cs5460/6460: Operating Systems

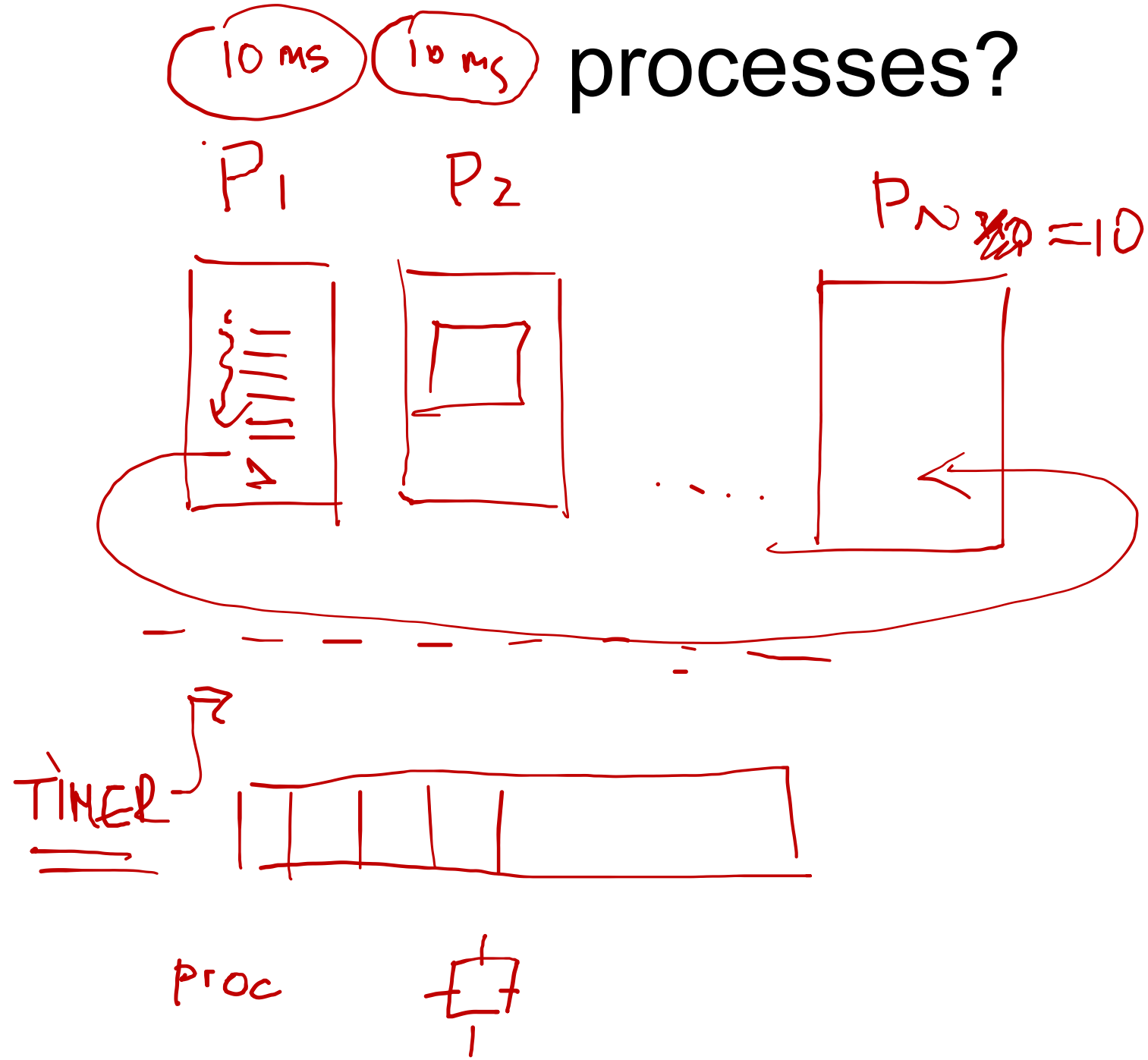
Lecture 11: Context switch

Anton Burtsev

April, 2024

A bit of recap...

When OS context switches between processes?



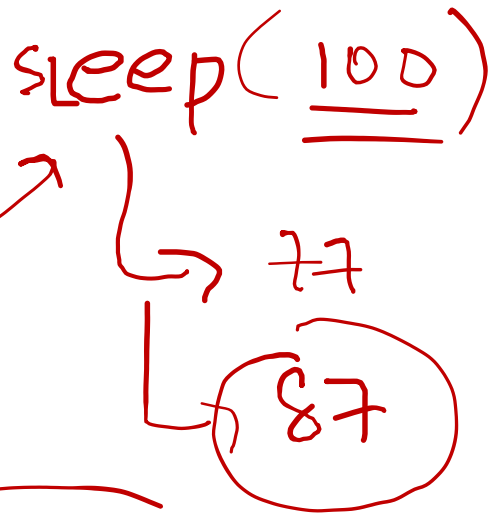
When OS context switches between processes?

- 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()

```
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();
```



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();
```

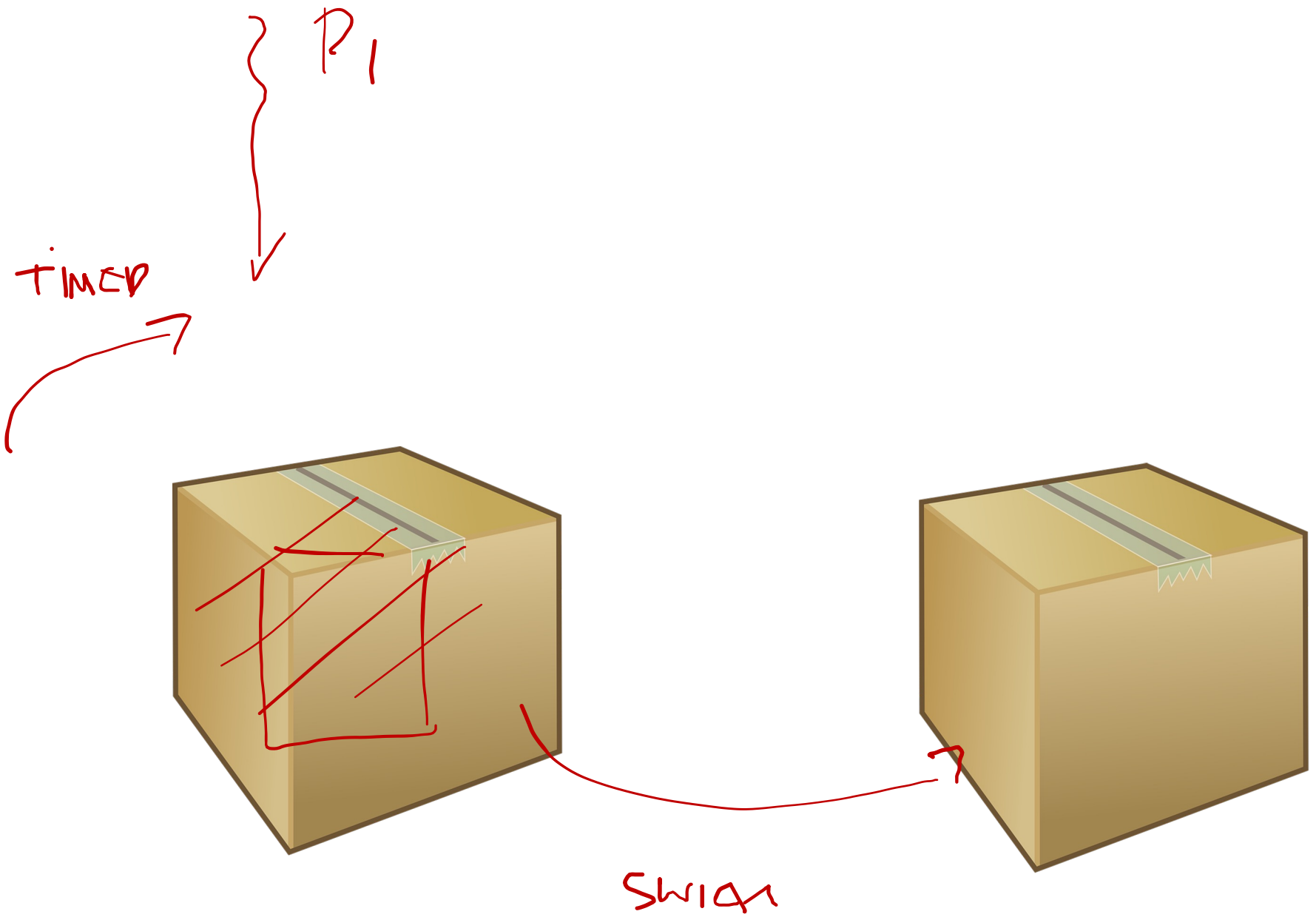
Invoke the scheduler

```
2777 yield(void)
2778 {
2779   acquire(&ptable.lock);
2780   proc->state = RUNNABLE;
2781   sched();
2782   release(&ptable.lock);
2783 }
```

Start the context switch

```
2758 sched(void)
2759 {
...
2771   swtch(&proc->context,
                cpu->scheduler);
...
2773 }
```

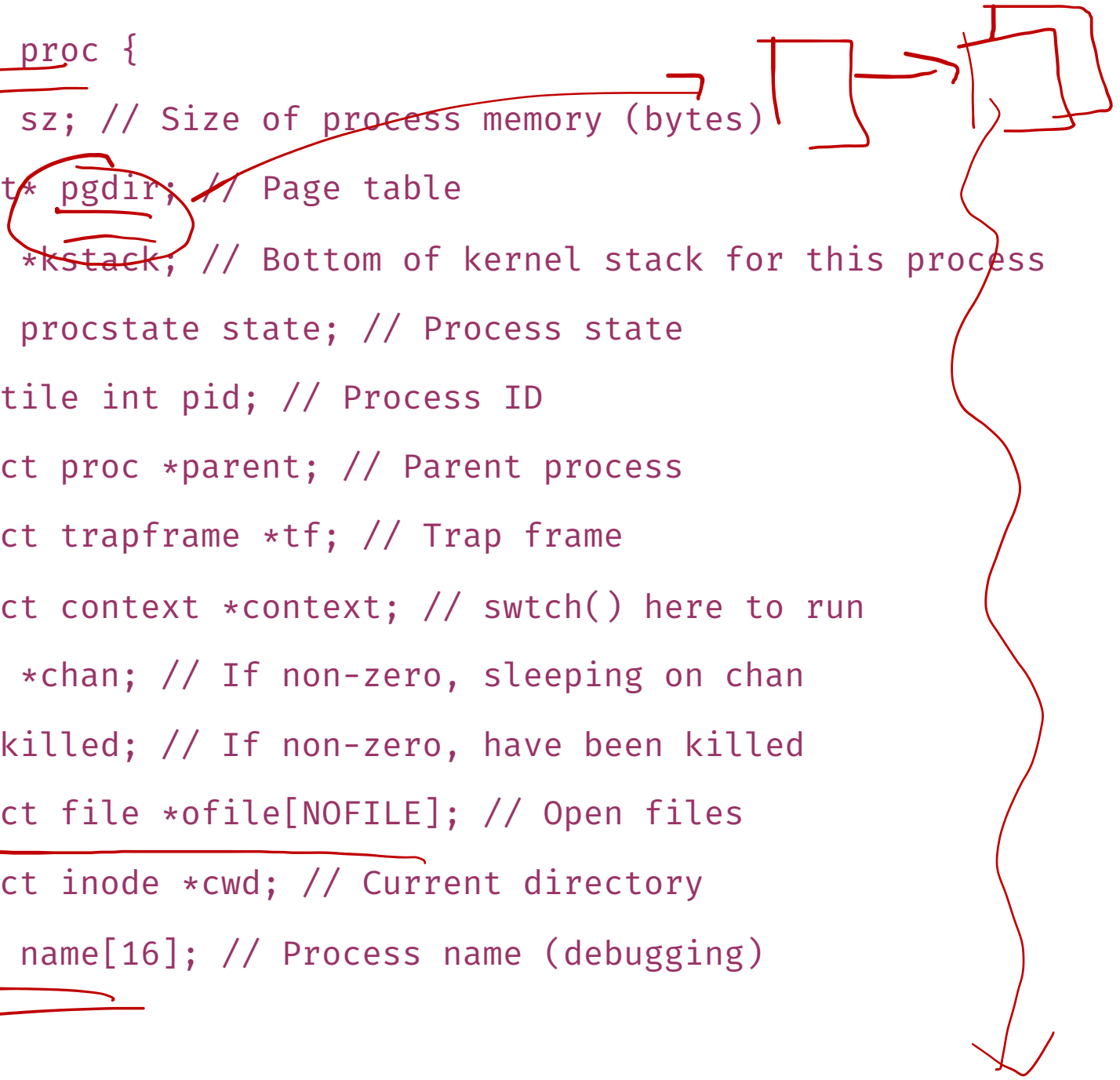
But what do you think needs to happen inside switch()?


P1
TIMER
Swap
Process 1
Process 2

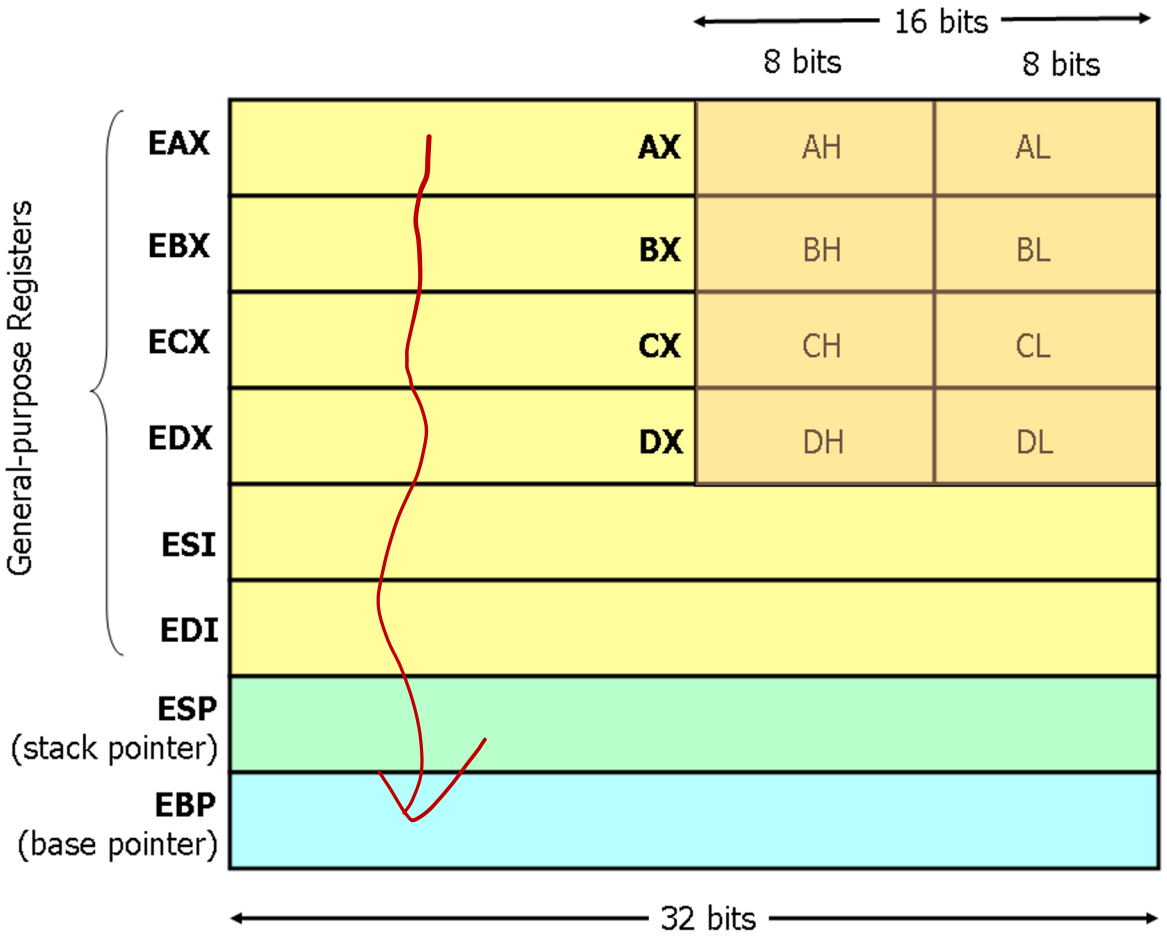
What should be in the box?

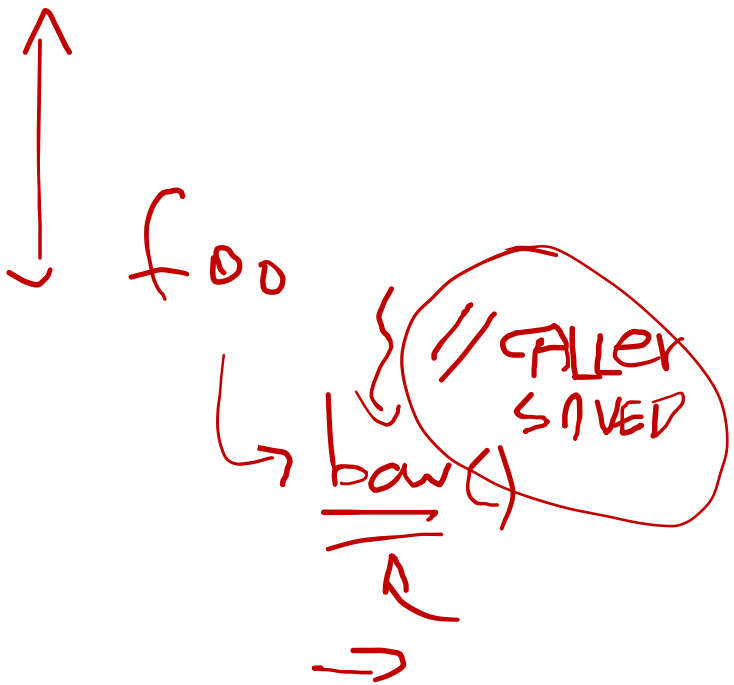
i.e., what is the state of the process?

```
2103 struct proc {
2104   uint sz; // Size of process memory (bytes)
2105   pde_t* pgdir; // Page table
2106   char *kstack; // Bottom of kernel stack for this process
2107   enum procstate state; // Process state
2108   volatile int pid; // Process ID
2109   struct proc *parent; // Parent process
2110   struct trapframe *tf; // Trap frame
2111   struct context *context; // swtch() here to run
2112   void *chan; // If non-zero, sleeping on chan
2113   int killed; // If non-zero, have been killed
2114   struct file *ofile[NOFILE]; // Open files
2115   struct inode *cwd; // Current directory
2116   char name[16]; // Process name (debugging)
2117 };
```



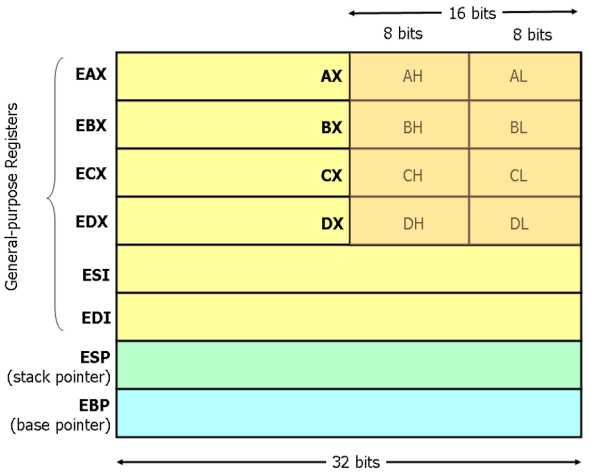
What about general registers?



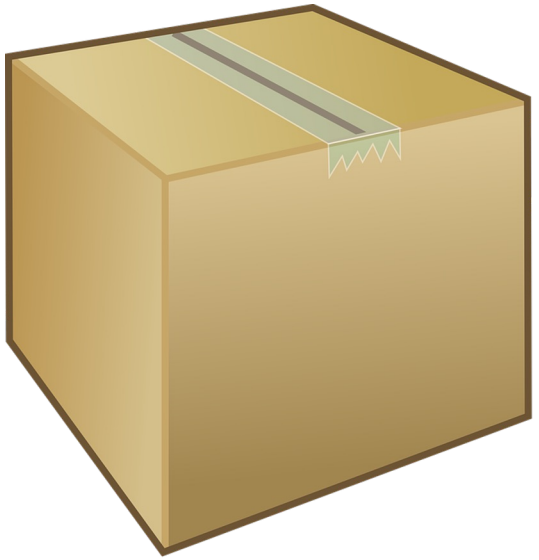


We need to save callee saved registers

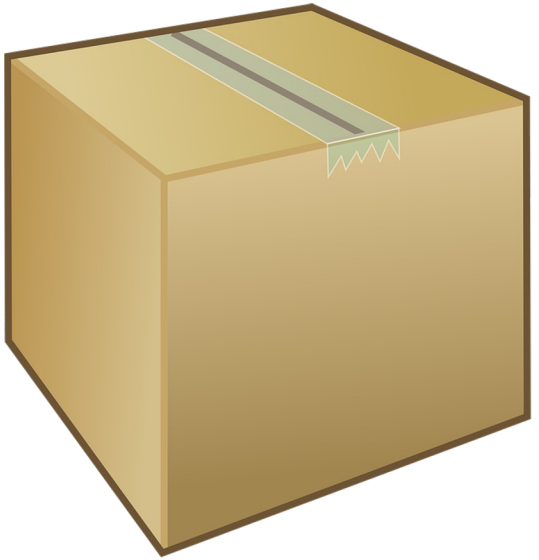
```
2093 struct context {
2094   uint edi;
2095   uint esi;
2096   uint ebx;
2097   uint ebp;
2098   uint eip;
2099 };
```



- Caller saved registers are already saved by the caller



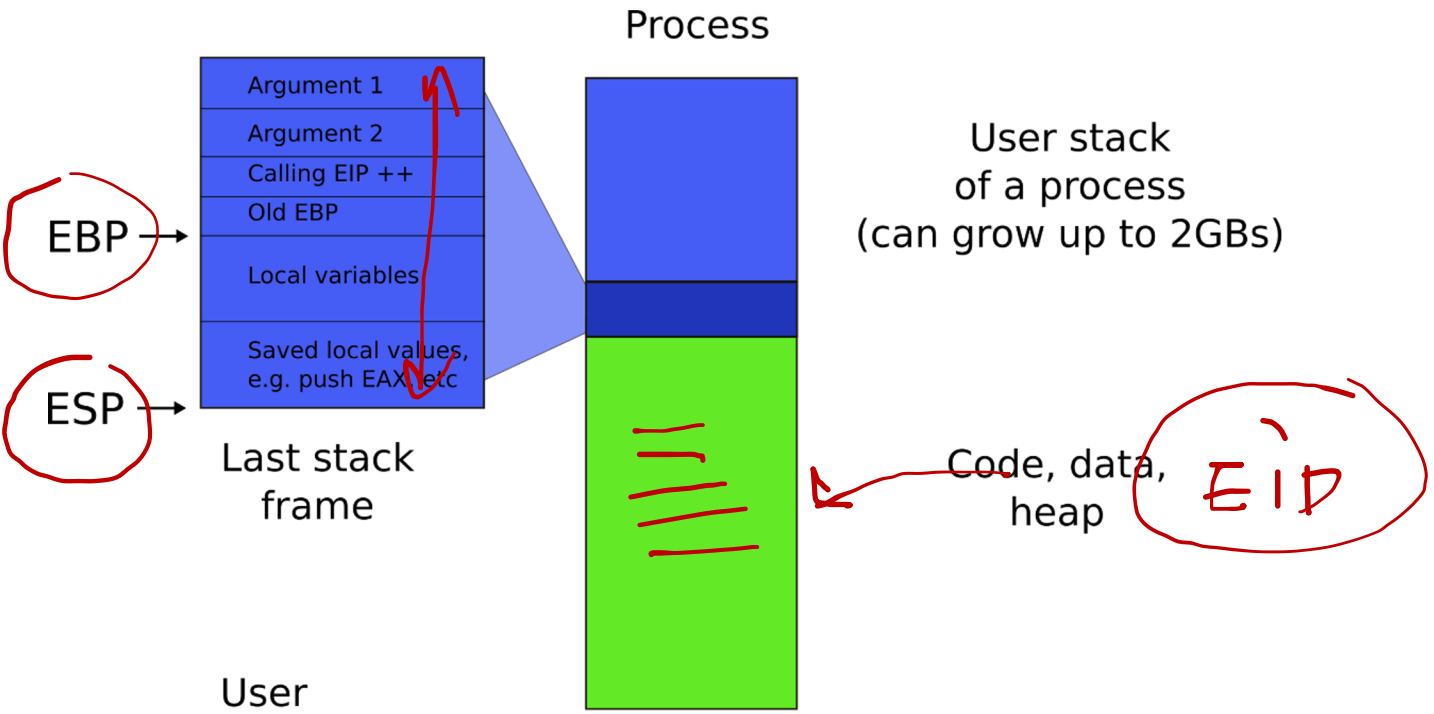
Process 1

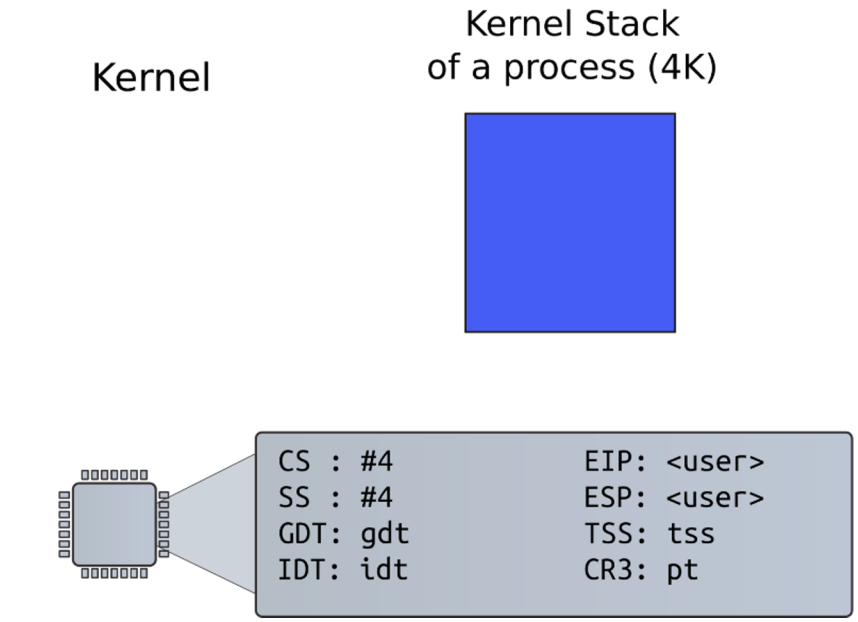


Process 2

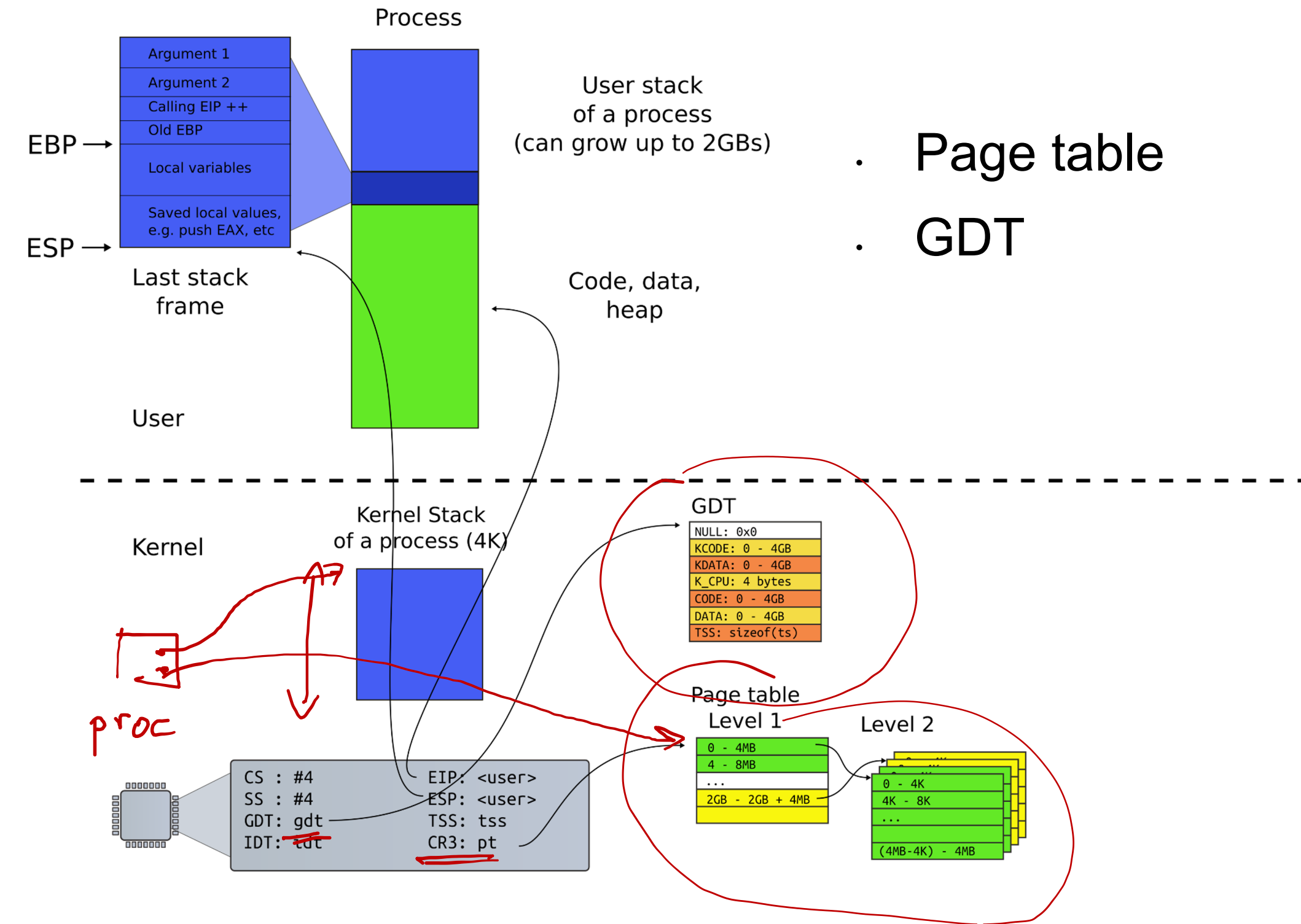
Back to the timer interrupt path

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



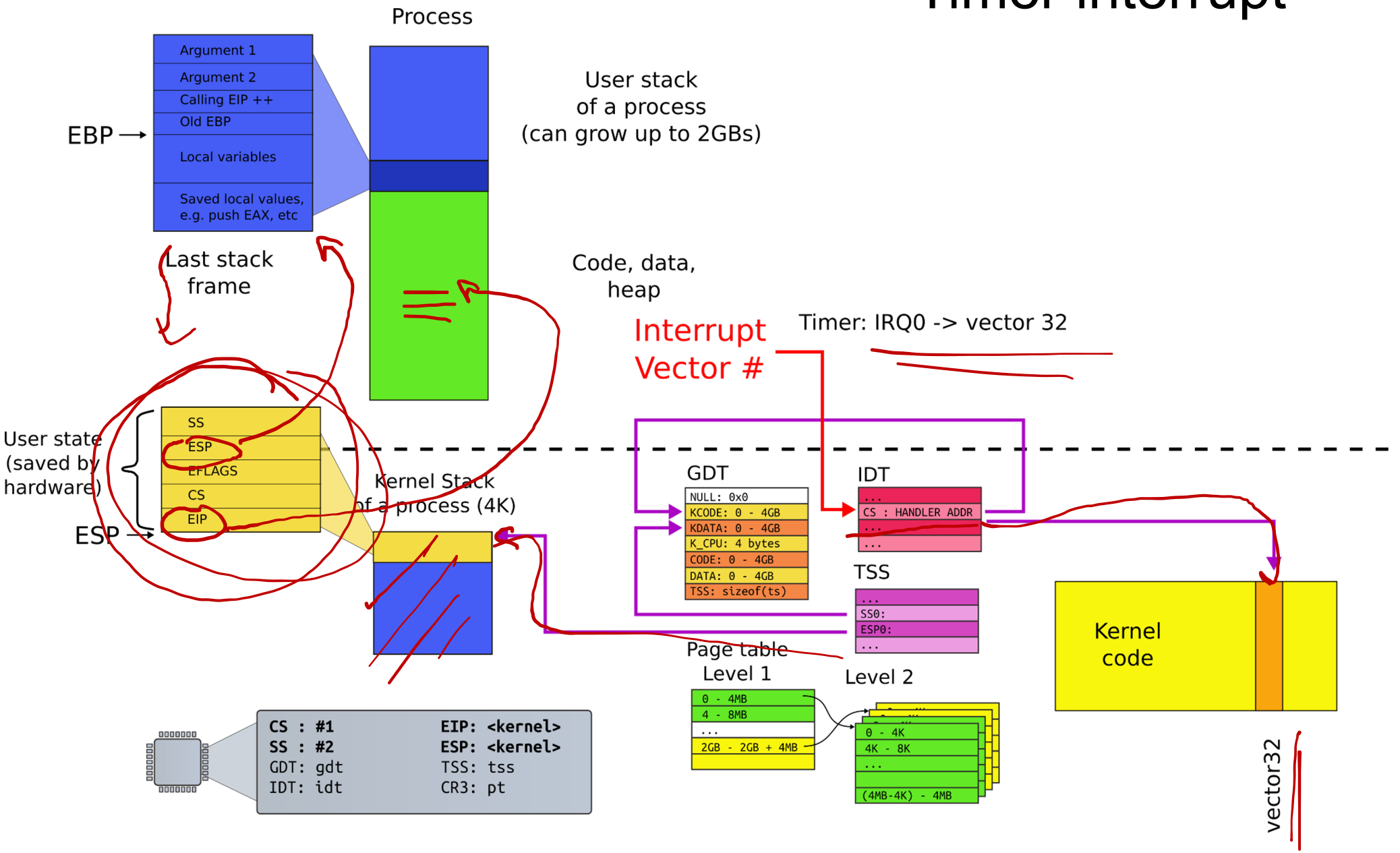


- User mode
- Two stacks
  - Kernel and user
  - Kernel stack is empty



- Page table
- GDT

Timer interrupt

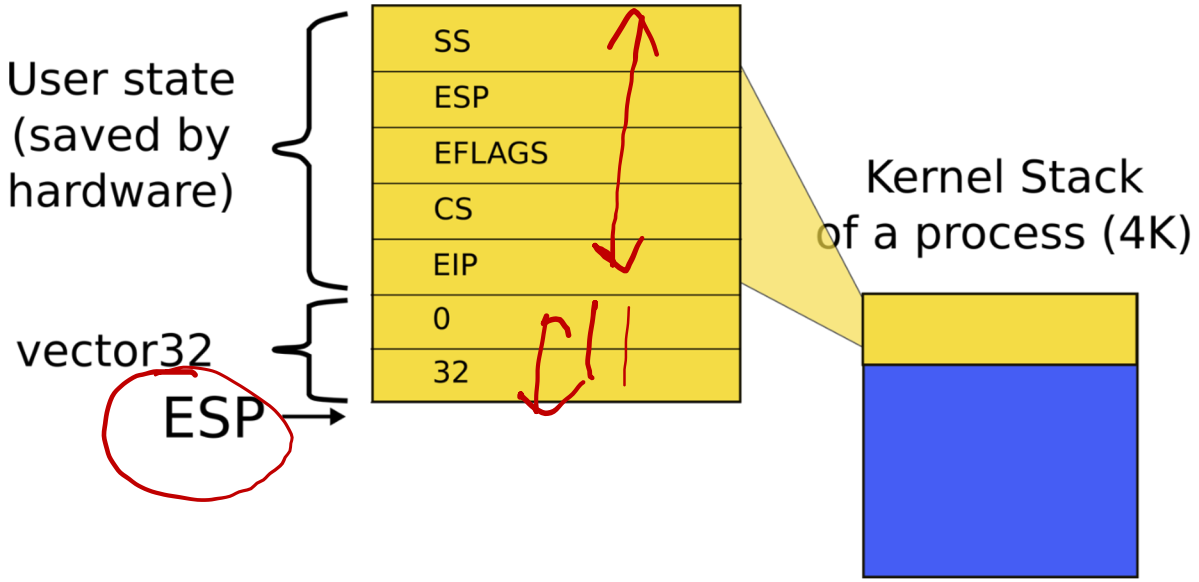


Where does IDT (entry 32) point to?

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

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

Kernel stack after interrupt

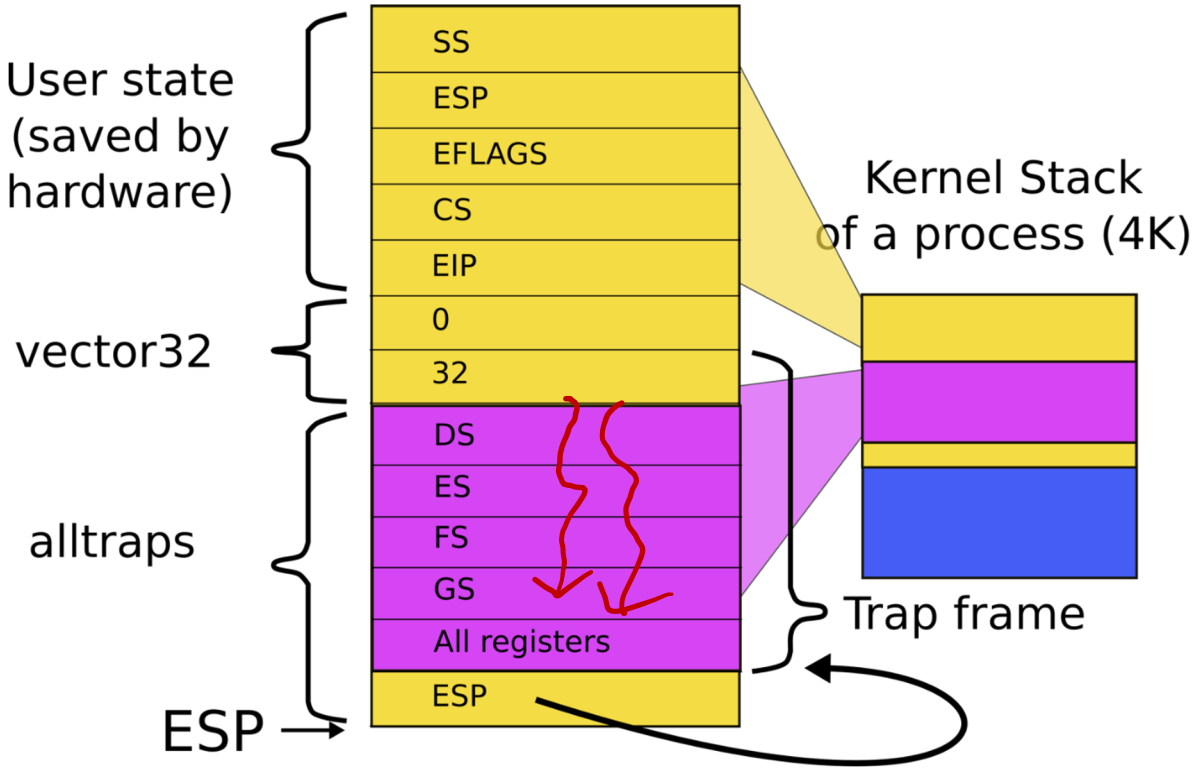


Call stack: vector32()

alltraps()

```
3254 alltraps:
3255   # Build trap frame.
3256   pushl %ds
3257   pushl %es
3258   pushl %fs
3259   pushl %gs
3260   pushal
3261
3262   # Set up data and per-cpu segments.
3263   movw $(SEG_KDATA<<3), %ax
3264   movw %ax, %ds
3265   movw %ax, %es
3266   movw $(SEG_KCPU<<3), %ax
3267   movw %ax, %fs
3268   movw %ax, %gs
3269
3270   # Call trap(tf), where tf=%esp
3271   pushl %esp
3272   call trap
```

Kernel stack after interrupt



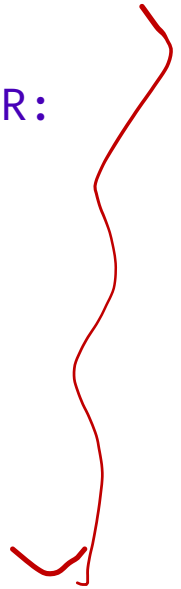
Call stack: vector32()
alltraps()

alltraps()

```
3254 alltraps:
3255 # Build trap frame.
3256 pushl %ds
3257 pushl %es
3258 pushl %fs
3259 pushl %gs
3260 pushal
3261
3262 # Set up data and per-cpu segments.
3263 movw $(SEG_KDATA<<3), %ax
3264 movw %ax, %ds
3265 movw %ax, %es
3266 movw $(SEG_KCPU<<3), %ax
3267 movw %ax, %fs
3268 movw %ax, %gs
3269
3270 # Call trap(tf), where tf=%esp
3271 pushl %esp
3272 call trap
```

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();
```



trap()

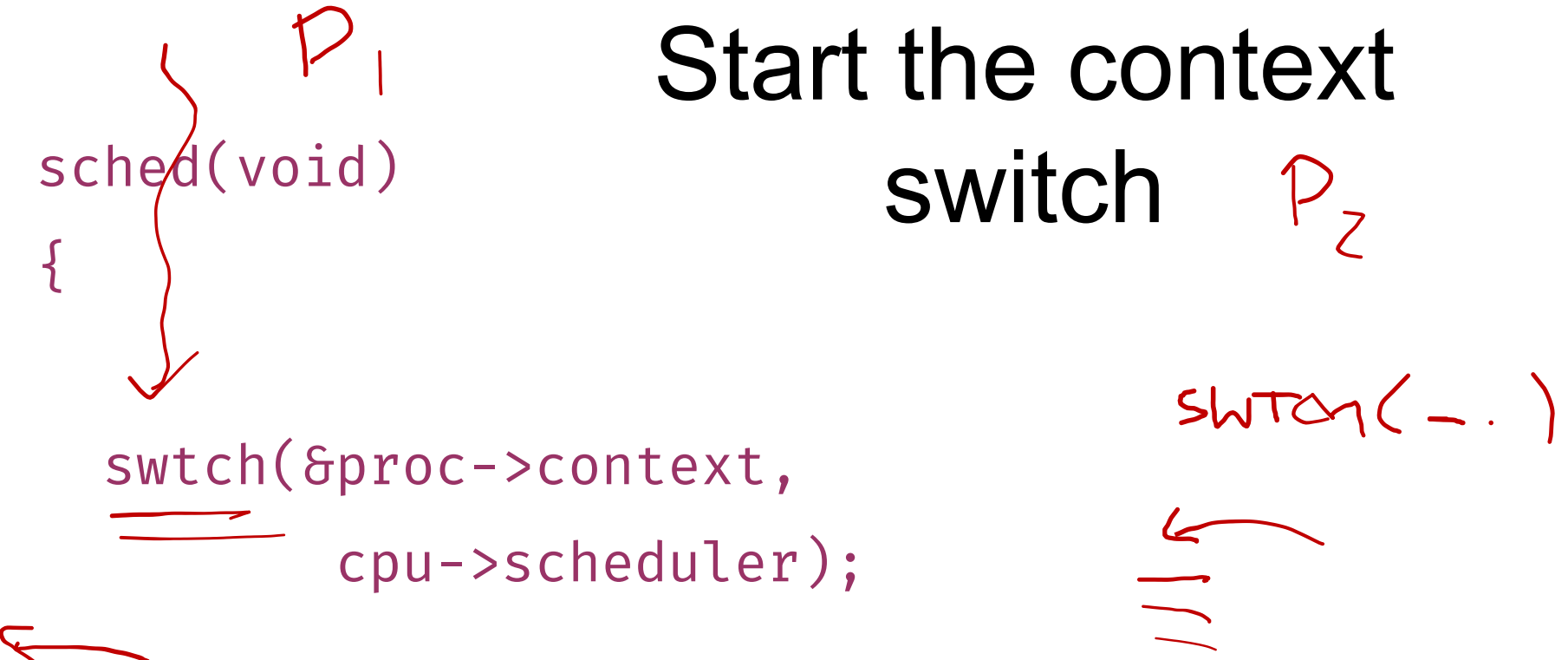
```
3351 trap(struct trapframe *tf)
3352 {
...
3363   switch(tf->trapno){
3364   case T_IRQ0 + IRQ_TIMER:
3365     if(cpu->id == 0){
3366     acquire(&tickslock);
3367     ticks++;
3368     wakeup(&ticks);
3369     release(&tickslock);
3370   }
3372   break;
...
3423   if(proc && proc->state == RUNNING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

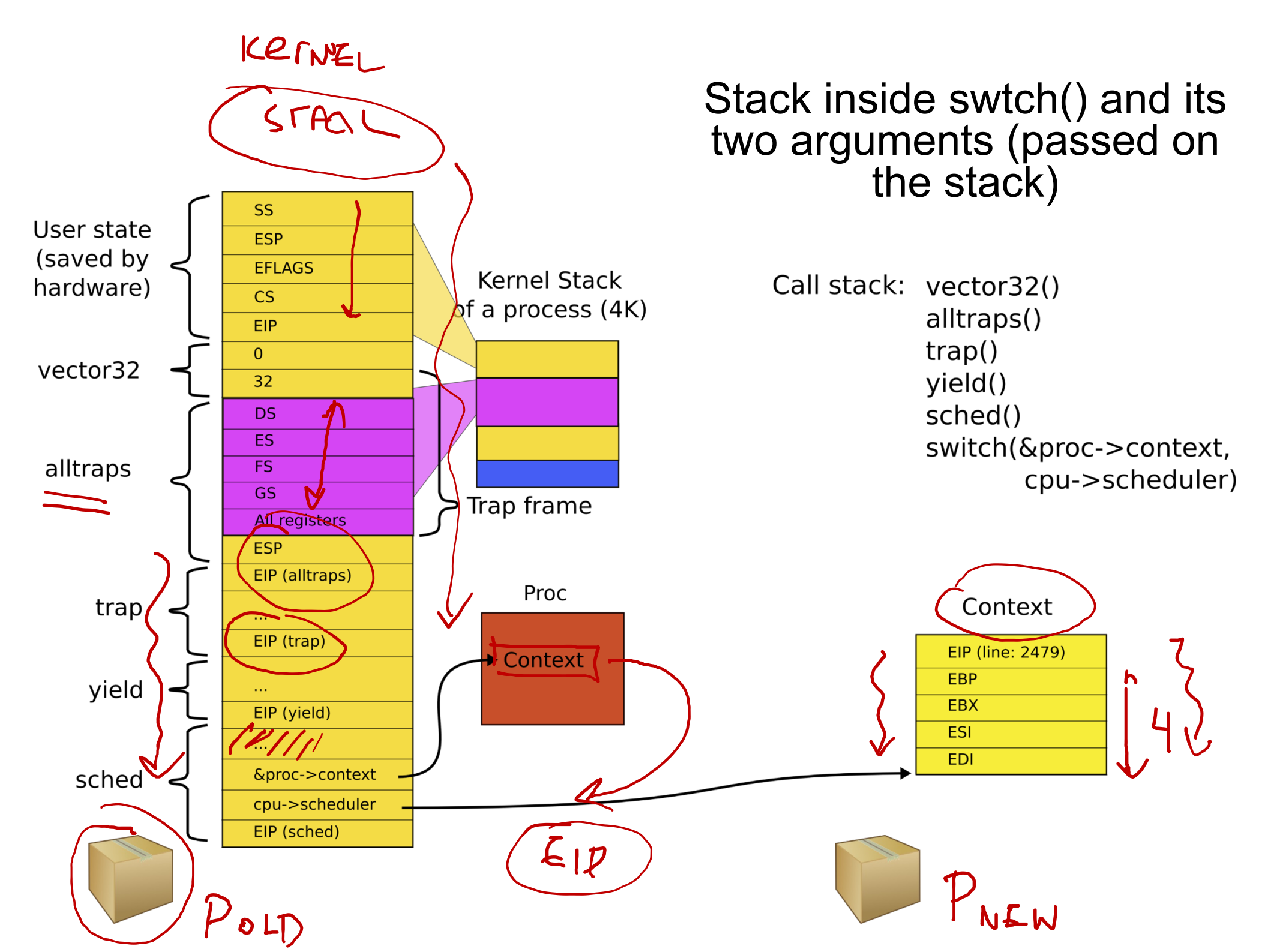
Invoke the scheduler

```
2777 yield(void)
2778 {
2779   acquire(&ptable.lock);
2780   proc->state = RUNNABLE;
2781   sched();
2782   release(&ptable.lock);
2783 }
```

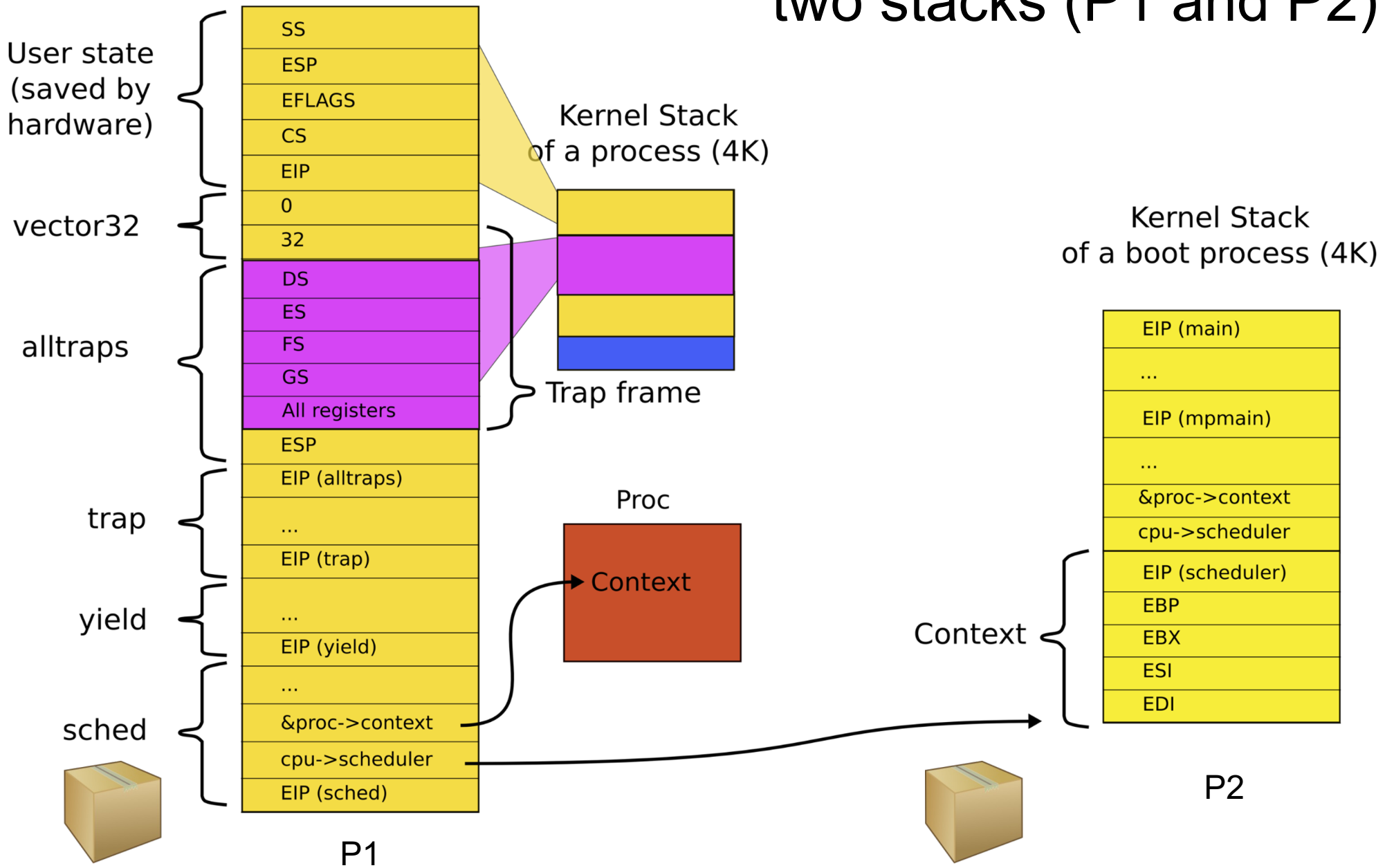
Start the context switch

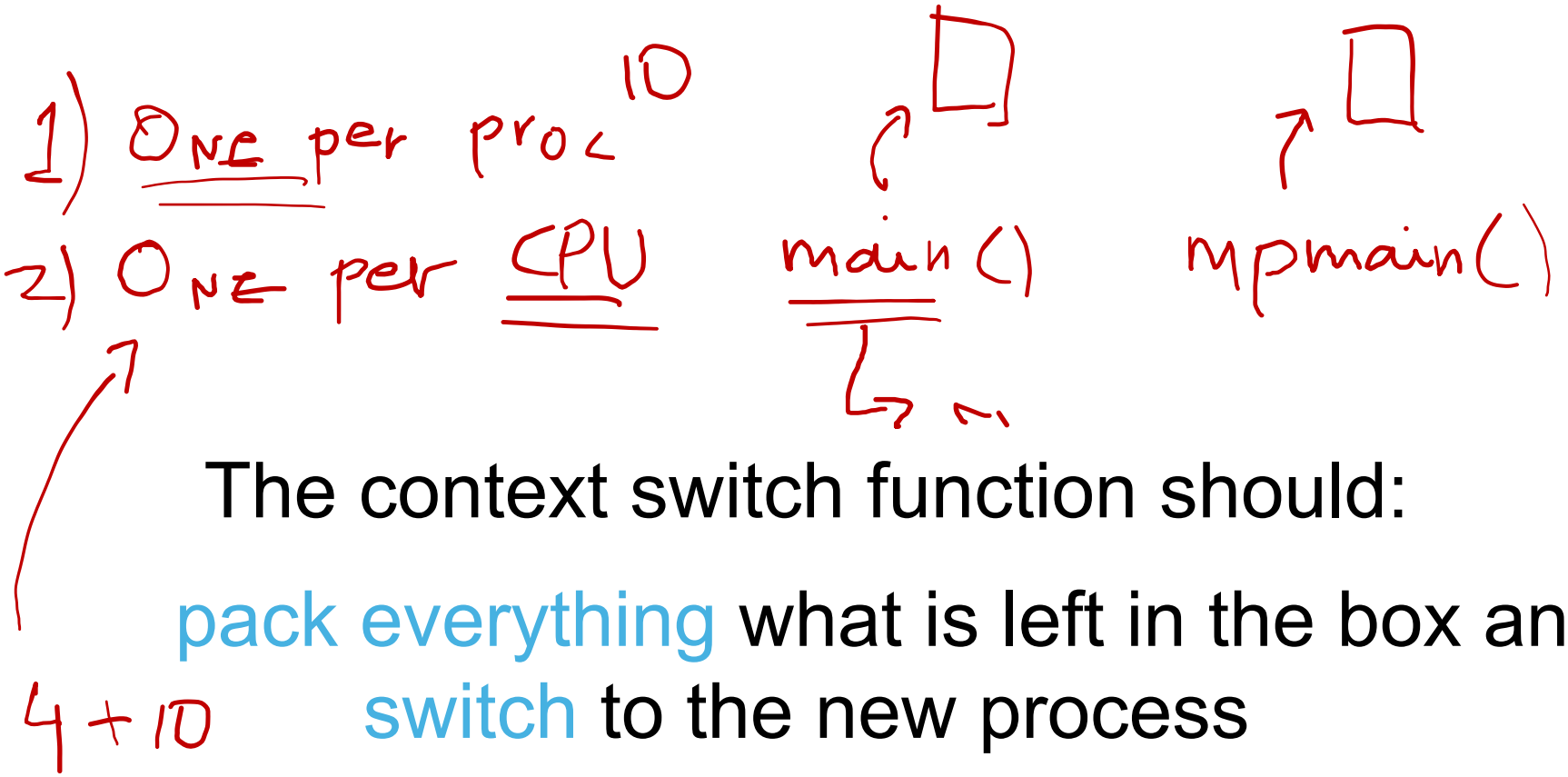
```
2758 sched(void)
2759 {
...
2771   swtch(&proc->context,
                cpu->scheduler);
...
2773 }
```




KERNEL
STACK
Stack inside swtch() and its two arguments (passed on the stack)
User state (saved by hardware)
vector32
alltraps
trap
yield
sched
SS
ESP
EFLAGS
CS
EIP
0
32
DS
ES
FS
GS
All registers
ESP
EIP (alltraps)
...
EIP (trap)
...
EIP (yield)
...
&proc->context
cpu->scheduler
EIP (sched)
Kernel Stack of a process (4K)
Trap frame
Proc
Context
Call stack:
vector32()
alltraps()
trap()
yield()
sched()
switch(&proc->context,
cpu->scheduler)
Context
EIP (line: 2479)
EBP
EBX
ESI
EDI
4
EIP
P OLD
P NEW

Remember you have two stacks (P1 and P2)

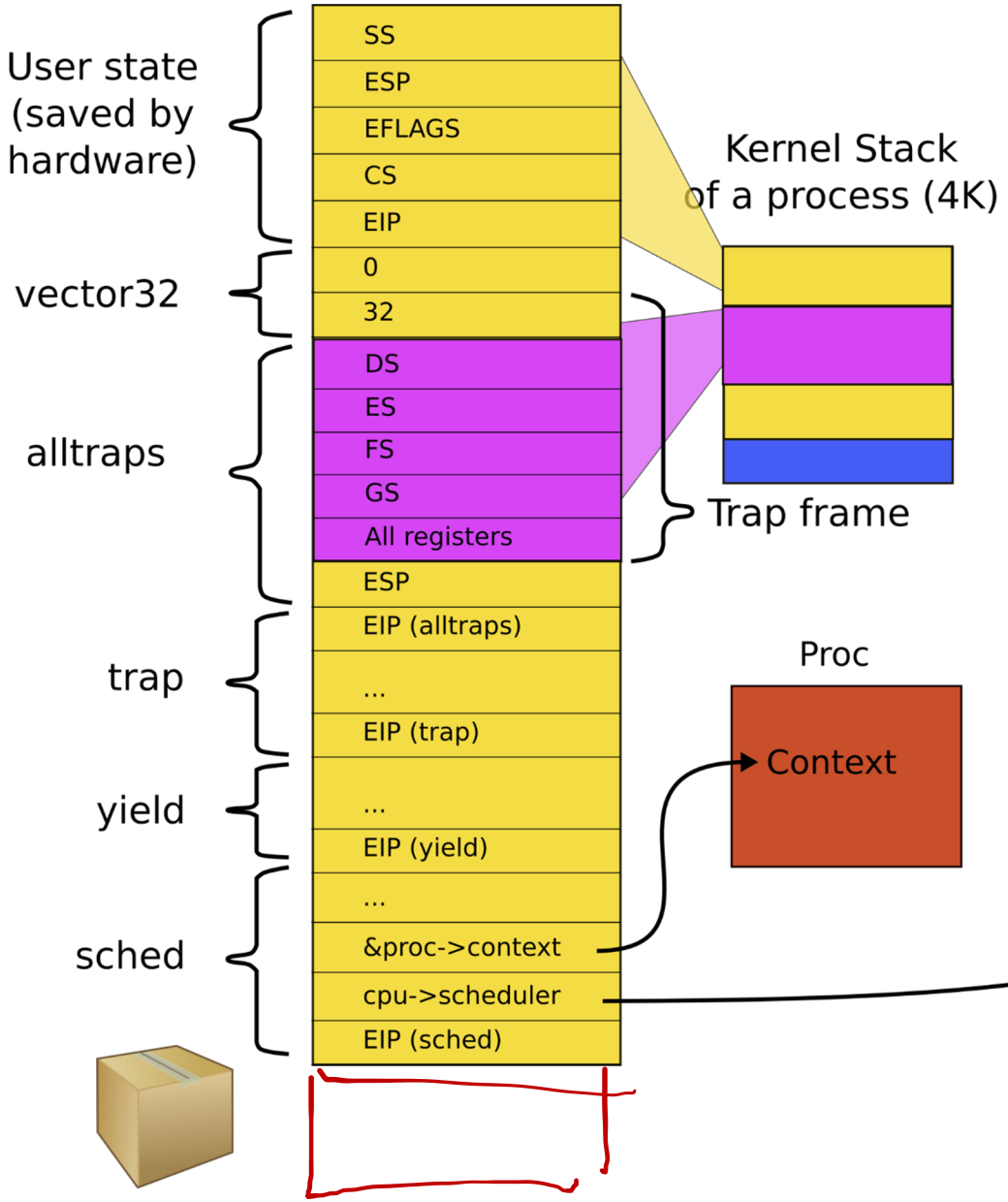




The context switch function should:

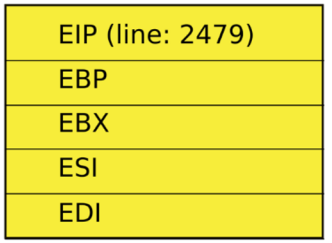
pack everything what is left in the box and switch to the new process

Callee saved registers



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

Context



swtch(): save registers on the stack

```
2958 swtch:
2959 movl 4(%esp), %eax
2960 movl 8(%esp), %edx
2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi
2967
2968 # Switch stacks
2969 movl %esp, (%eax)
2970 movl %edx, %esp
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

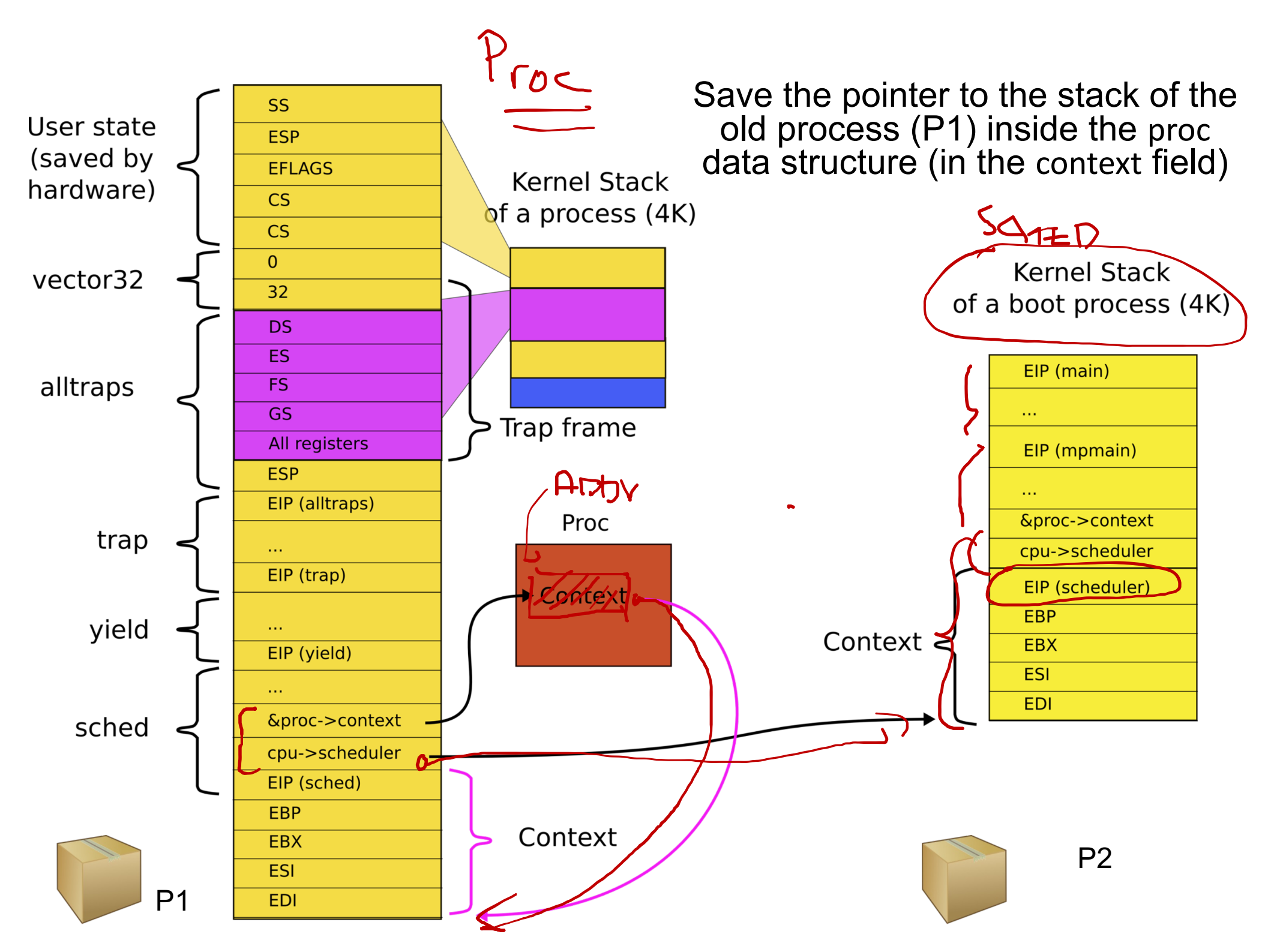
mov

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

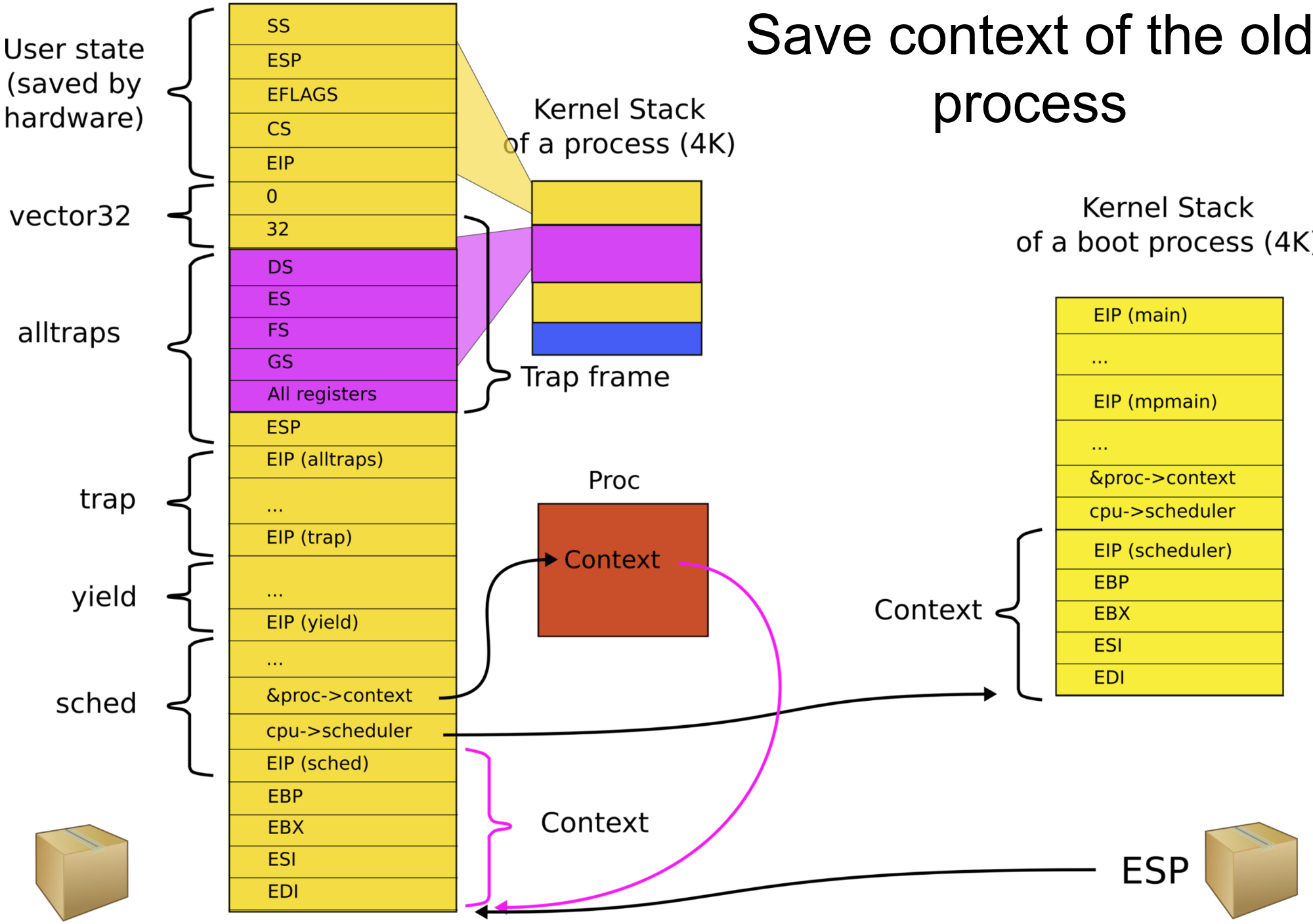
CALL SWTCH

A 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)
User state (saved by hardware)
vector32
alltraps
trap
yield
sched
SS
ESP
EFLAGS
CS
CS
0
32
DS
ES
FS
GS
All registers
ESP
EIP (alltraps)
...
EIP (trap)
...
EIP (yield)
...
&proc->context
cpu->scheduler
EIP (sched)
EBP
EBX
ESI
EDI
Kernel Stack of a process (4K)
Trap frame
Proc
Context
Context
P1
Kernel Stack of a boot process (4K)
EIP (main)
...
EIP (mpmain)
...
&proc->context
cpu->scheduler
EIP (scheduler)
EBP
EBX
ESI
EDI
P2

Save context of the old process



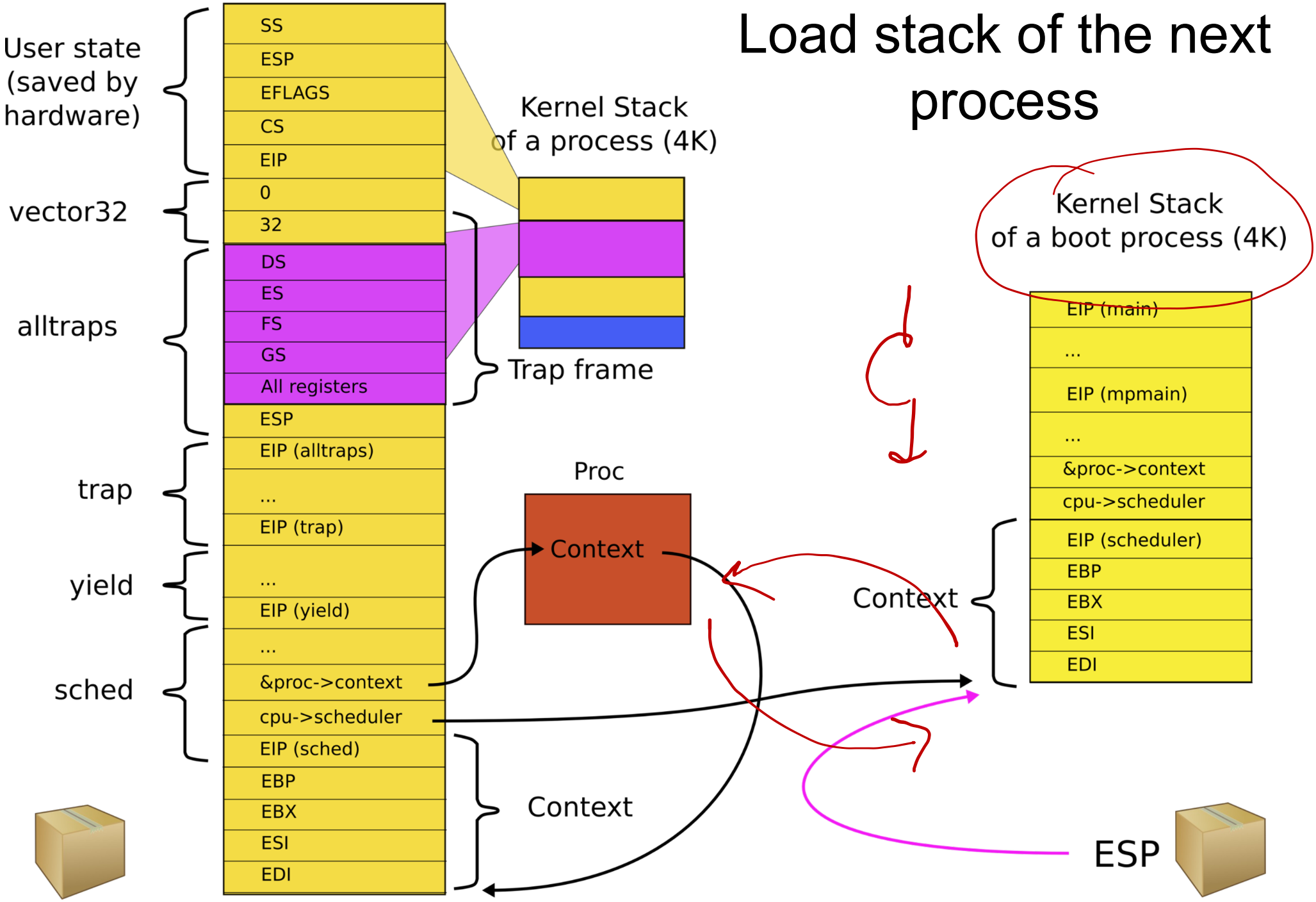
Save the current stack

```
2958 swtch:
2959 movl 4(%esp), %eax    // struct context **old
2960 movl 8(%esp), %edx    // struct context *new
2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi
2967
2968 # Switch stacks
2969 movl %esp, (%eax)  // save current context (top of current stack)
2970 movl %edx, %esp    // set stack to be equal to the new
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

Switch to the new stack

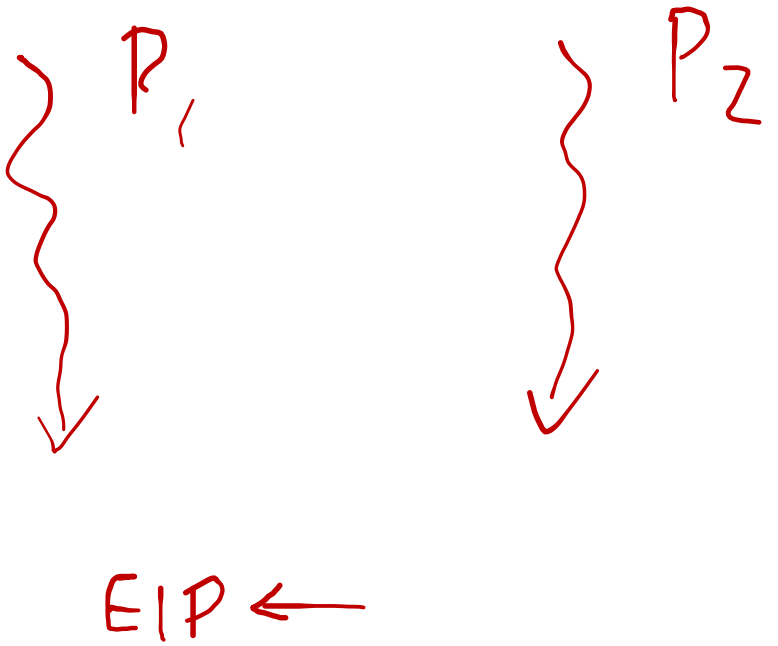
```
2958 swtch:
2959 movl 4(%esp), %eax    // struct context **old
2960 movl 8(%esp), %edx    // struct context *new
2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi
2967
2968 # Switch stacks
2969 movl %esp, (%eax)  // save current context (top of current stack)
2970 movl %edx, %esp    // set stack to be equal to the new
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

Load stack of the next process



```
2958 swtch:
2959 movl 4(%esp), %eax
2960 movl 8(%esp), %edx
2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi
2967
2968 # Switch stacks
2969 movl %esp, (%eax)
2970 movl %edx, %esp
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

Restore the context of the next process

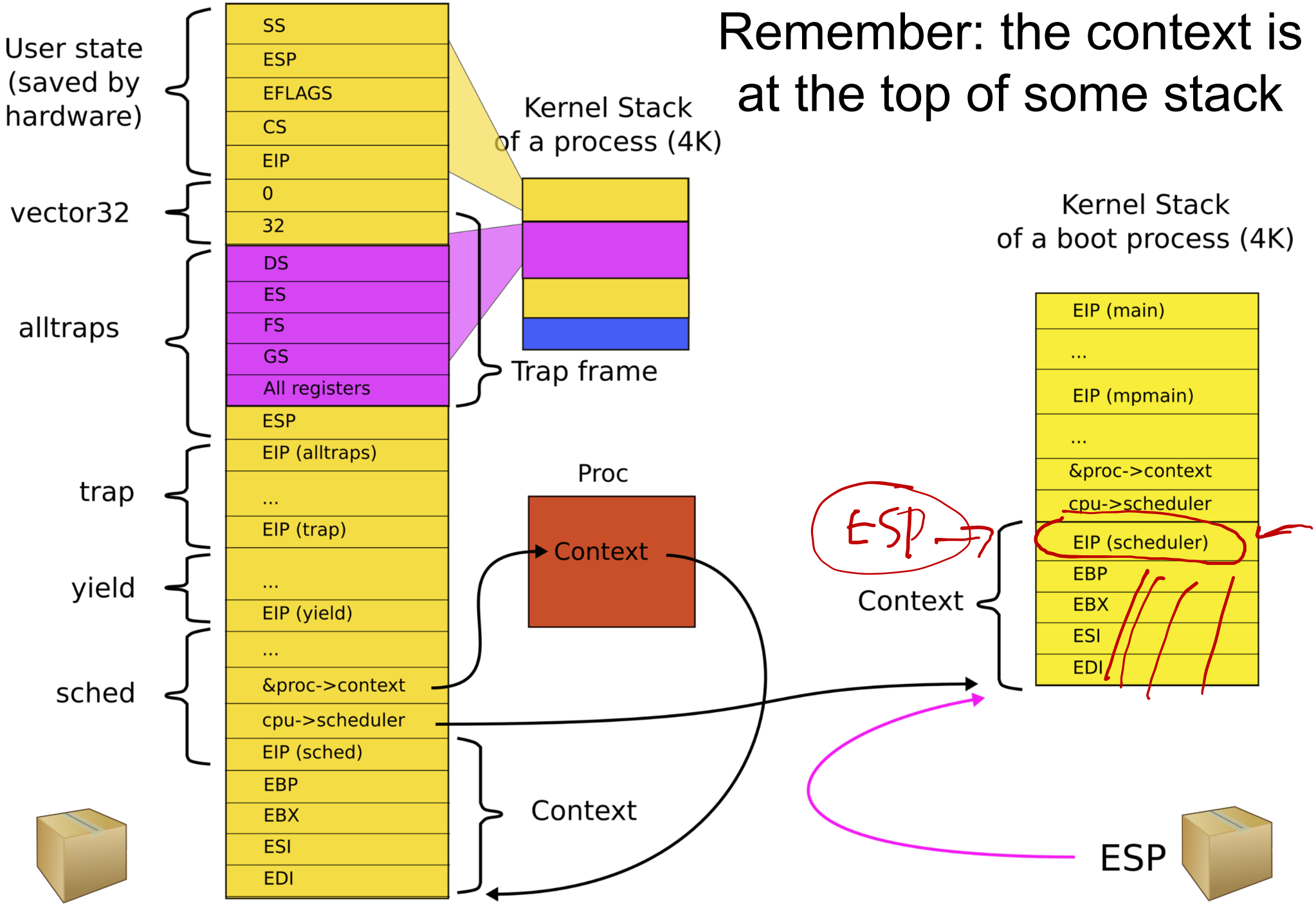


```
2958 swtch:
2959 movl 4(%esp), %eax
2960 movl 8(%esp), %edx
2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi
2967
2968 # Switch stacks
2969 movl %esp, (%eax)
2970 movl %edx, %esp
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

But how do we switch the process?

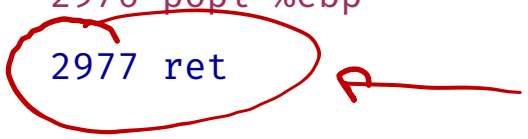
- I.e., switch the instruction pointer?

Remember: the context is at the top of some stack



If we load the new context into ESP we can return on that new stack

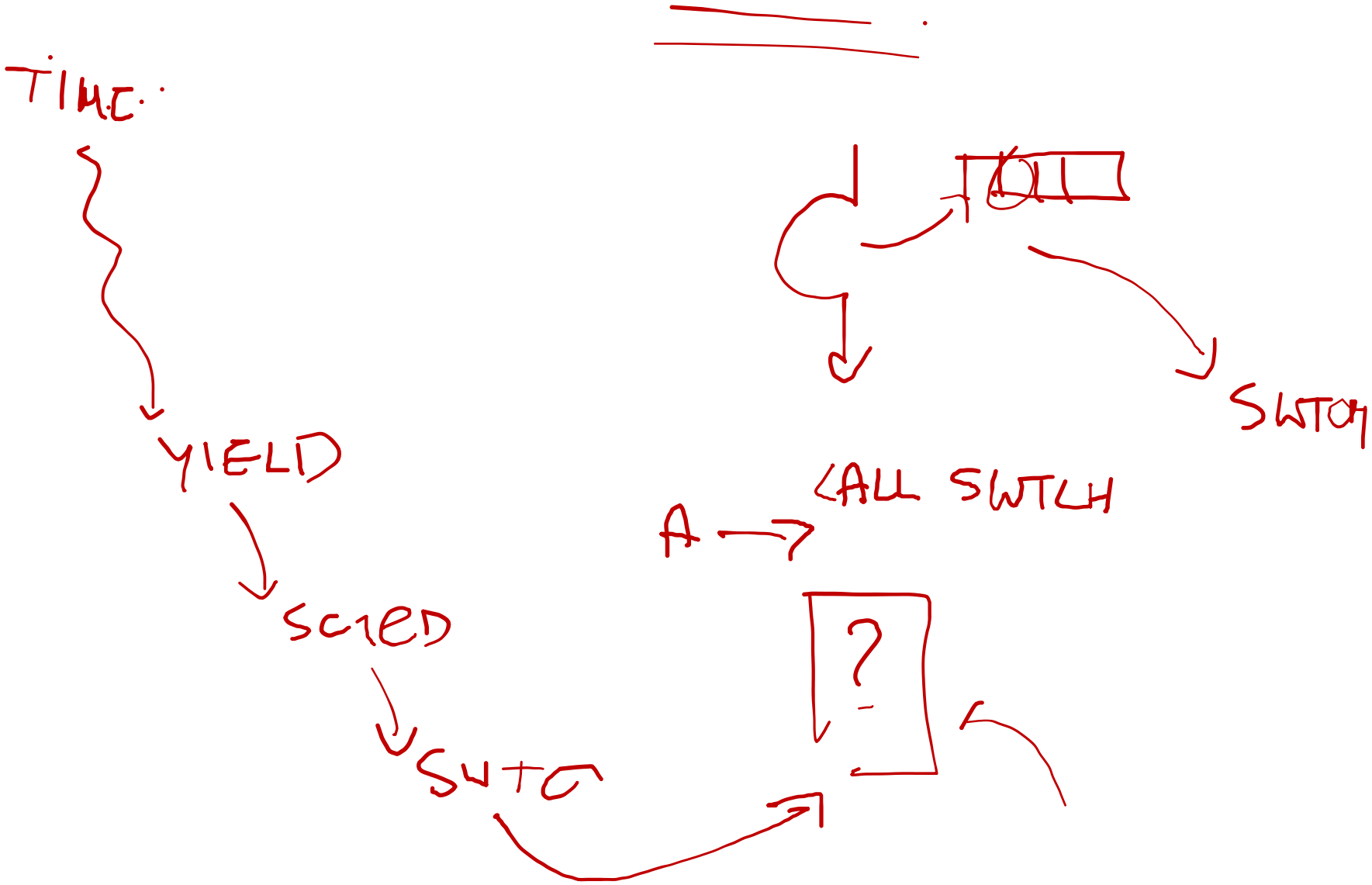
```
2958 swtch:
2959 movl 4(%esp), %eax
2960 movl 8(%esp), %edx
2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi
2967
2968 # Switch stacks
2969 movl %esp, (%eax)
2970 movl %edx, %esp
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```



But how do we switch the process?

- I.e., switch the instruction pointer?

Where does this swtch() return?



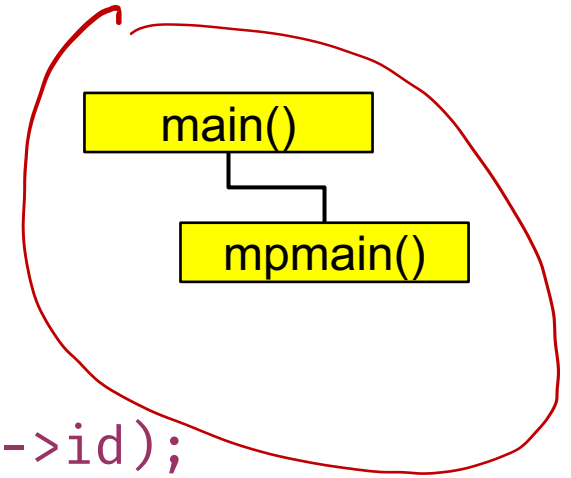
Context is always top of some stack...ok, but how?

- How does initialization of each CPU end?

```
1317 main(void)
1318 {
1319   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320   kvmalloc(); // kernel page table
1321   mpinit(); // detect other processors
...
1323   seginit(); // segment descriptors
...
1330   tvinit(); // trap vectors
...
1338   userinit(); // first user process
1339   mpmain(); // finish this processor's setup
1340 }
```

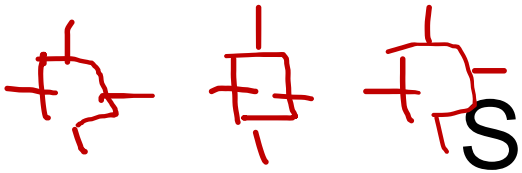
main()

```
1260 // Common CPU setup code.
1261 static void
1262 mpmain(void)
1263 {
1264   cprintf("cpu%d: starting\n", cpu->id);
1265   idtinit(); // load idt register
1266   xchg(&cpu->started, 1);
1267   scheduler(); // start running processes
1268 }
```



We ended boot by starting the scheduler

Scheduler()



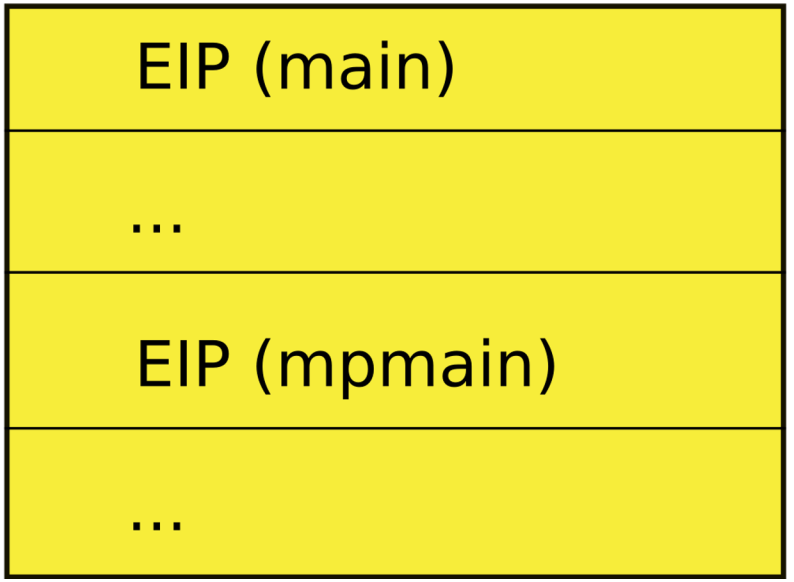
```
2458 scheduler(void)
2459 {
2462   for(;;){
2468     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2469       if(p->state != RUNNABLE)
2470         continue;
2475       proc = p;
2476       switchuvm(p);
2477       p->state = RUNNING;
2478       swtch(&cpu->scheduler, proc->context);
2479       switchkvm();
2483       proc = 0;
2484     }
2487   }
2488 }
```

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

```
2301 struct cpu {
2302   uchar apicid;                // Local APIC
2303   struct context *scheduler;   // swtch() here to enter scheduler
2304   struct taskstate ts;         // TSS
2305   struct segdesc gdt[NSEGS];   // x86 global descriptor table
2306   volatile uint started;       // Has the CPU started?
2307   int ncli;                    // Depth of pushcli nesting.
2308   int intena;                  // Were interrupts enabled ...
2309   struct proc *proc;           // The process running on this cpu
2310 };
2311
2312 extern struct cpu cpus[NCPU];
2313 extern int ncpu;
```

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

Kernel Stack
of a boot process (4K)



- When the scheduler context switched the first time

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

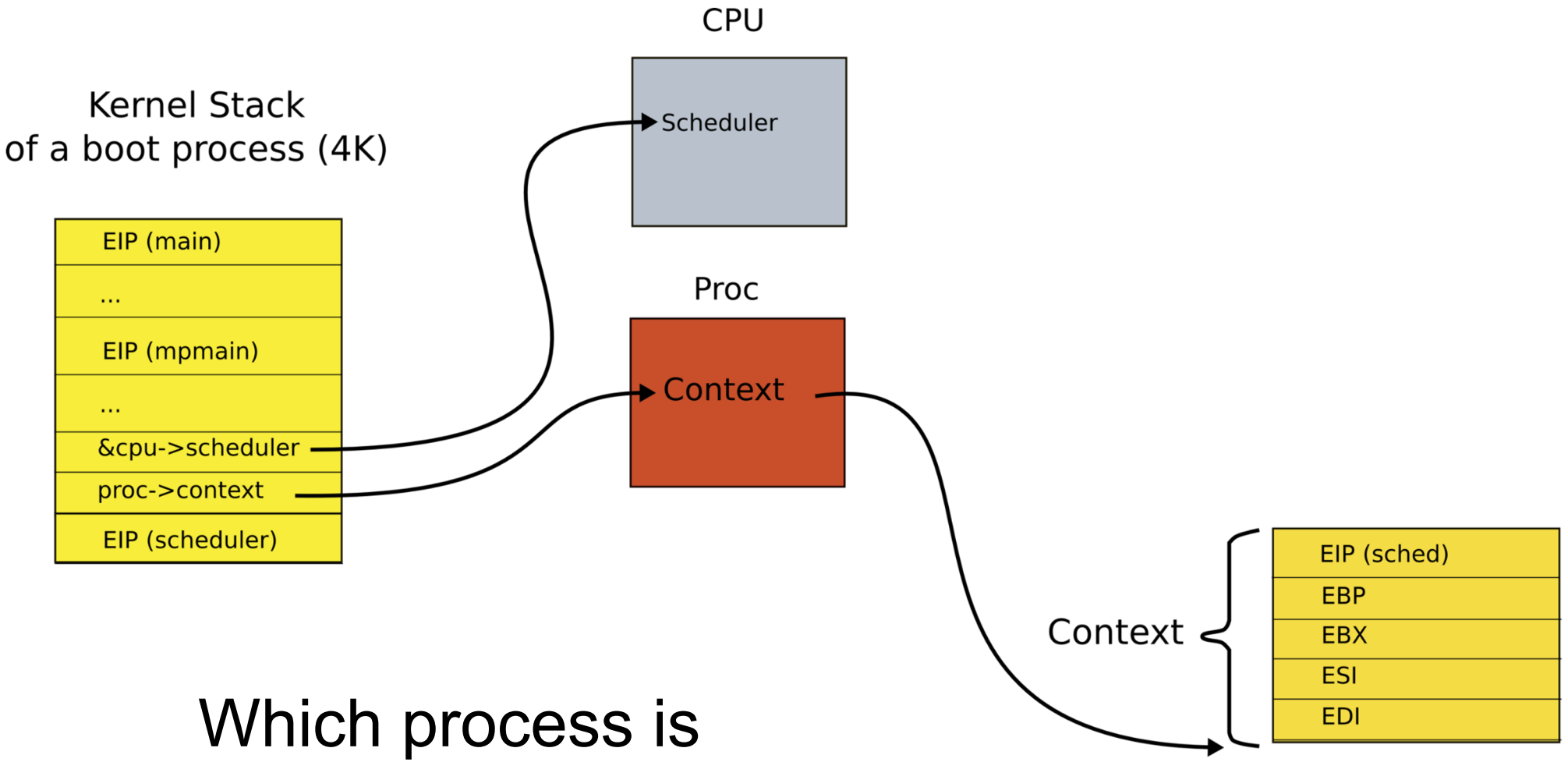
- We saved the current context of the scheduler into:

  `&cpu->scheduler`

- And restored 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 represented by this context?

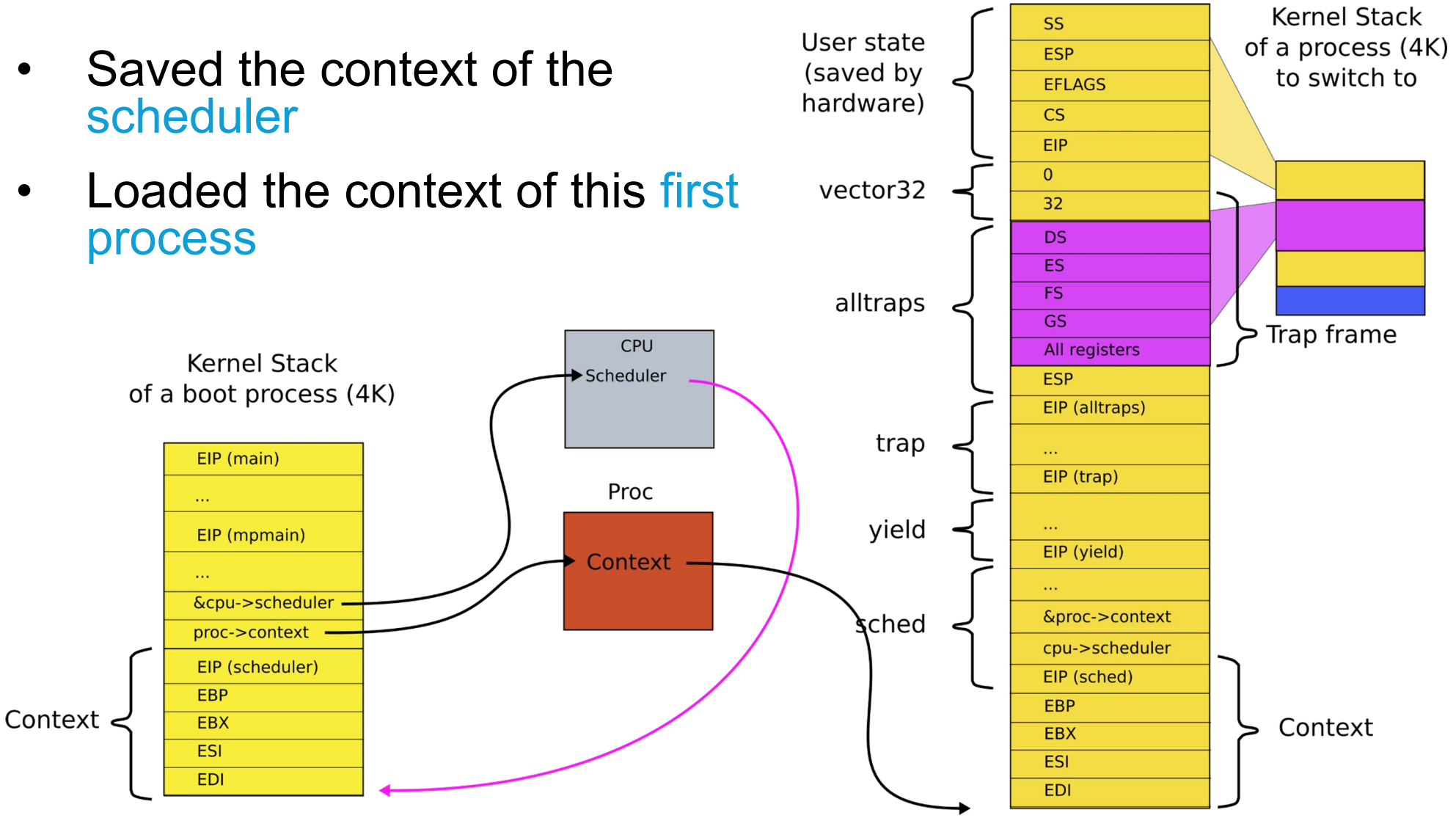
```
2456 allocproc(void)
2457 {
...
2477   // Leave room for trap frame.
2478   sp -= sizeof *p->tf;
2479   p->tf = (struct trapframe*)sp;
2480
2481   // Set up new context to start executing at forkret,
2482   // which returns to trapret.
2483   sp -= 4;
2484   *(uint*)sp = (uint)trapret;
2485
2486   sp -= sizeof *p->context;
2487   p->context = (struct context*)sp;
2488   memset(p->context, 0, sizeof *p->context);
2489   p->context->eip = (uint)forkret;
...
2492 }
```

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

- Remember fork()?

At the end of boot we switched to the first process

- Saved the context of the scheduler
- Loaded the context of this first process



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

(we now know we return into the scheduler)

Where does this swtch() return?

- Scheduler!
- Remember:
  - We started with timer interrupt
  - Entered the kernel
  - Entered schedule()
  - Entered switch

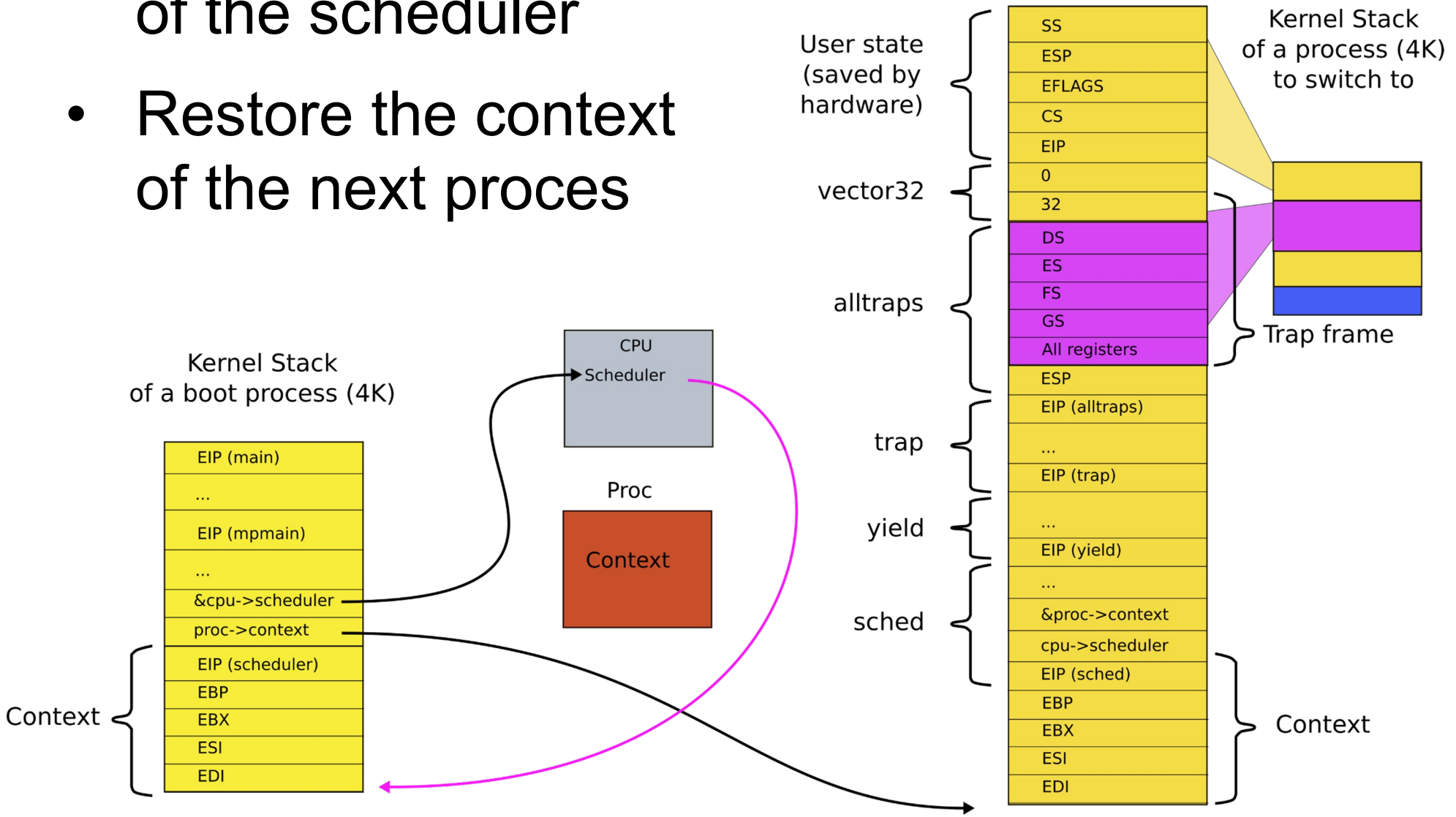
And are currently on our way from the process into the scheduler

What does scheduler do?

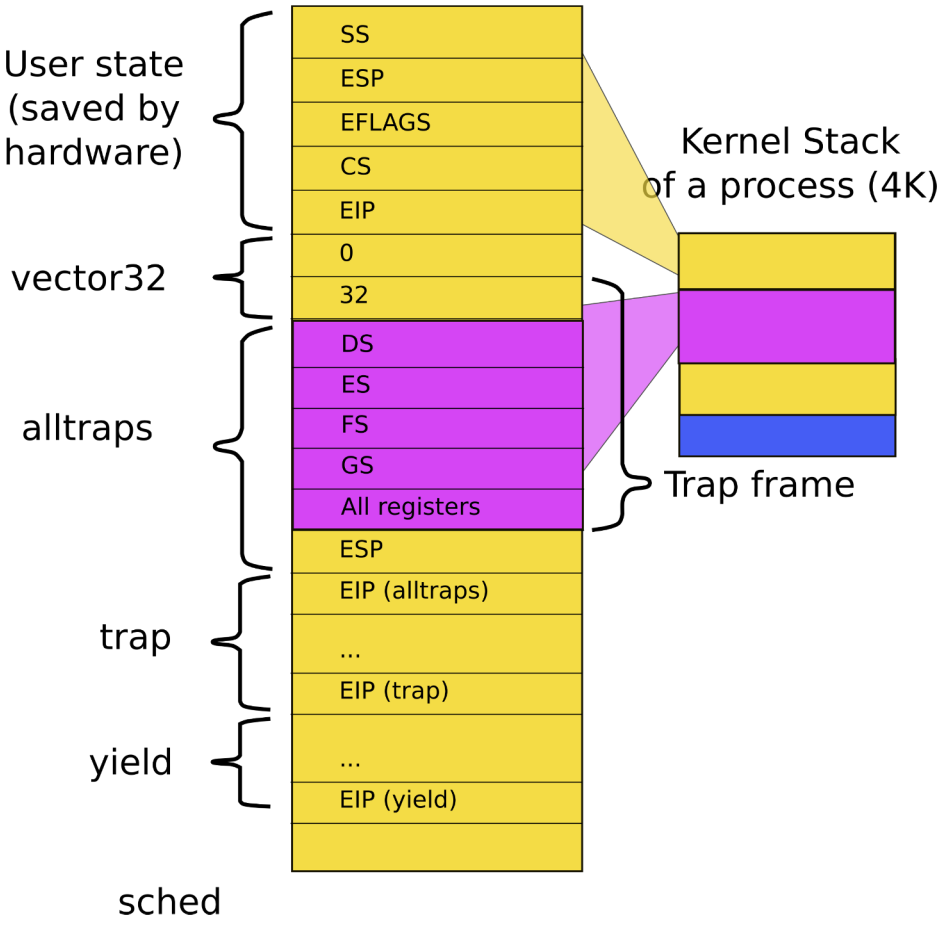
```
2458 scheduler(void)
2459 {
2462   for(;;){
2468     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2469       if(p->state != RUNNABLE)
2470         continue;
2475       proc = p;
2476       switchuvm(p);
2477       p->state = RUNNING;
2478       swtch(&cpu->scheduler, proc->context);
2479       switchkvm();
2483       proc = 0;
2484     }
2487   }
2488 }
```

- Chooses next process to run
- Switches to it

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

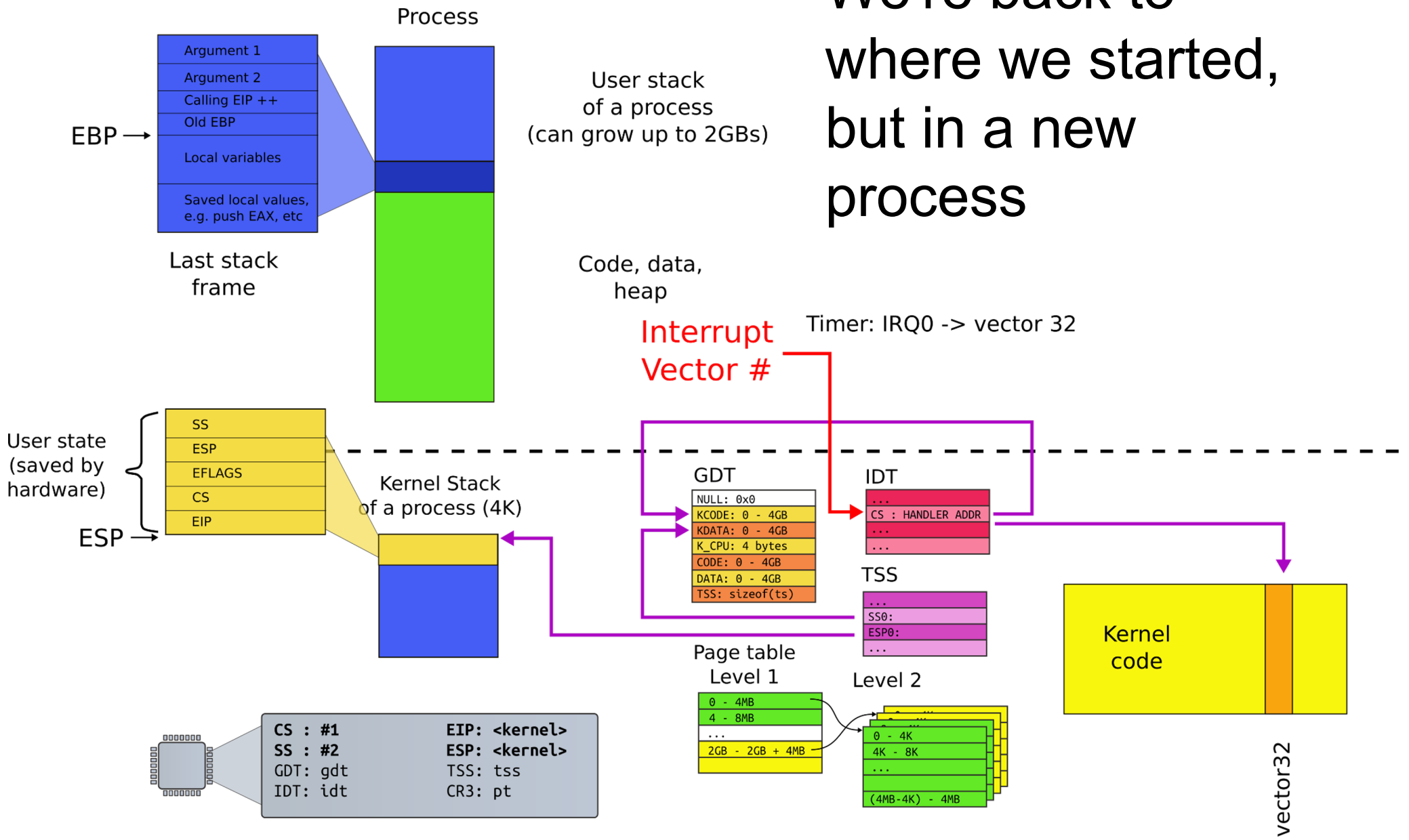


Exiting back to user-level



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

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



Summary

- We switch between processes now

Thank you