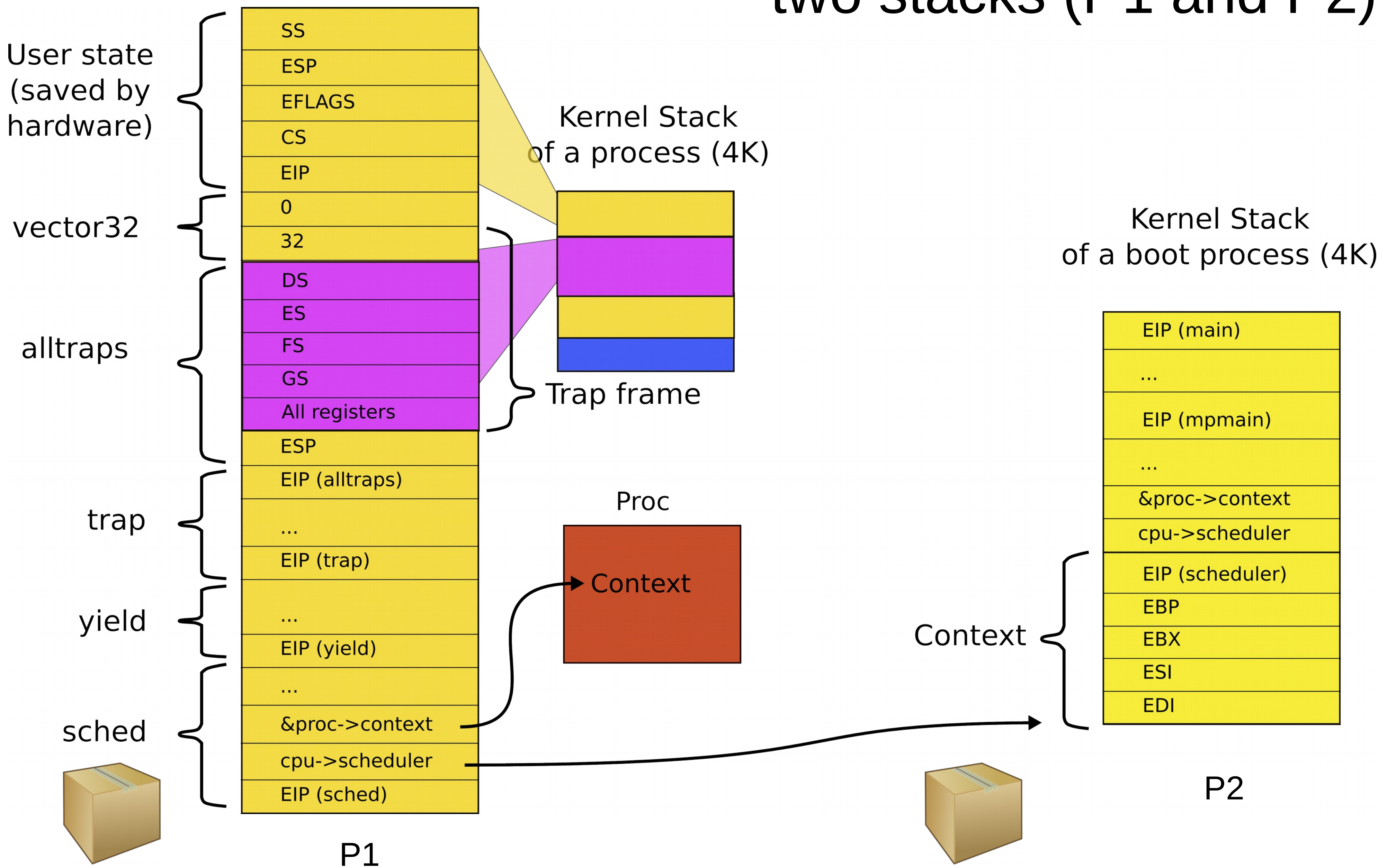
Stack inside swtch() and its two arguments (passed on the stack)

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





Remember you have two stacks (P1 and P2)



The context switch function should
pack everything what is left (the context!) in the box and switch stacks, i.e., save the pointer to the old stack (P1) and load the new stack (P2)

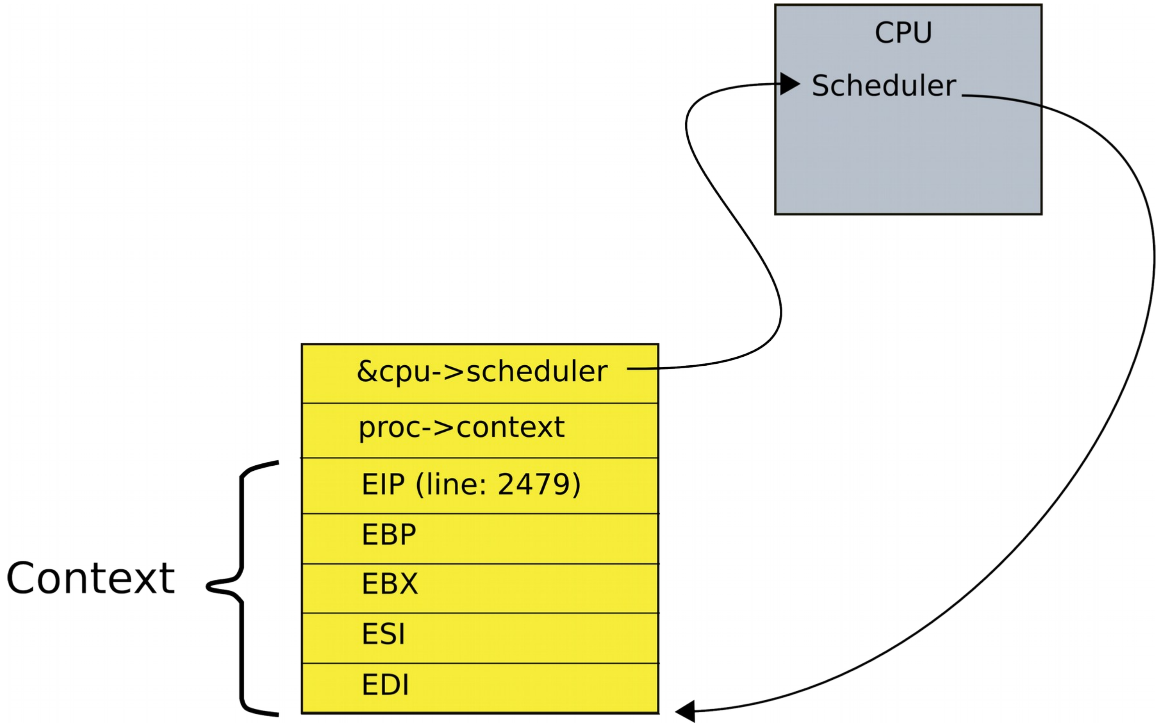
swtch(): save registers on the stack

```
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;
};
```

Context data structure

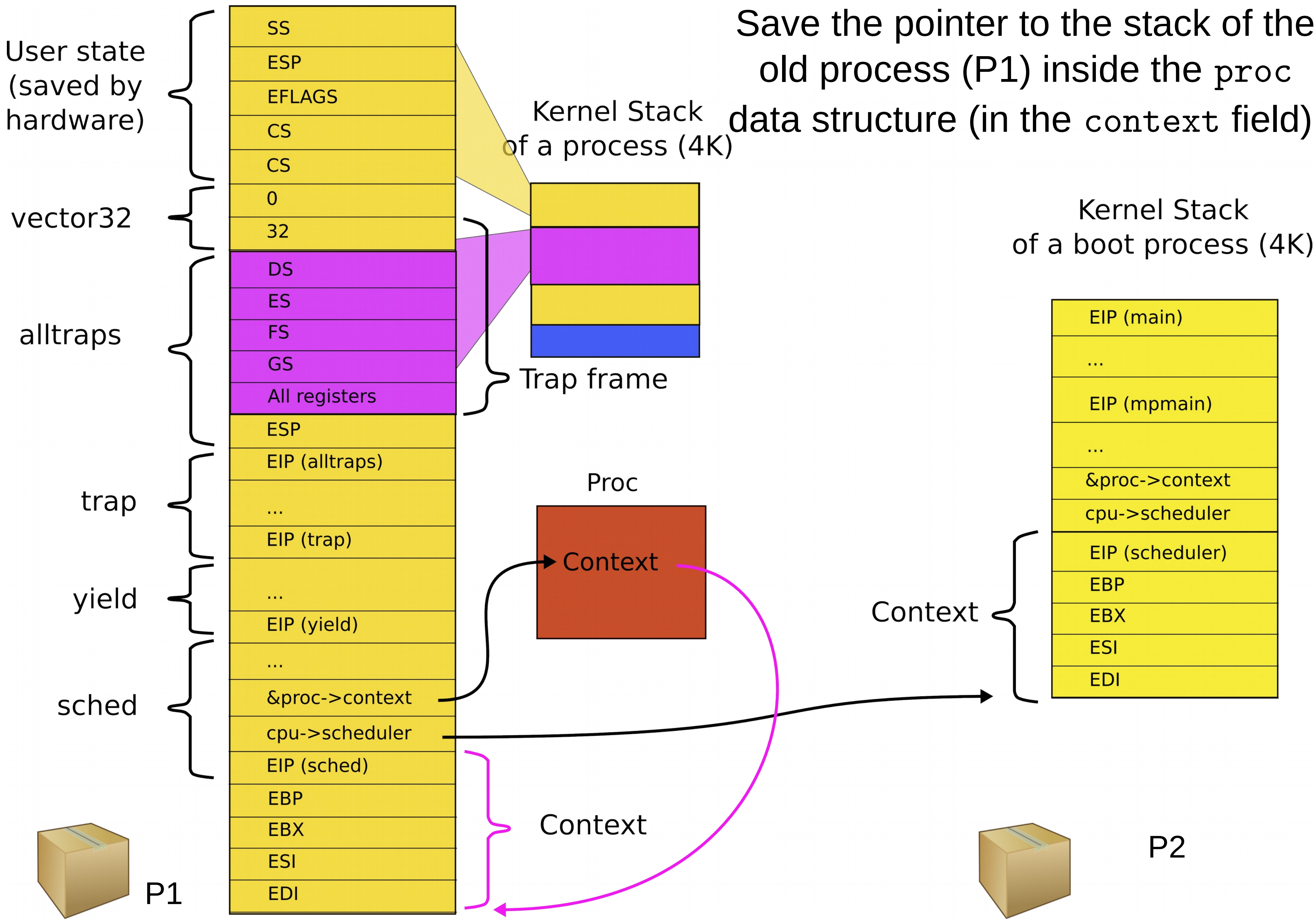
```
2093 struct context {
2094   uint edi;
2095   uint esi;
2096   uint ebx;
2097   uint ebp;
2098   uint eip;
2099 };
```



Main trick: context is always saved on the top of a stack

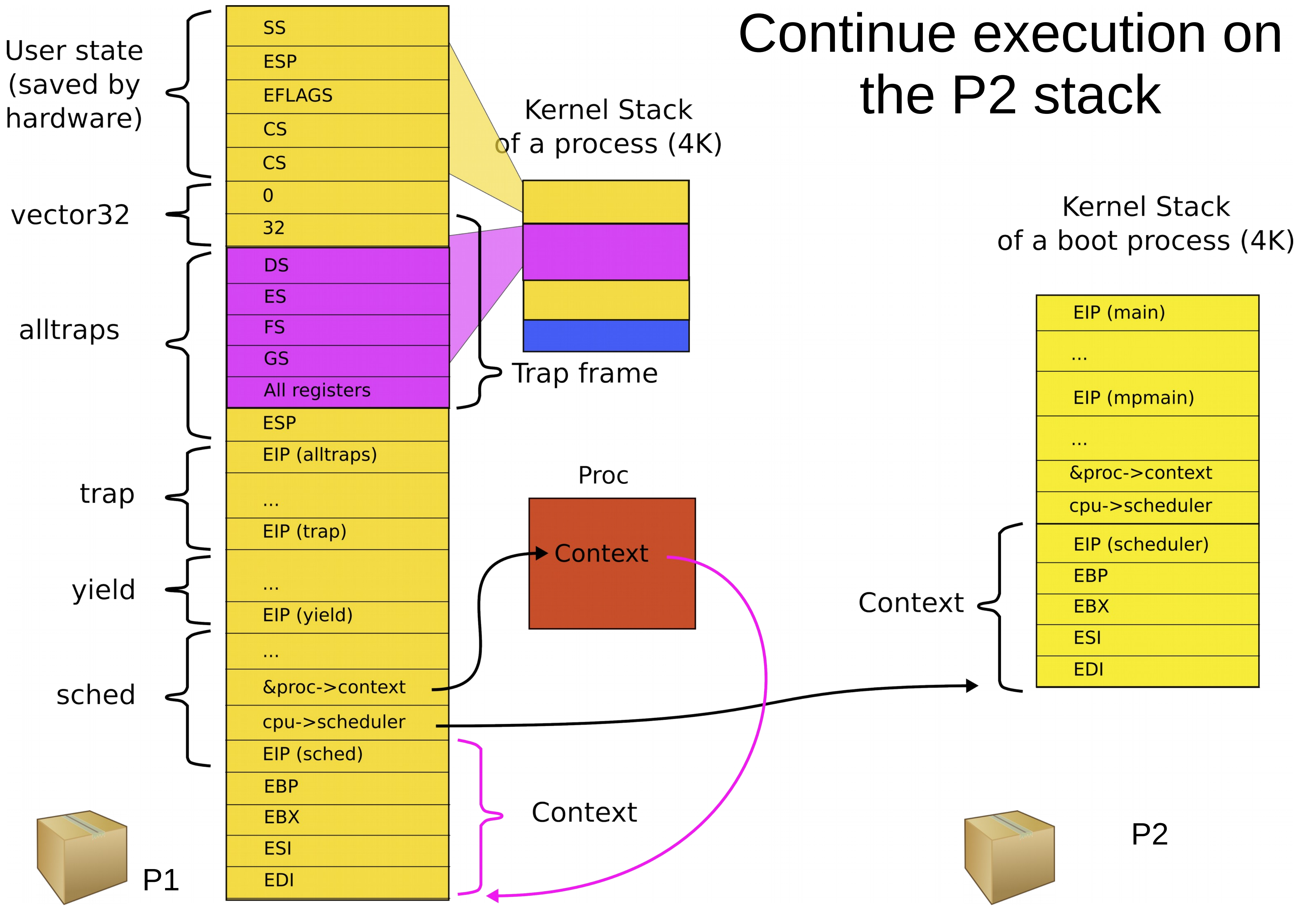
And the context switch just saves the old context and loads the new

Save the pointer to the stack of the old process (P1) inside the `proc` data structure (in the `context` field)



Now you can simply load the new context (P2) into ESP and continue returning on that new stack

Continue execution on the P2 stack



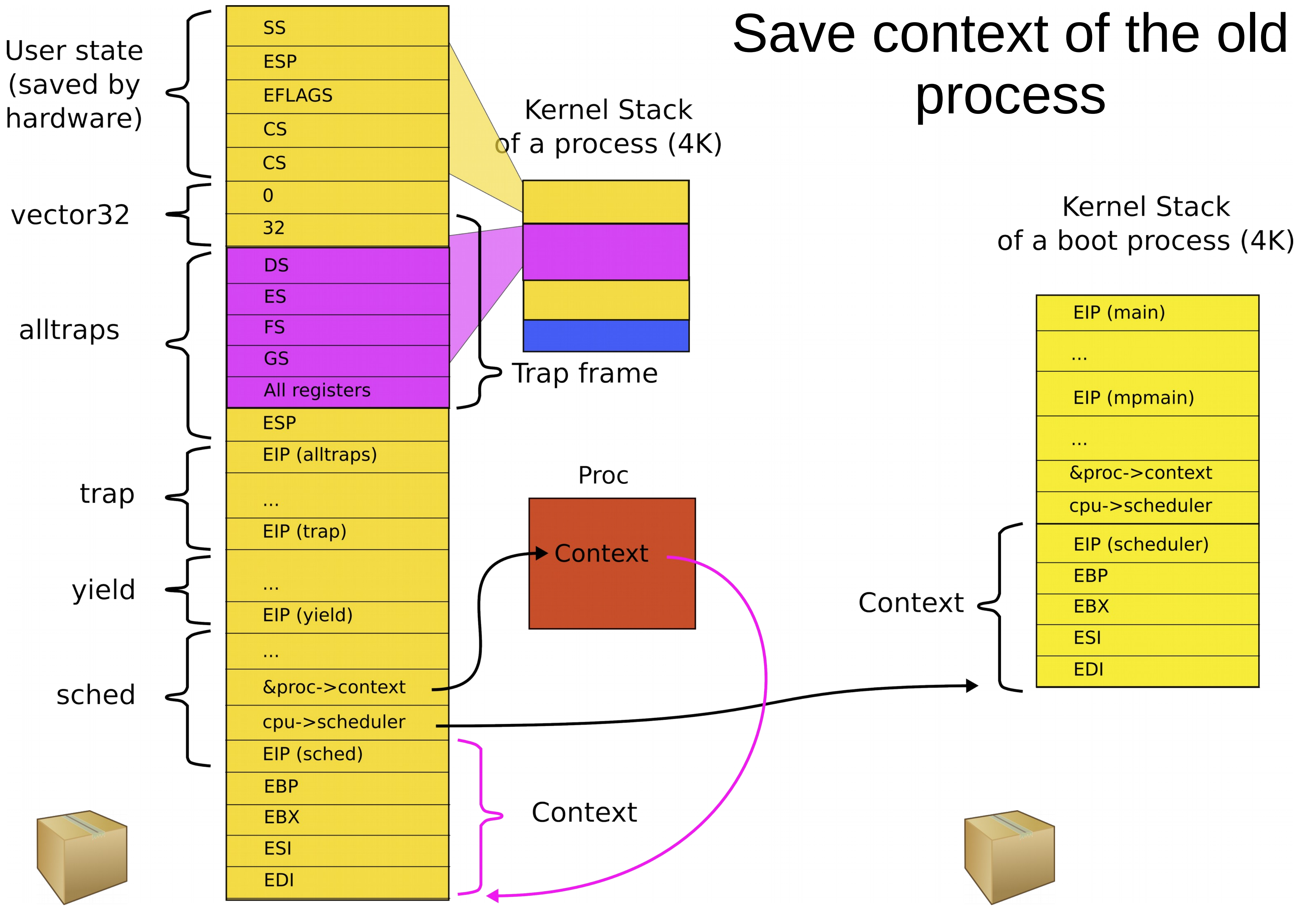
```
swtch:
movl 4(%esp), %eax  # **old
movl 8(%esp), %edx  # *new

# 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
```

swtch()

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

- First argument:
  - A pointer to a pointer to a context
  - Or in other words: a pointer to a memory location that can hold address of the context
  - We'll save the address of the current context there
- Second argument:
  - A pointer to a context of the next process
  - We'll load it into the ESP register switching to the next process

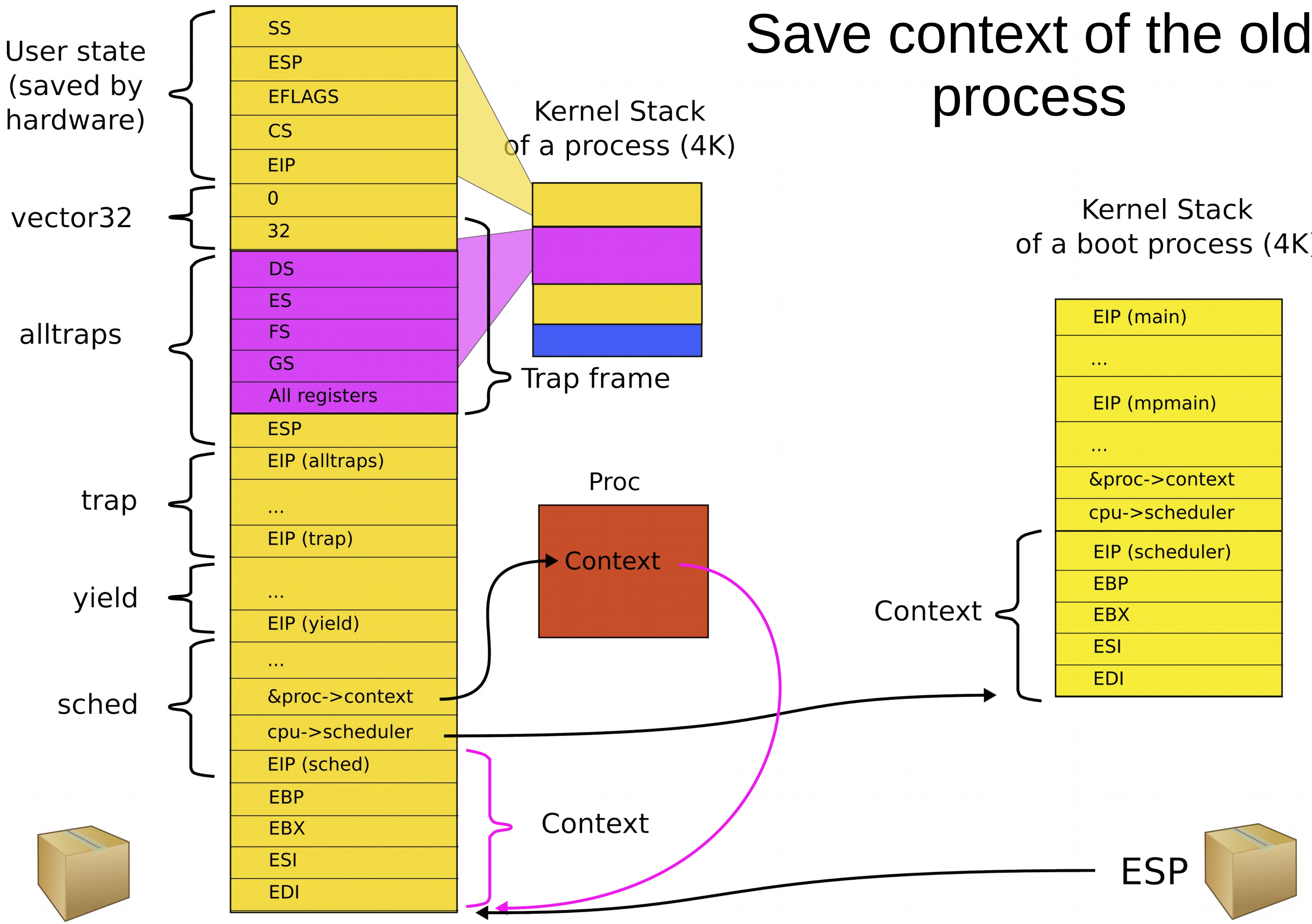
Save context of the old process



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
```

Save context of the old process

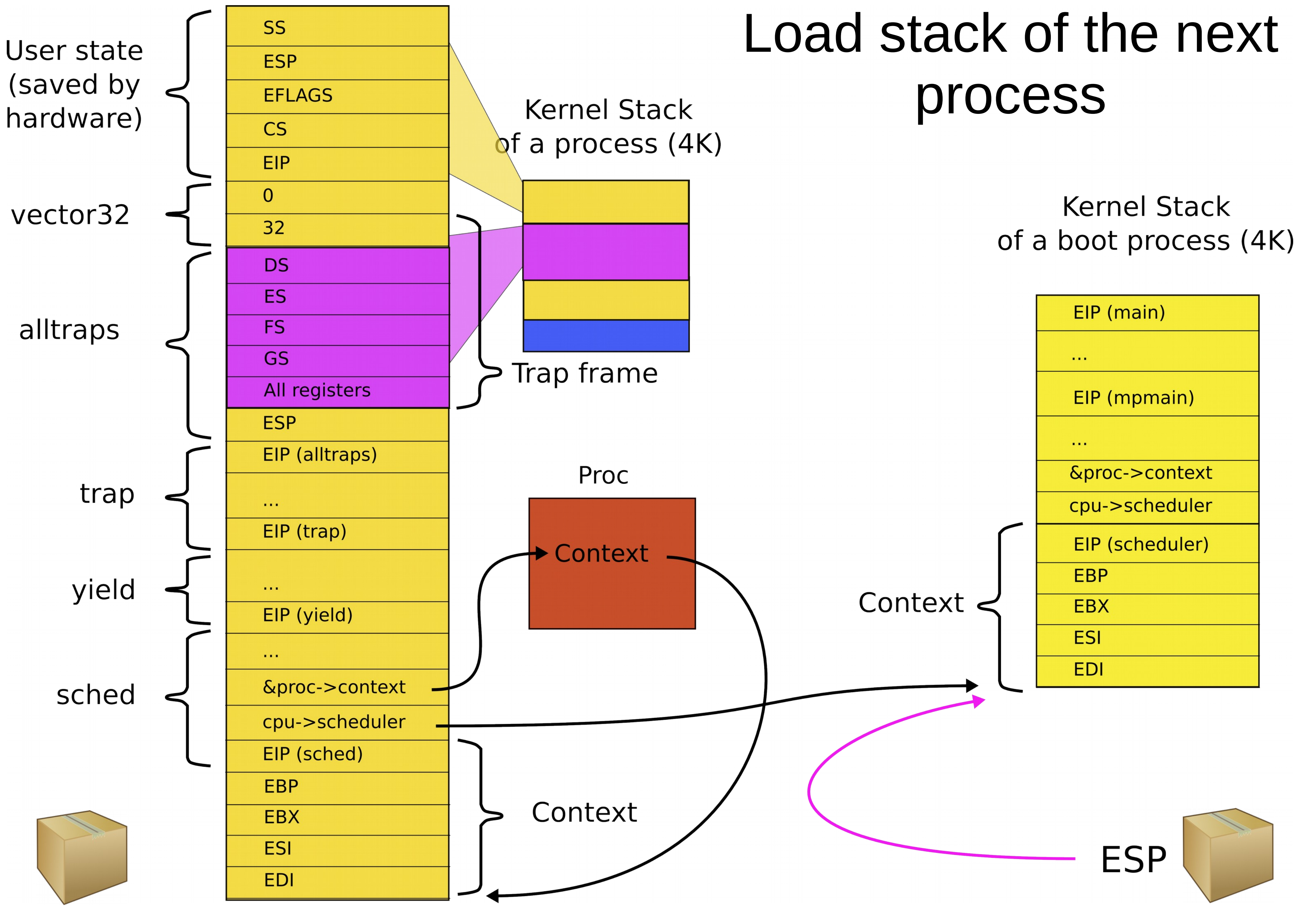


swtch(): load next context

```
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)   // save 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
```

- Load address of the next context (it's in %edx) into %esp

Load stack of the next process



Remember: The context switch function should just save the pointer to the old stack (P1) and load the new stack (P2)

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?

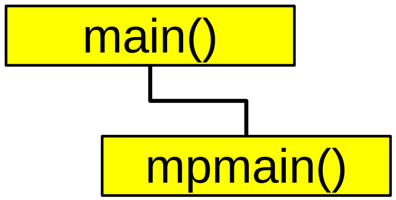
Context is always top of some stack...ok, but how?

- How does initialization of each CPU end?

```
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
...
  seginit(); // segment descriptors
...
  tvinit(); // trap vectors
...
  userinit(); // first user process
  mpmain(); // finish this processor's setup
}
```

main()

```
// 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 the scheduler

Scheduler()

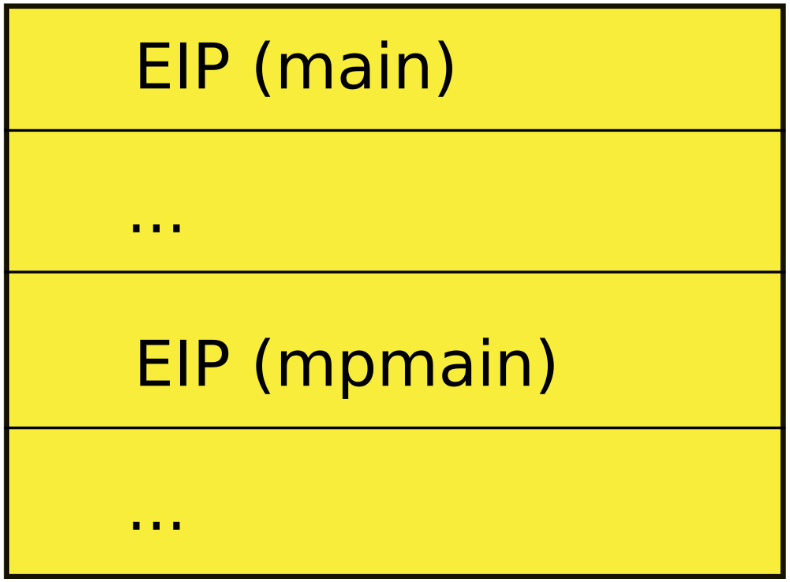
```
scheduler(void)
{
  for(;;){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      proc = 0;
    }
  }
}
```

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

```
struct cpu {
  uchar apicid;                // Local APIC
  struct context *scheduler;  // swtch() here to enter scheduler
  struct taskstate ts;         // TSS
  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 ...
  struct proc *proc;           // The process running on this cpu
};

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

This is how the stack looked after boot finished, i.e., inside mpmain()

Kernel Stack
of a boot process (4K)



- So when the scheduler context switched the first time

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

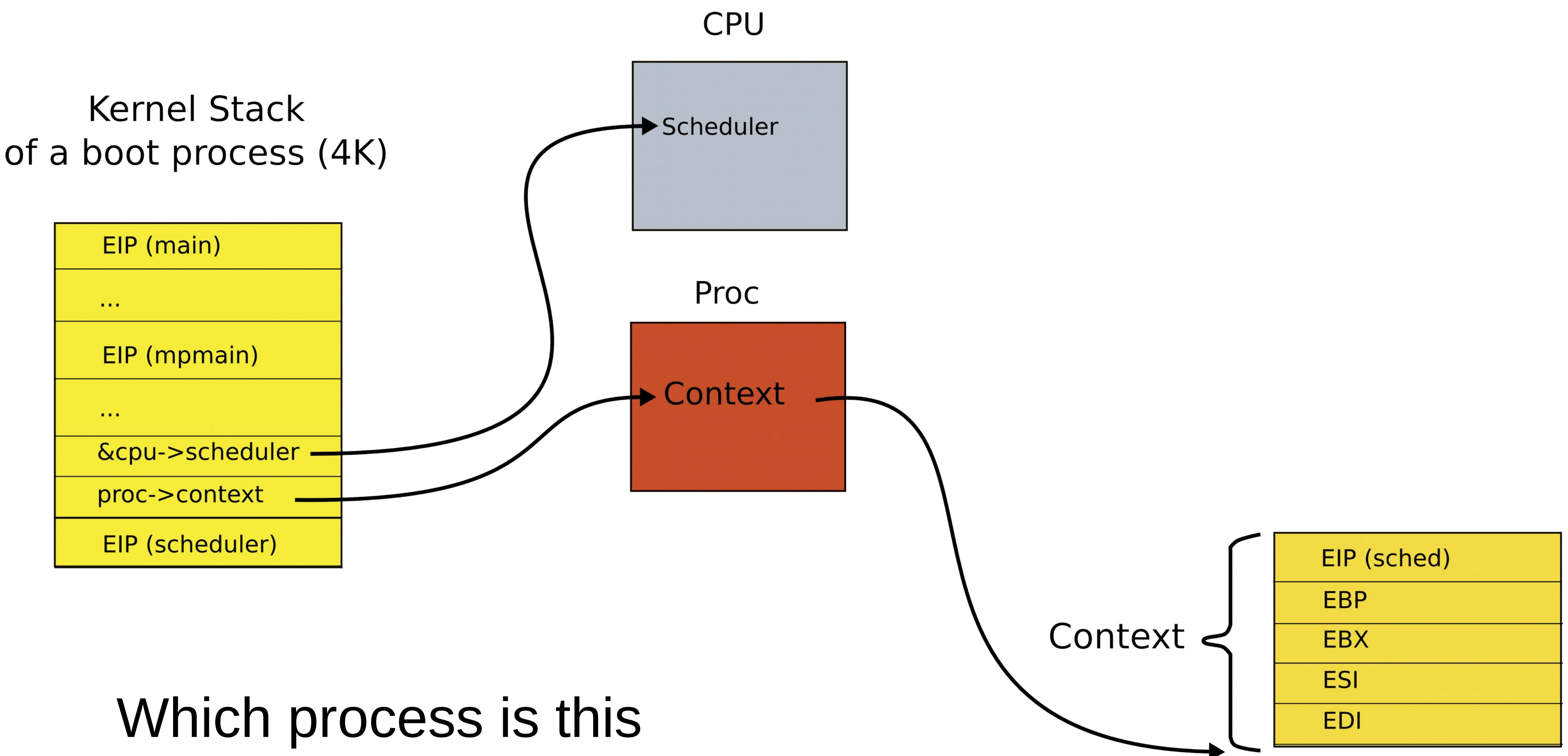
- We save the current context of the scheduler into:

```
&cpu->scheduler
```

- And restore the context of the first process

```
proc->context
```

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



Which process is this context?

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

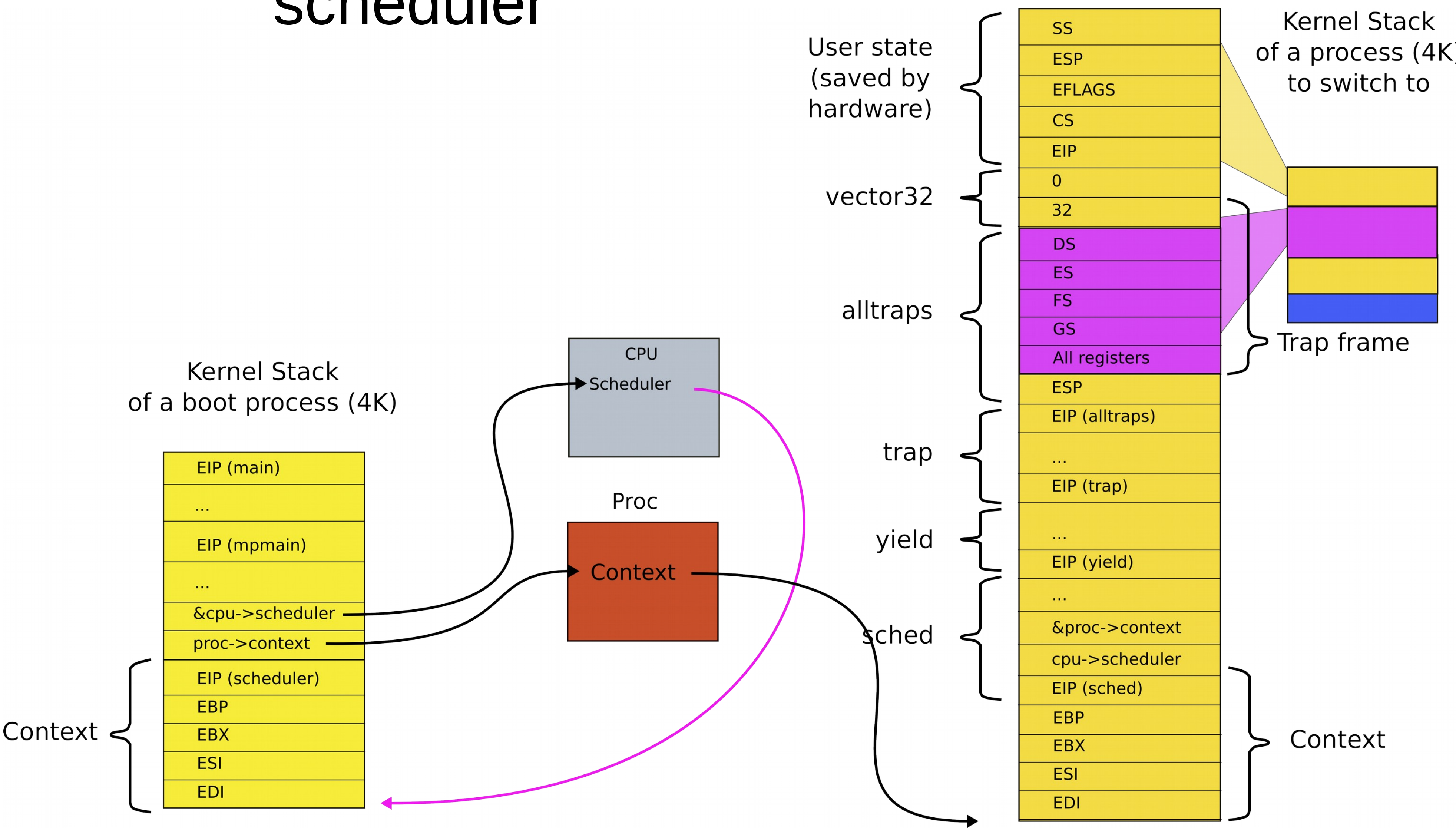
- Which process is this context?
- It's the context of the first process scheduler decides to run

```c
allocproc(void)
{
...
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
...
}
```

Context is configured as top of the stack when new process is created inside allocproc() function

- Remember exec()?

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 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
```

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 the context switch
(end of the detour)

Where does this swtch() return?

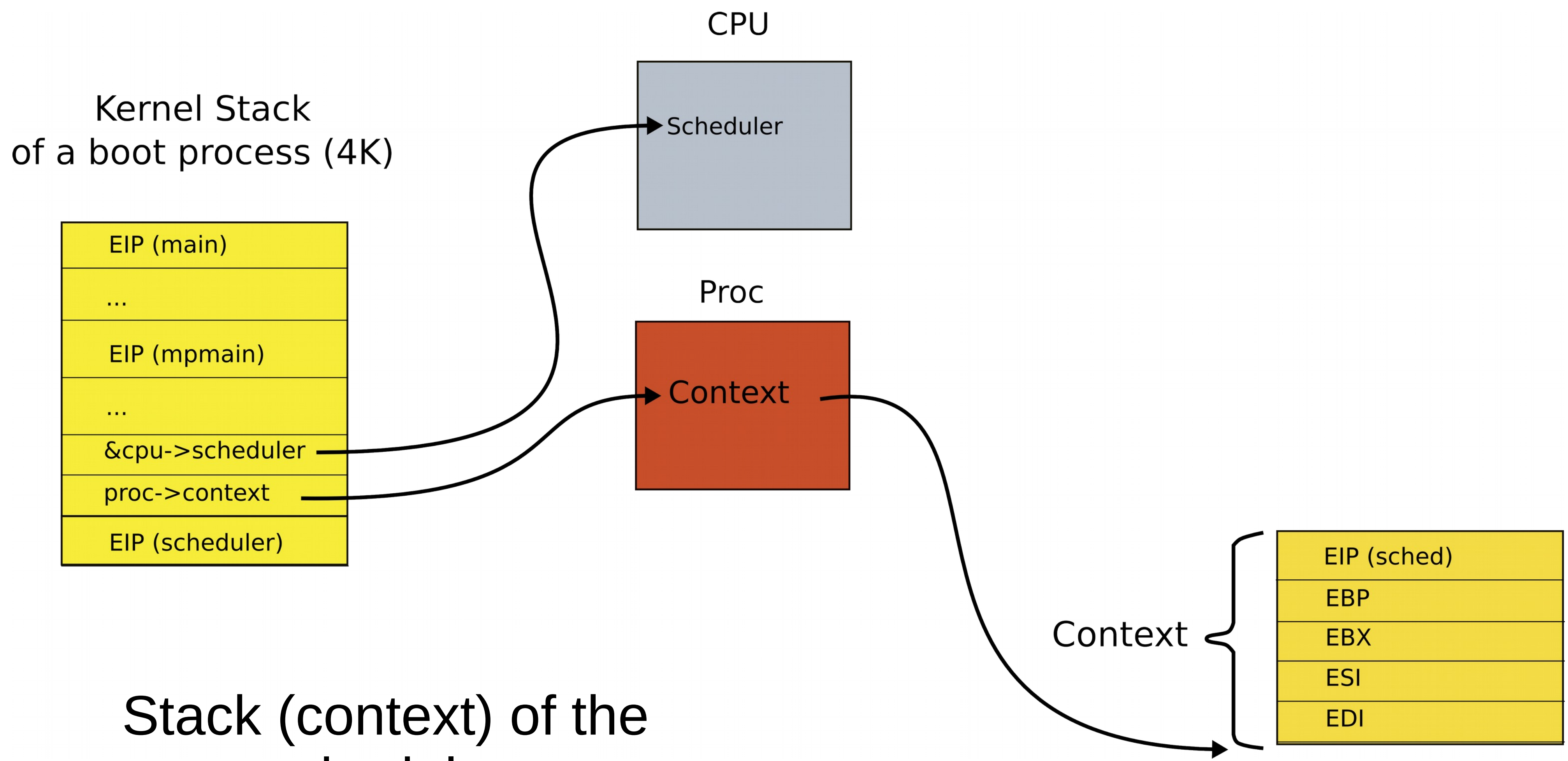
- Scheduler!
- After all remember
  - We started with timer interrupt
  - Entered the kernel
  - Entered schedule()
  - Entered switch
- And are currently on our way from the process into the scheduler

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;
    }
  }
}
```

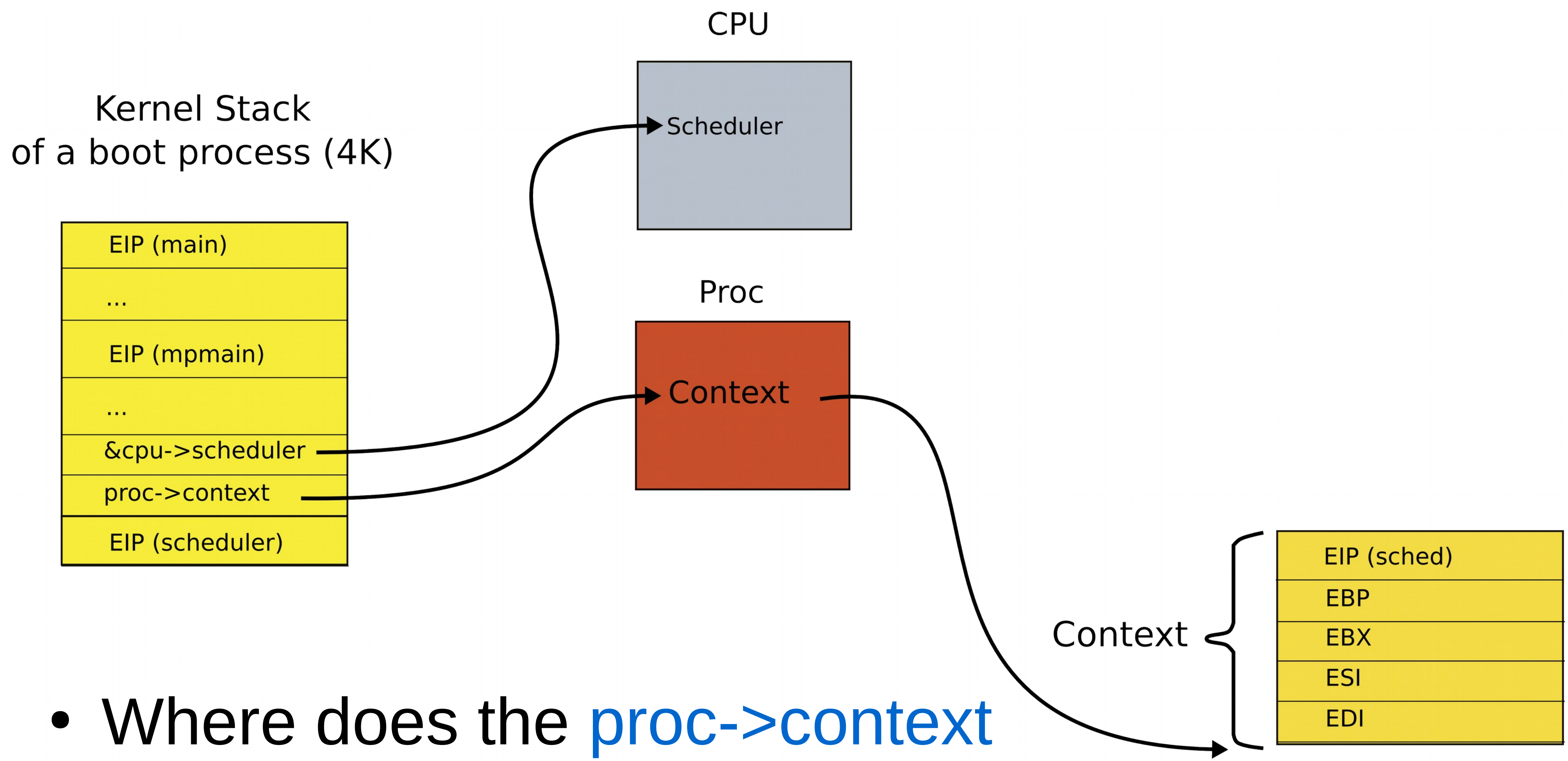
- Chooses next process to run
- Switches to it

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



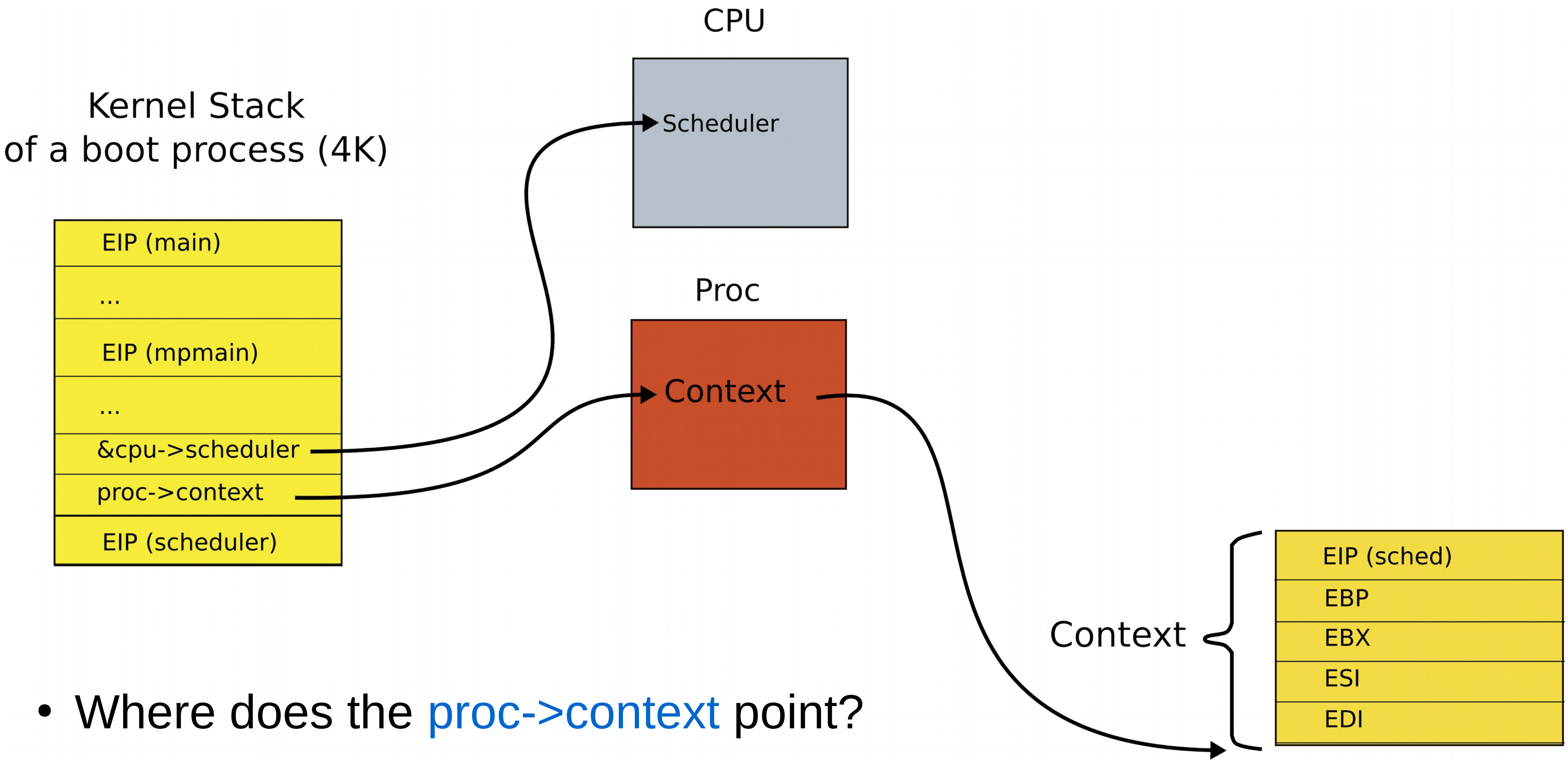
Stack (context) of the scheduler

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



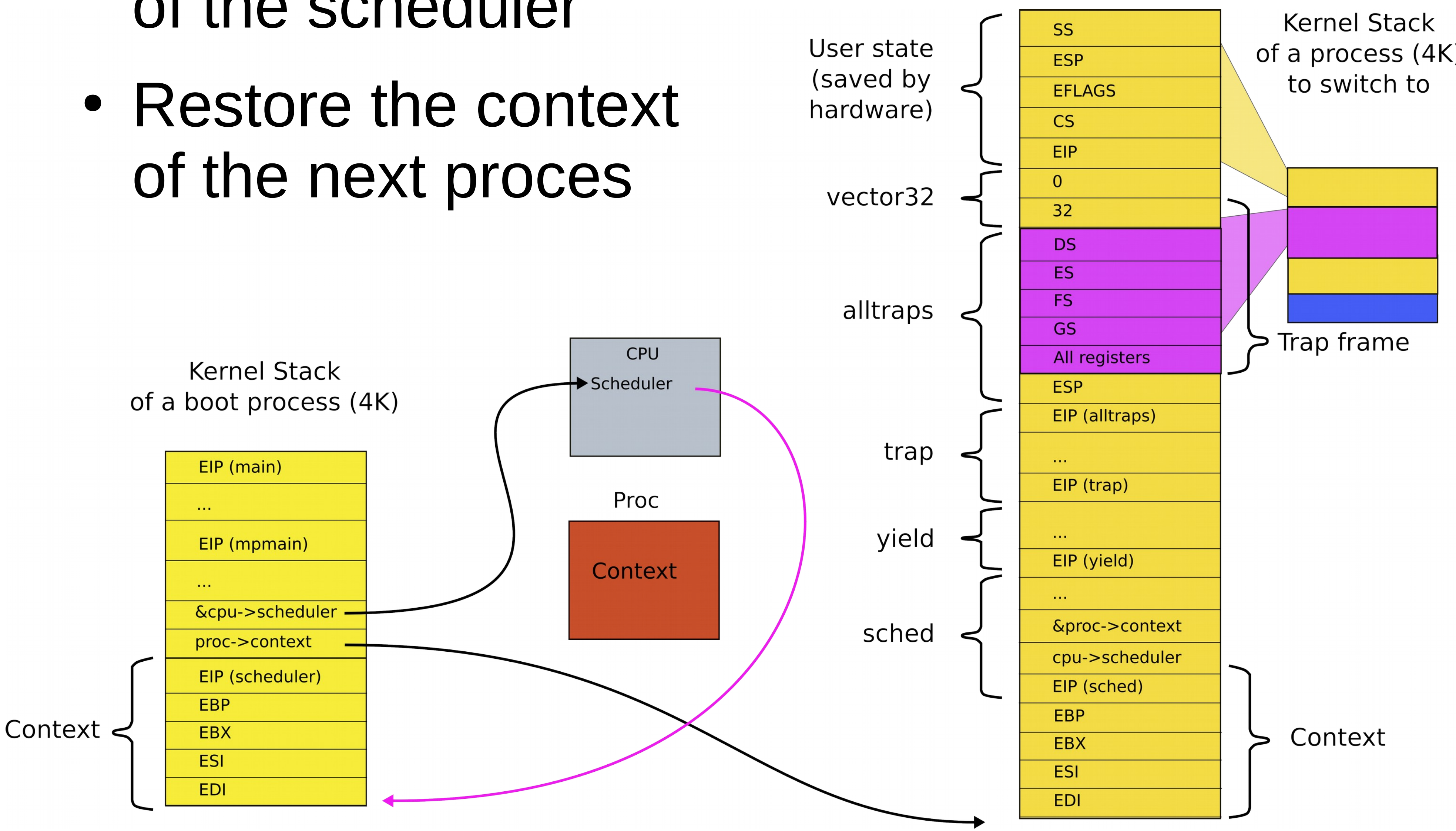
- Where does the proc->context point?

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



- Where does the proc->context point?
  - Right the context (stack) of the next process to run

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



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

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

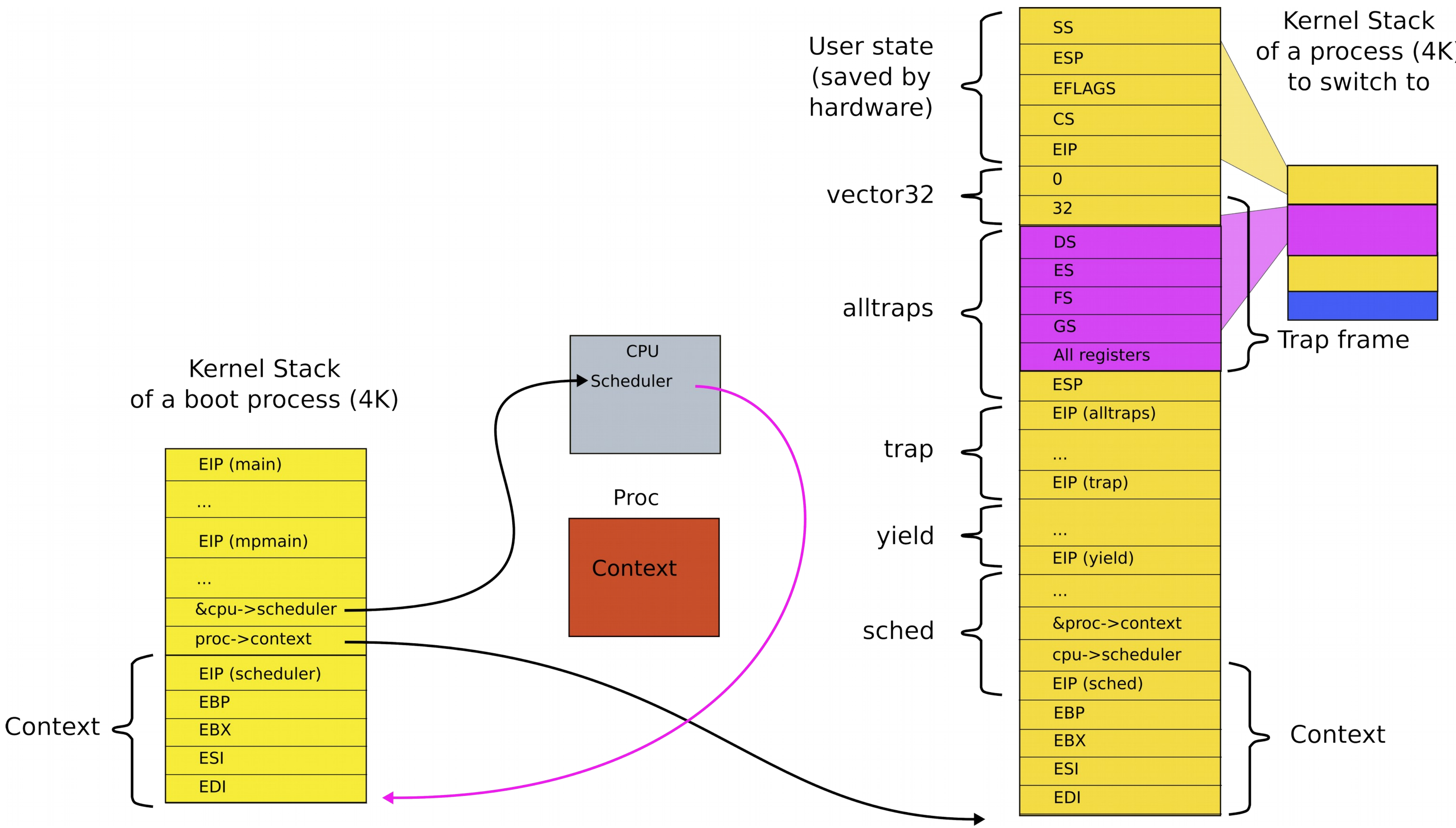
- Hence, we save context of the scheduler into

```
&cpu->scheduler
```

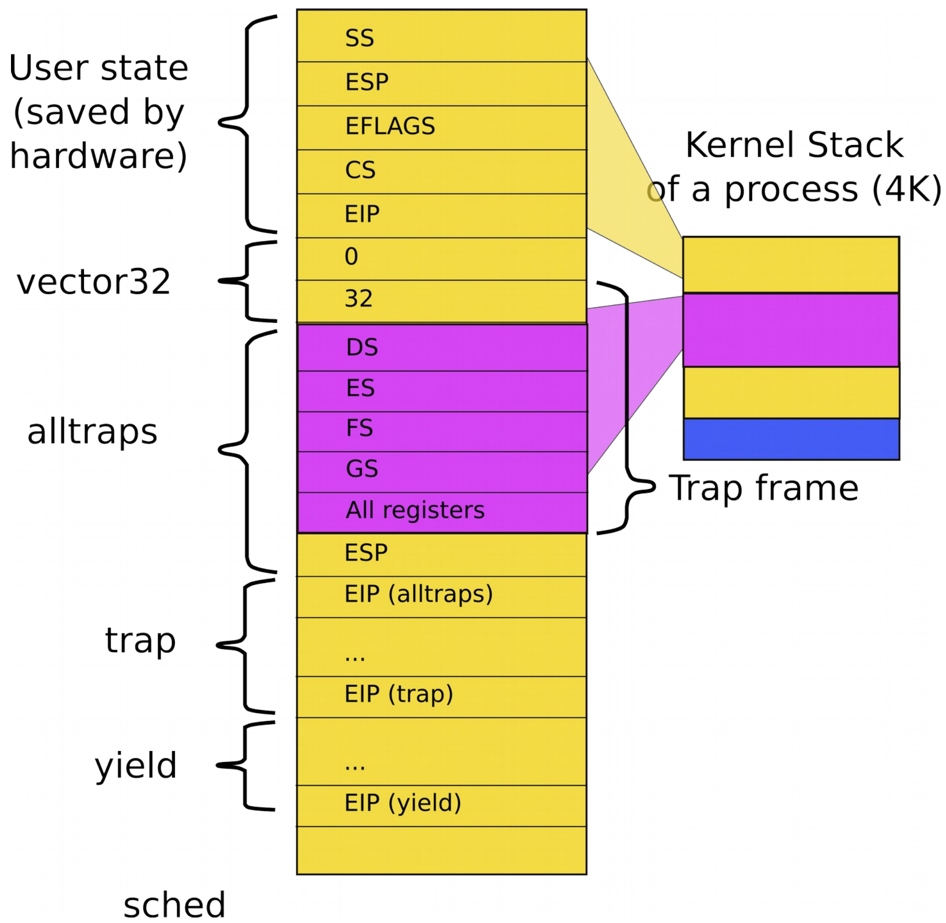
- And restore

```
proc->context
```

Stacks and contexts inside the swtch()



Exiting back to user-level

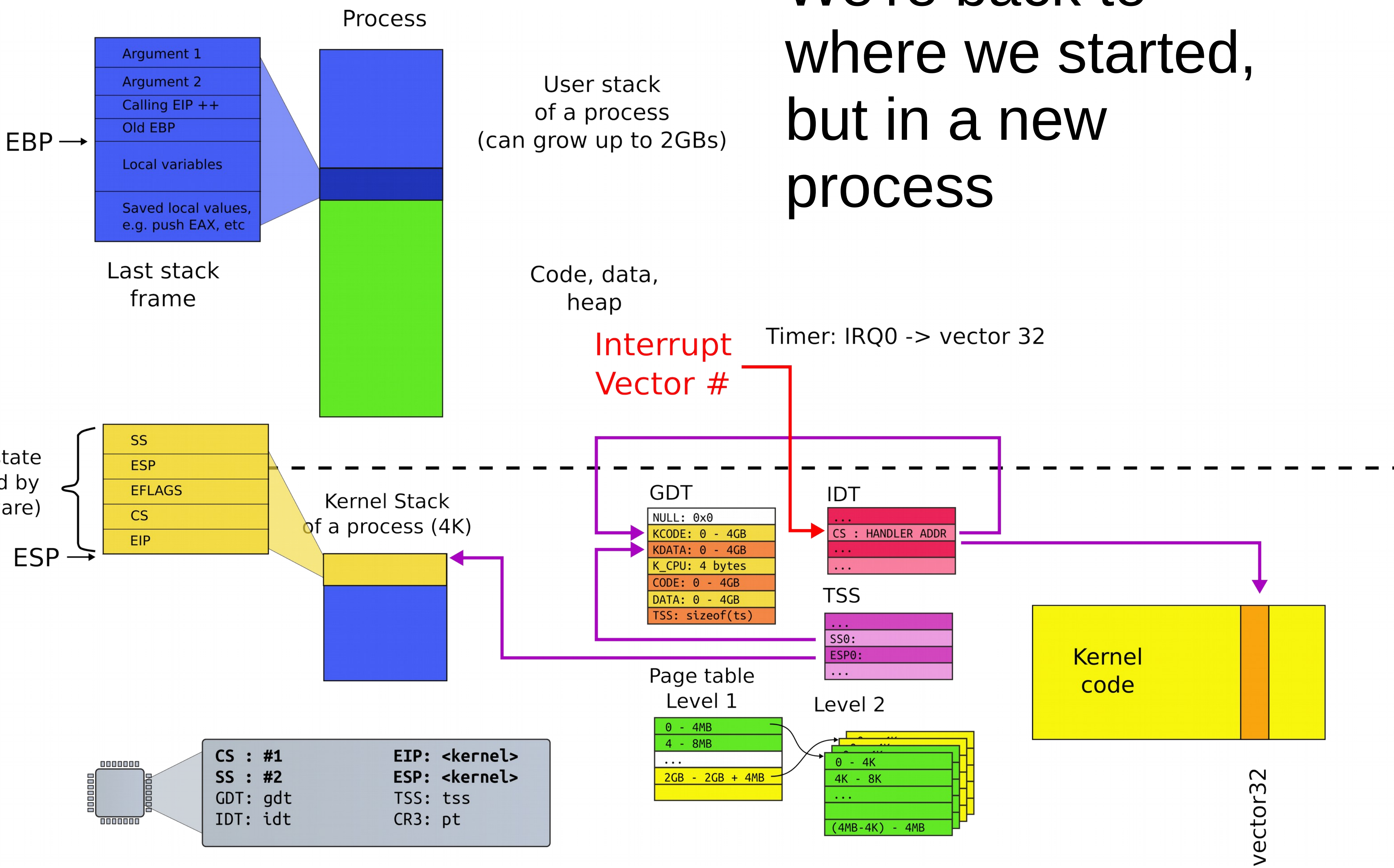


- Stack of the process after context switch, i.e., inside sched()
  - Return as usual all the way to alltrap()

alltraps(): exit into user-level

```
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
```

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



Summary

- We switch between processes now

Thank you