

238P: Operating Systems

Lecture 10: Context switch

Anton Burtsev
November, 2018

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

```
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();
```

trap()

Invoke the scheduler

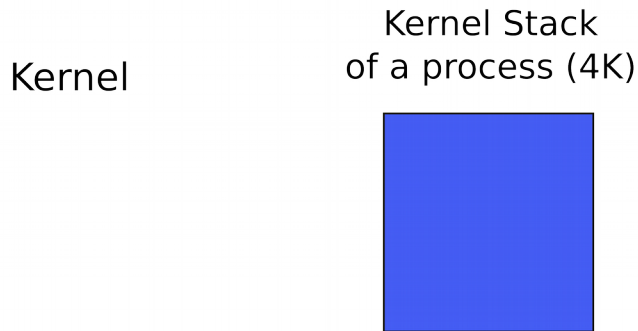
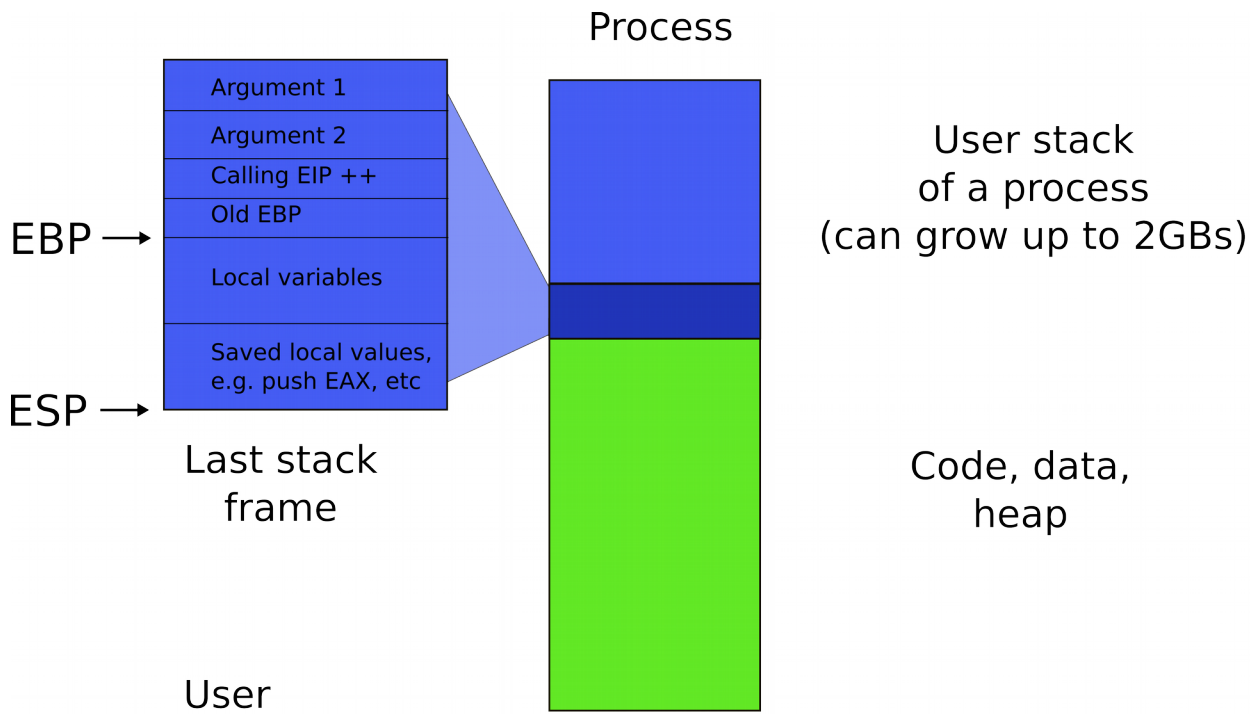
```
2777 yield(void)
2778 {
2779     acquire(&ptable.lock);
2780     proc->state = RUNNABLE;
2781     sched();
2782     release(&ptable.lock);
2783 }
```

Start the context switch

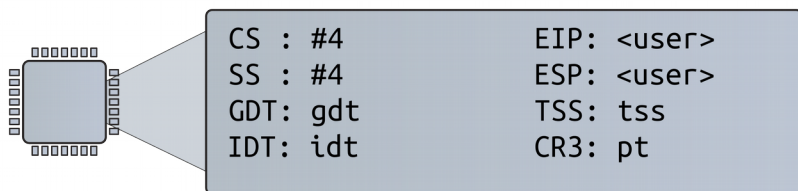
```
2758 sched(void)
2759 {
...
2771     swtch(&proc->context,
           cpu->scheduler);
...
2773 }
```

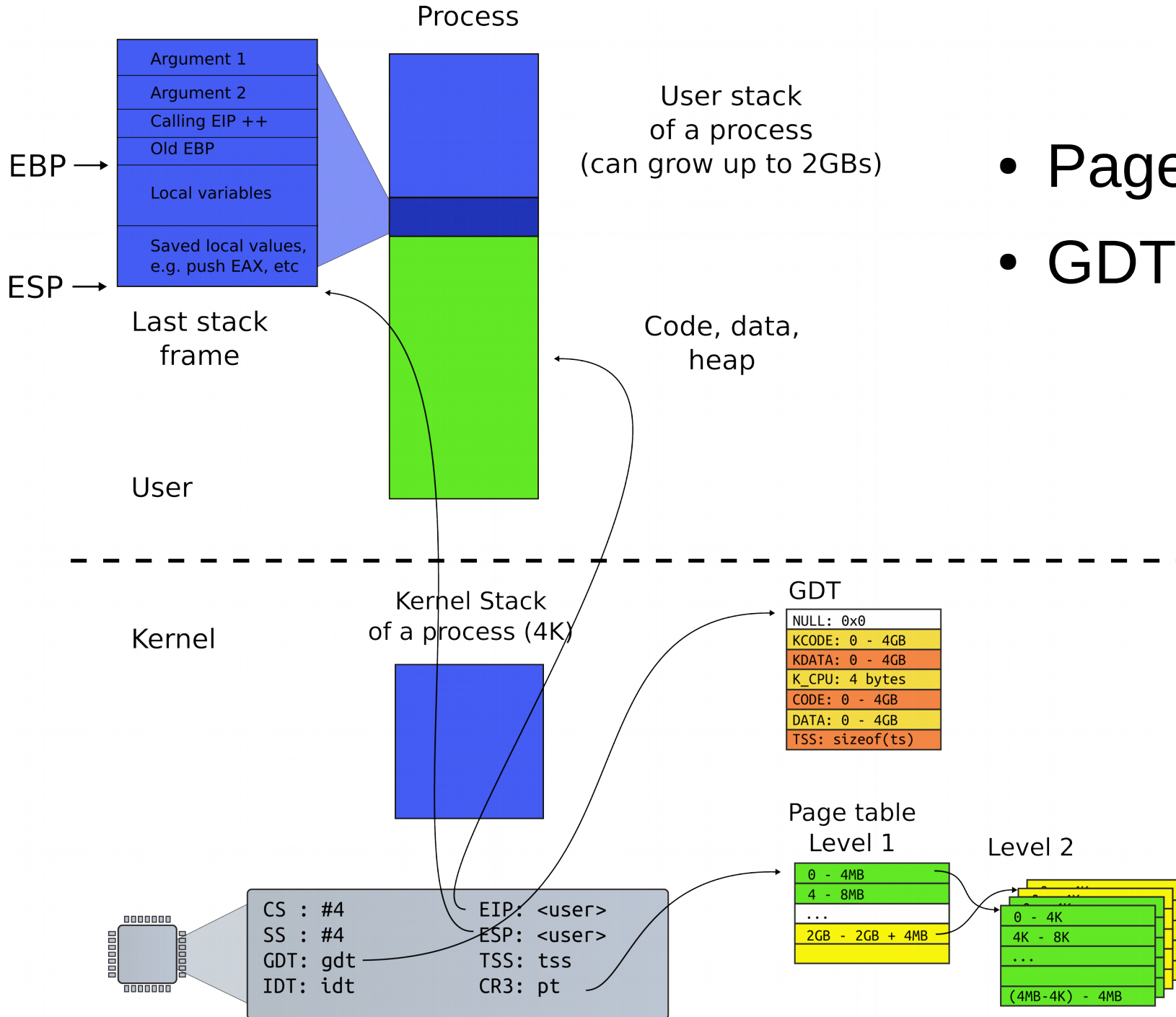

But what do you think needs to happen inside
`switch()`?

Back to the timer interrupt path
(keep track of what happens to the stack!)



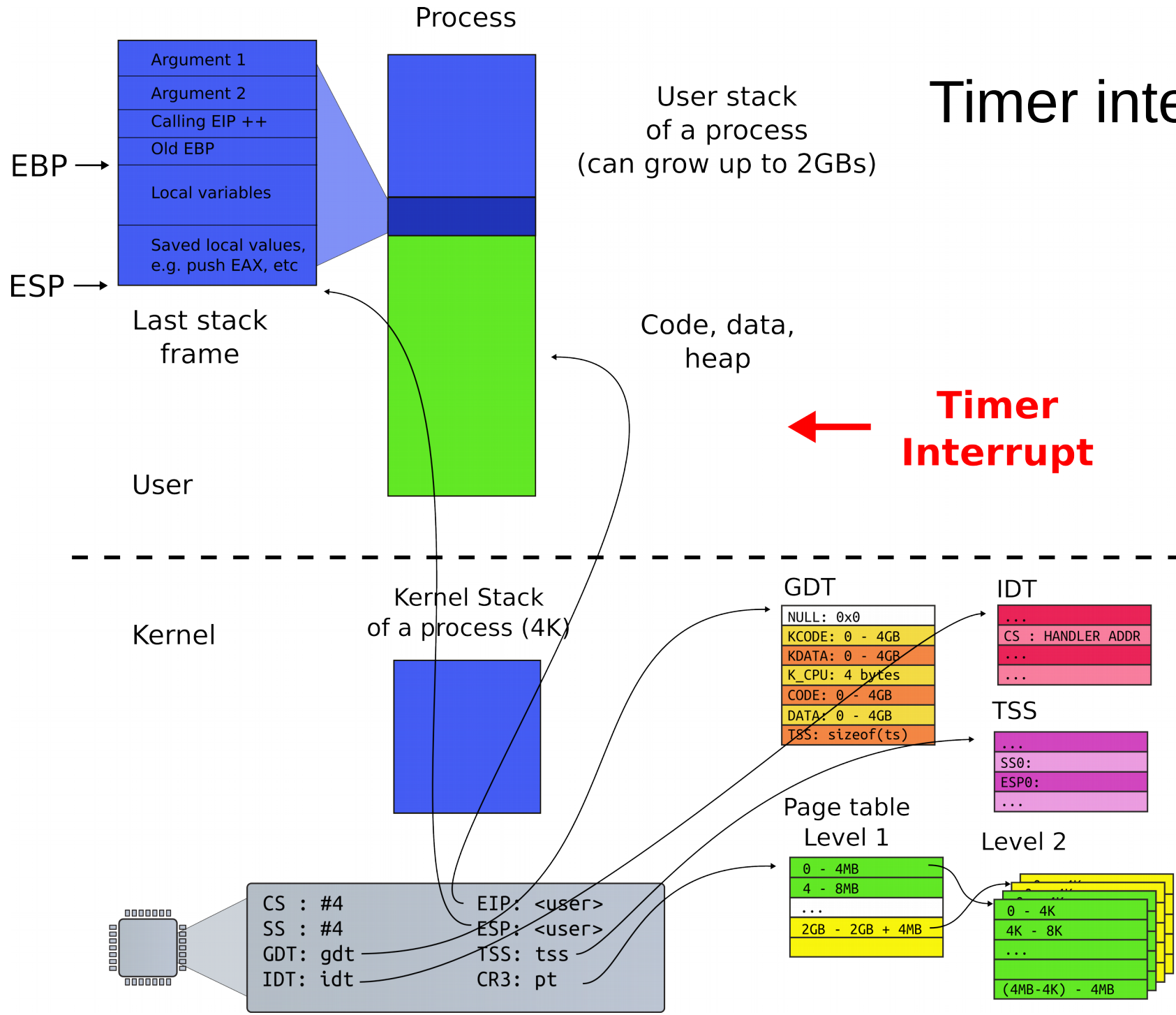
- User mode
- Two stacks
 - Kernel and user
 - Kernel stack is empty



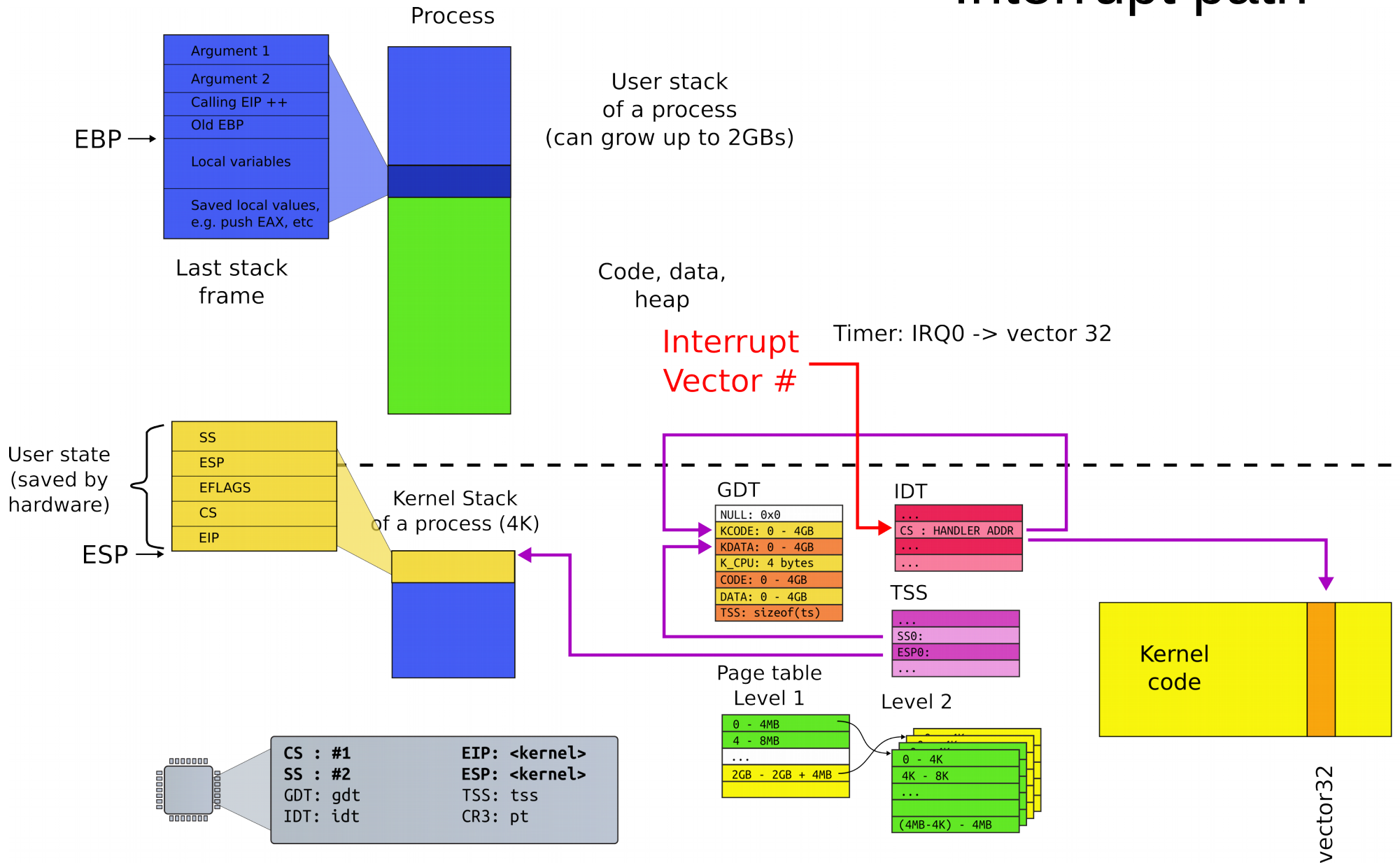


- Page table
- GDT

Timer interrupt



Interrupt path



Where does IDT (entry 32) point to?

```
vector32:
```

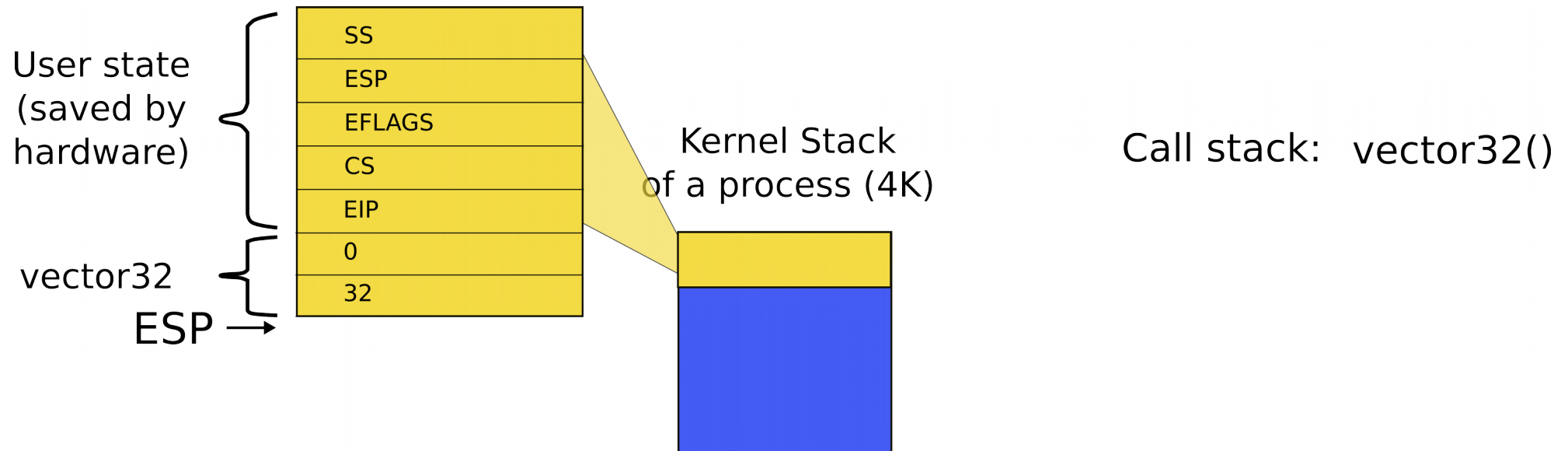
```
    pushl $0        // error code
```

```
    pushl $32      // vector #
```

```
    jmp alltraps
```

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

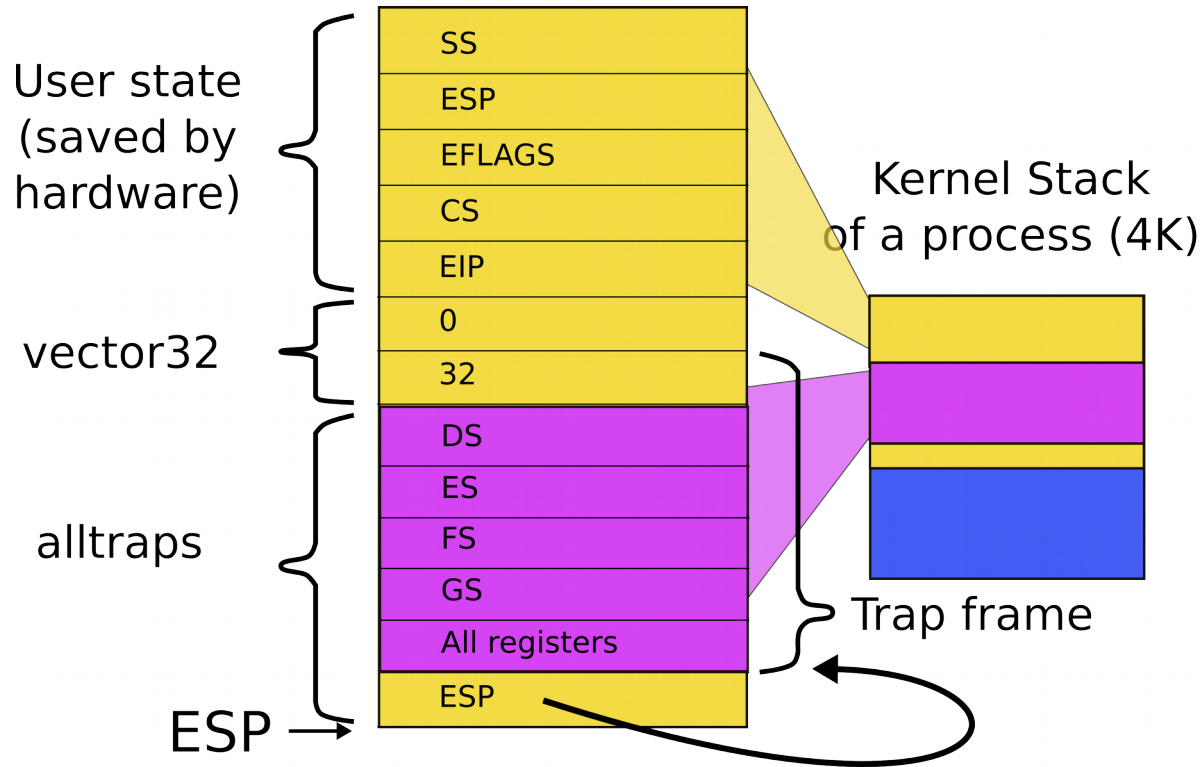
Kernel stack after interrupt



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()

```
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
```

alltraps()

```
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();
```

trap()

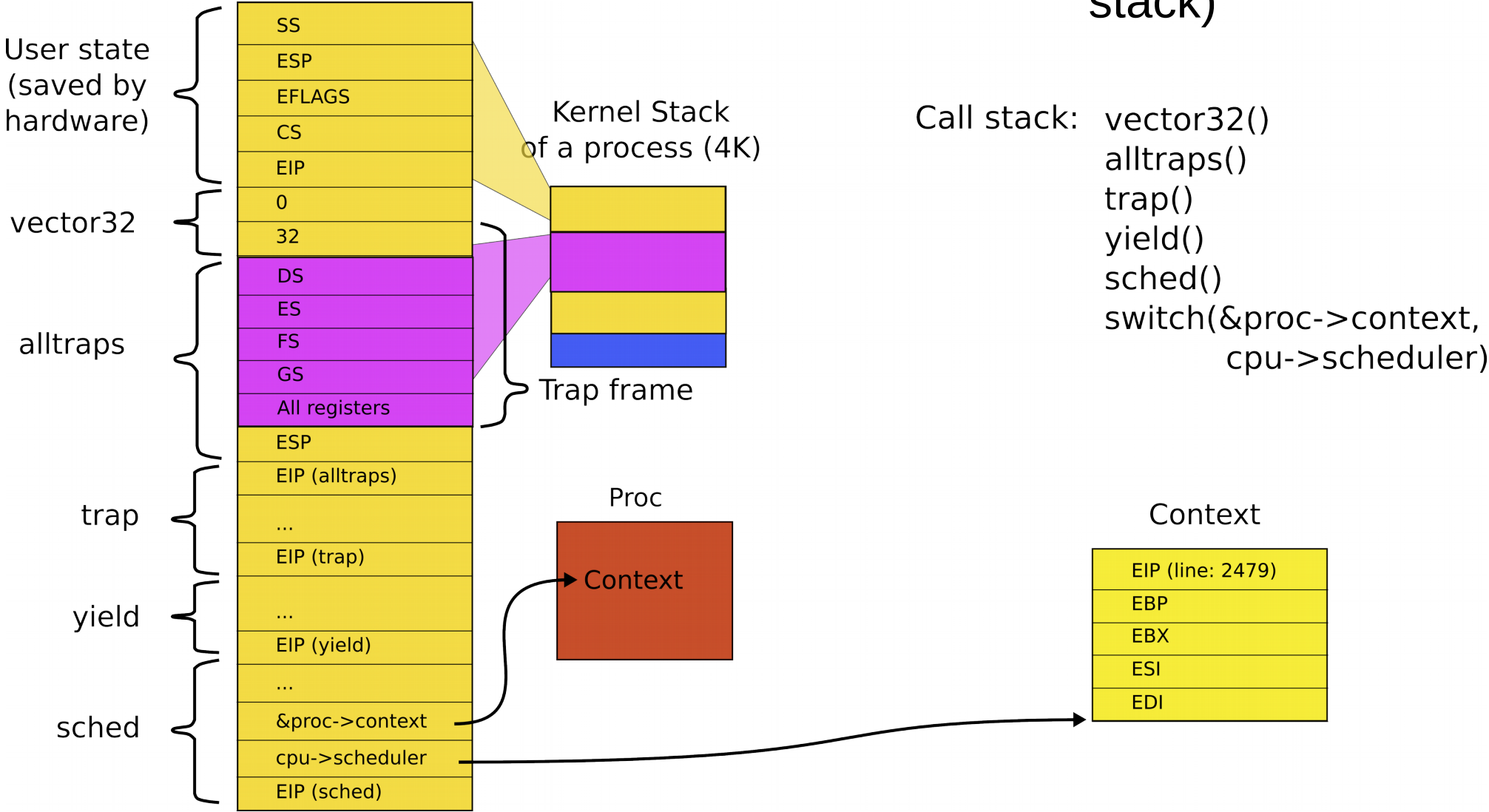
Invoke the scheduler

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

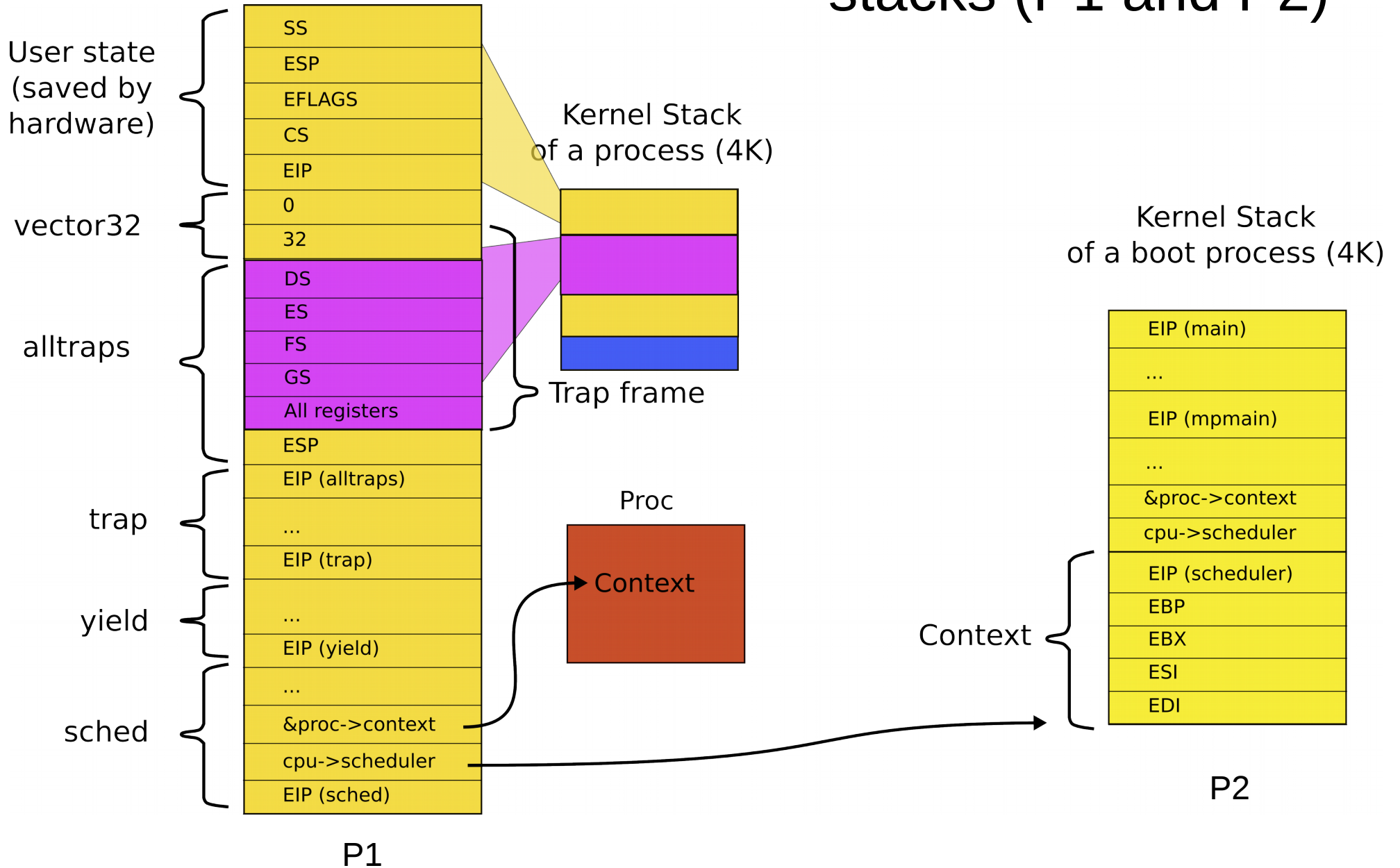
Start the context switch

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

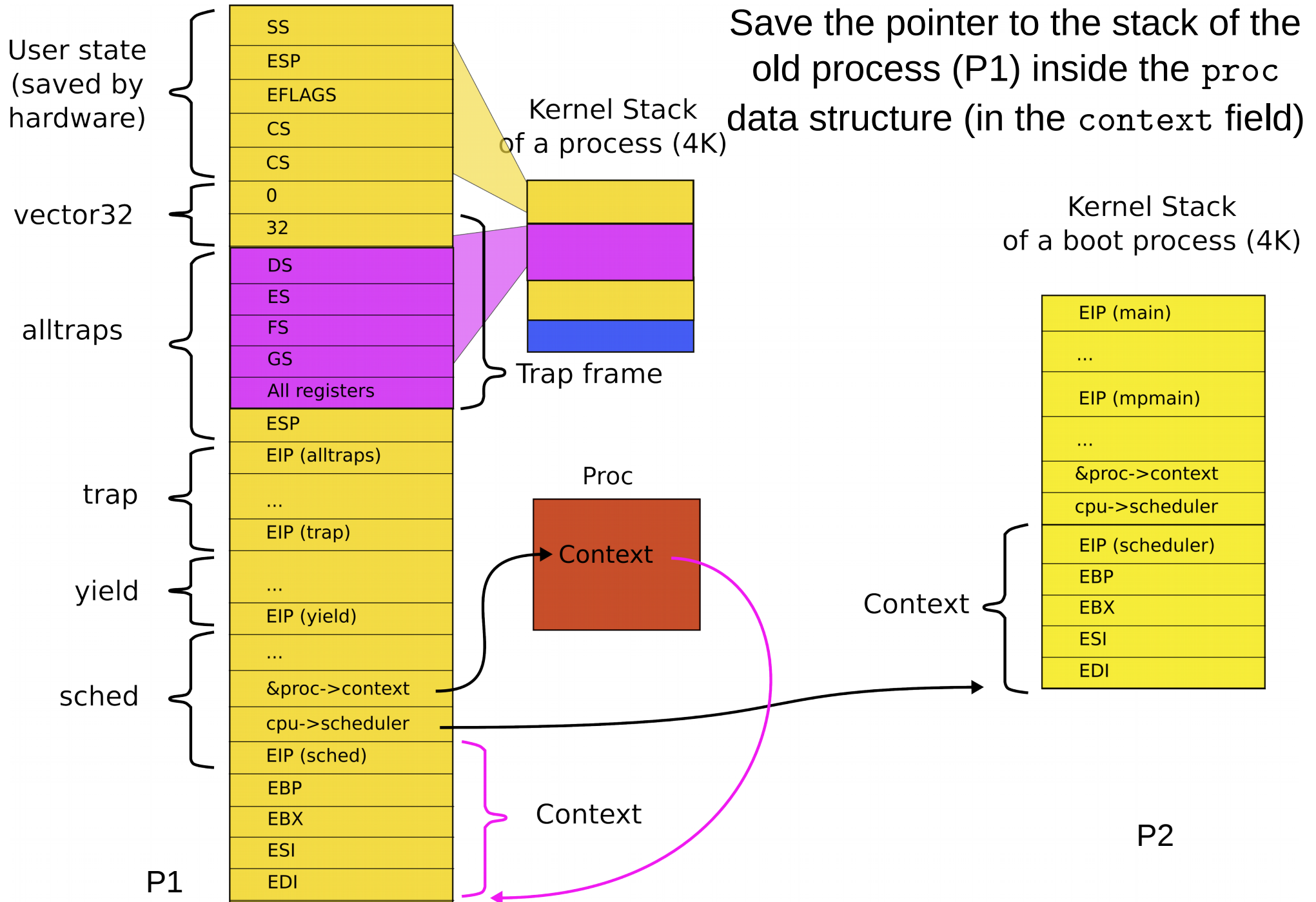
Stack inside switch() and its two arguments (passed on the stack)



Imagine you have two stacks (P1 and P2)

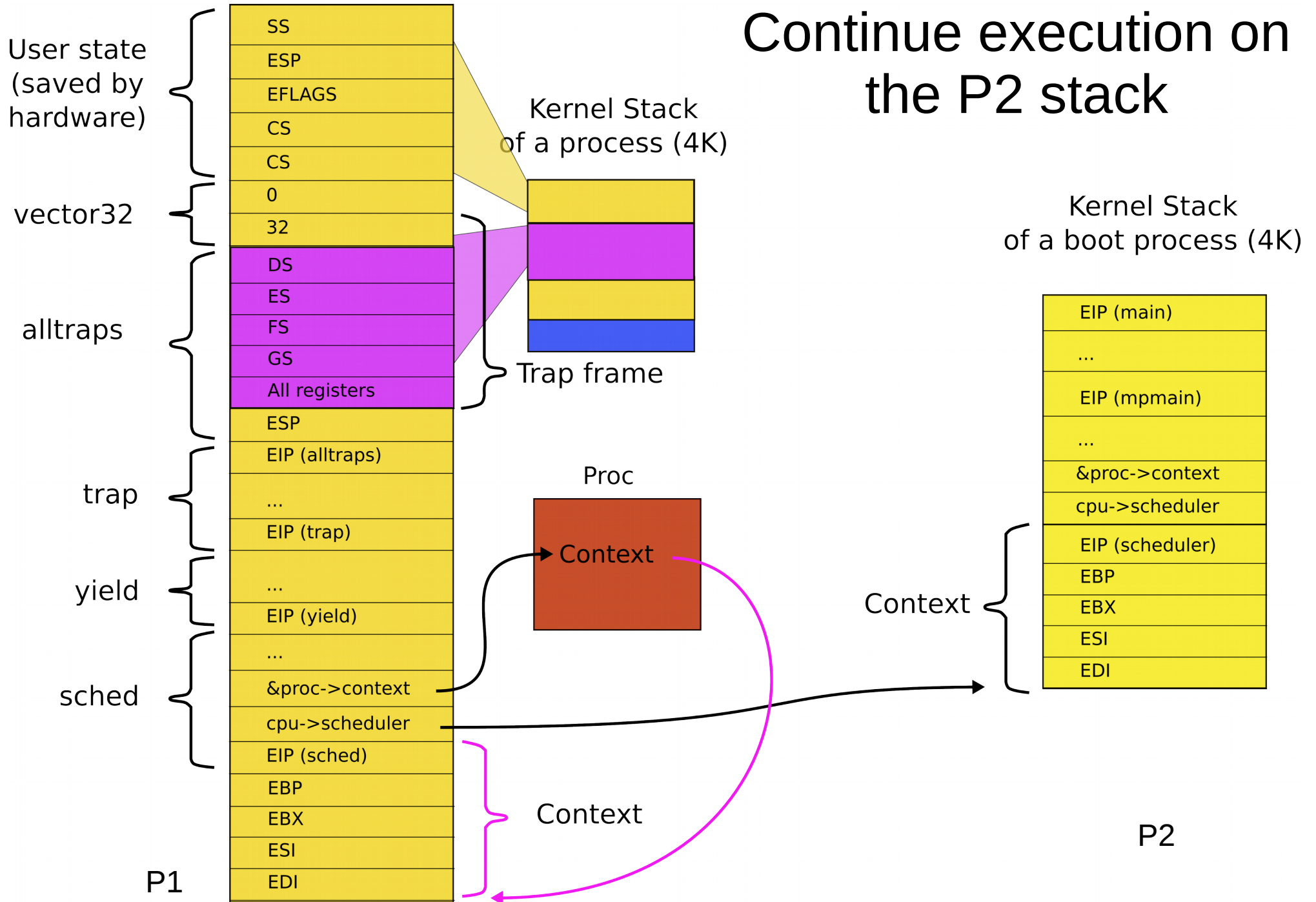


The context switch function should just save the pointer to the old stack (P1) and load the new stack (P2)



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

Continue execution on the P2 stack



```

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

```

swtch()

```

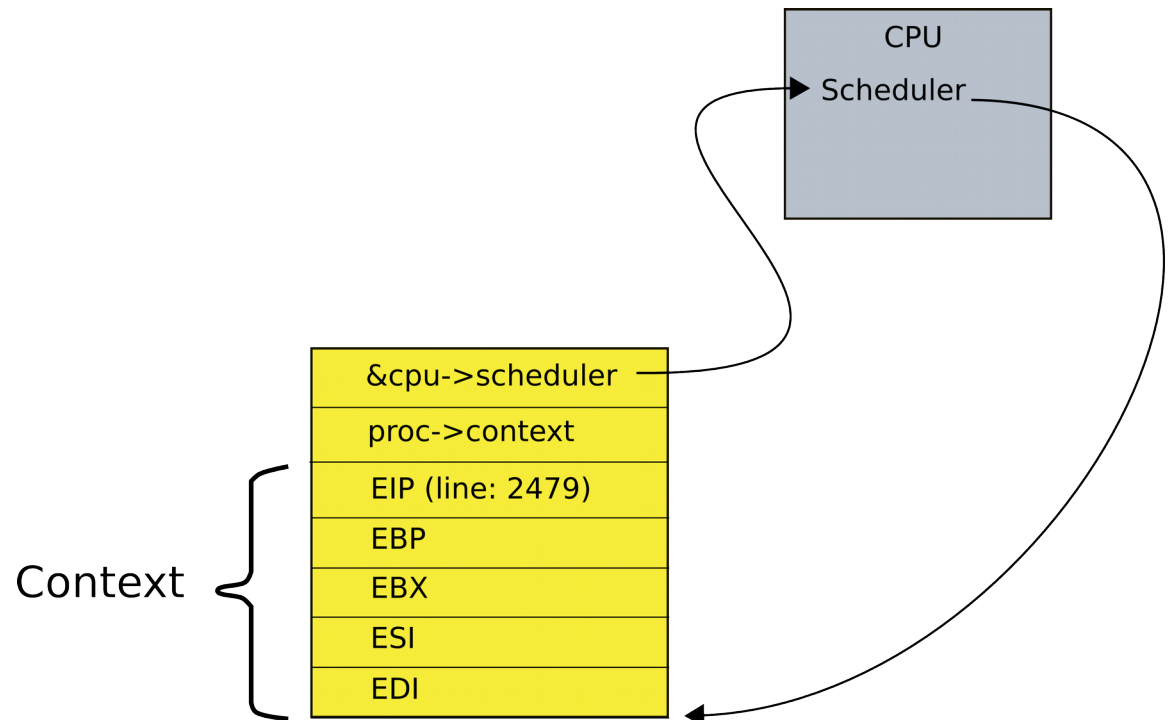
void swtch(struct context **old,
           struct context *new);

```

- 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

Context data structure

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

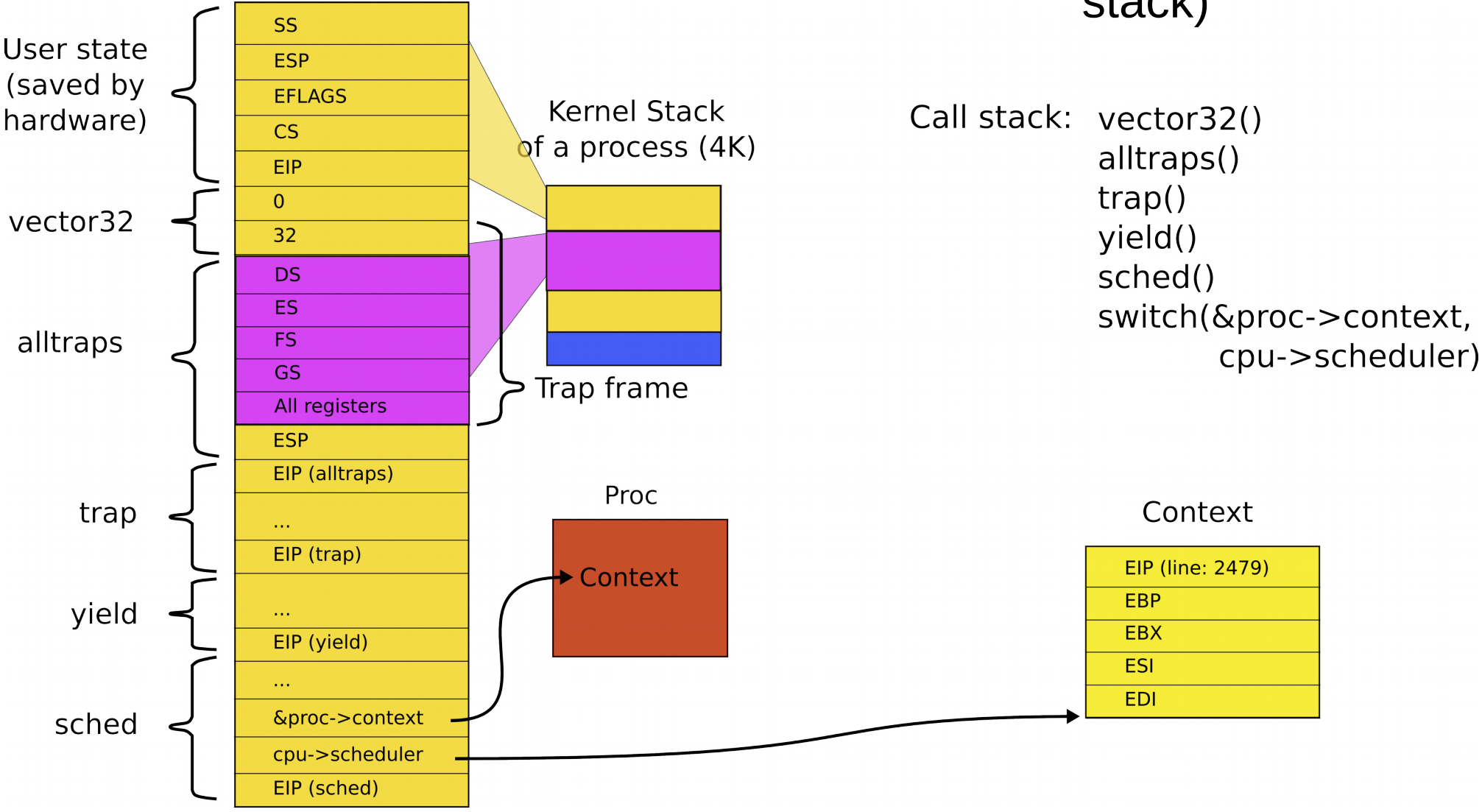


```
2958 swtch:
2959 movl 4(%esp), %eax
2960 movl 8(%esp), %edx
2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi
2967
2968 # Switch stacksh
2969 movl %esp, (%eax)
2970 movl %edx, %esp
2971
2972 # Load new callee-save registers
2973 popl %edi
2974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

swtch()

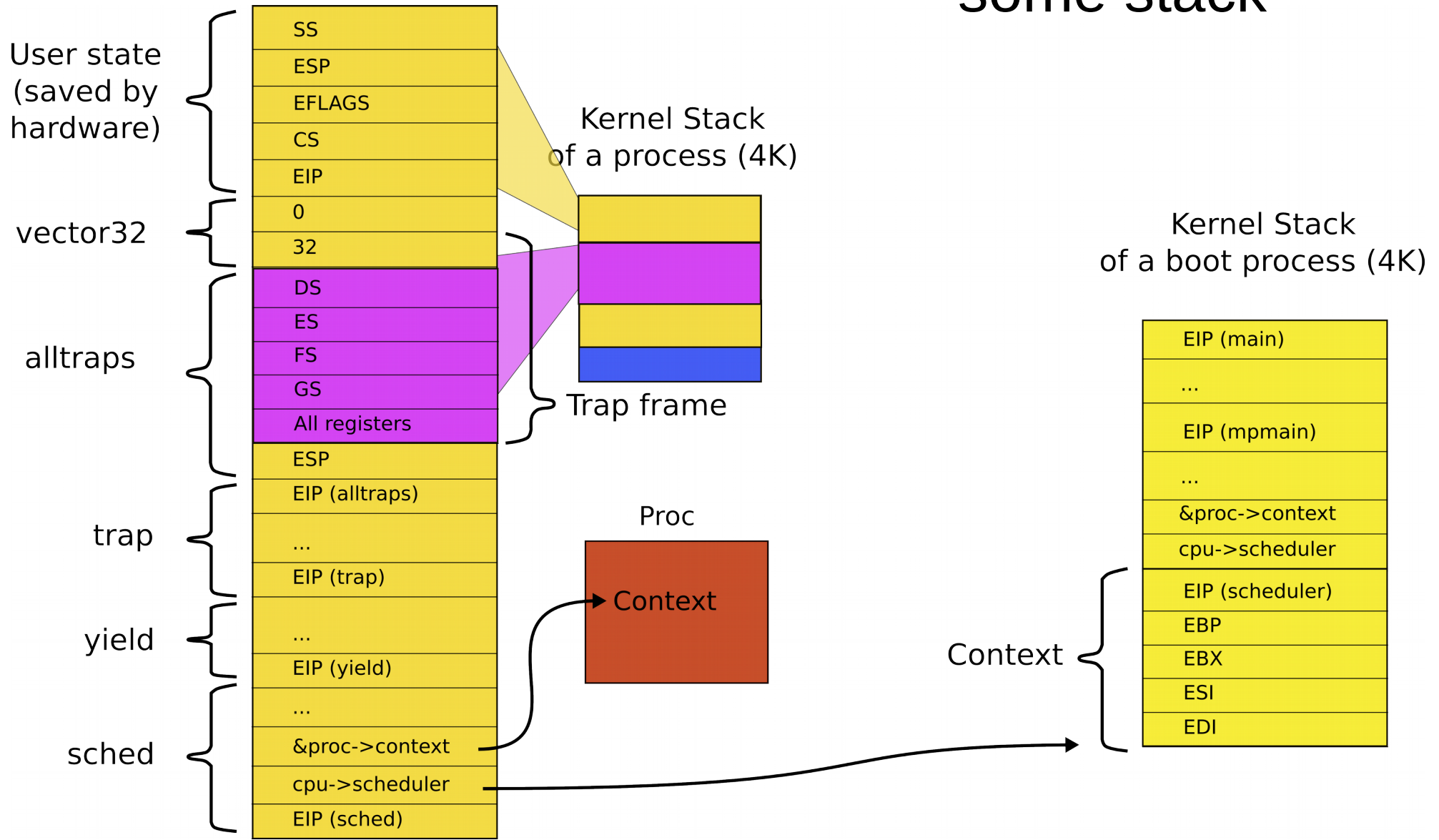
```
2093 struct context {
2094     uint edi;
2095     uint esi;
2096     uint ebx;
2097     uint ebp;
2098     uint eip;
2099 };
```


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



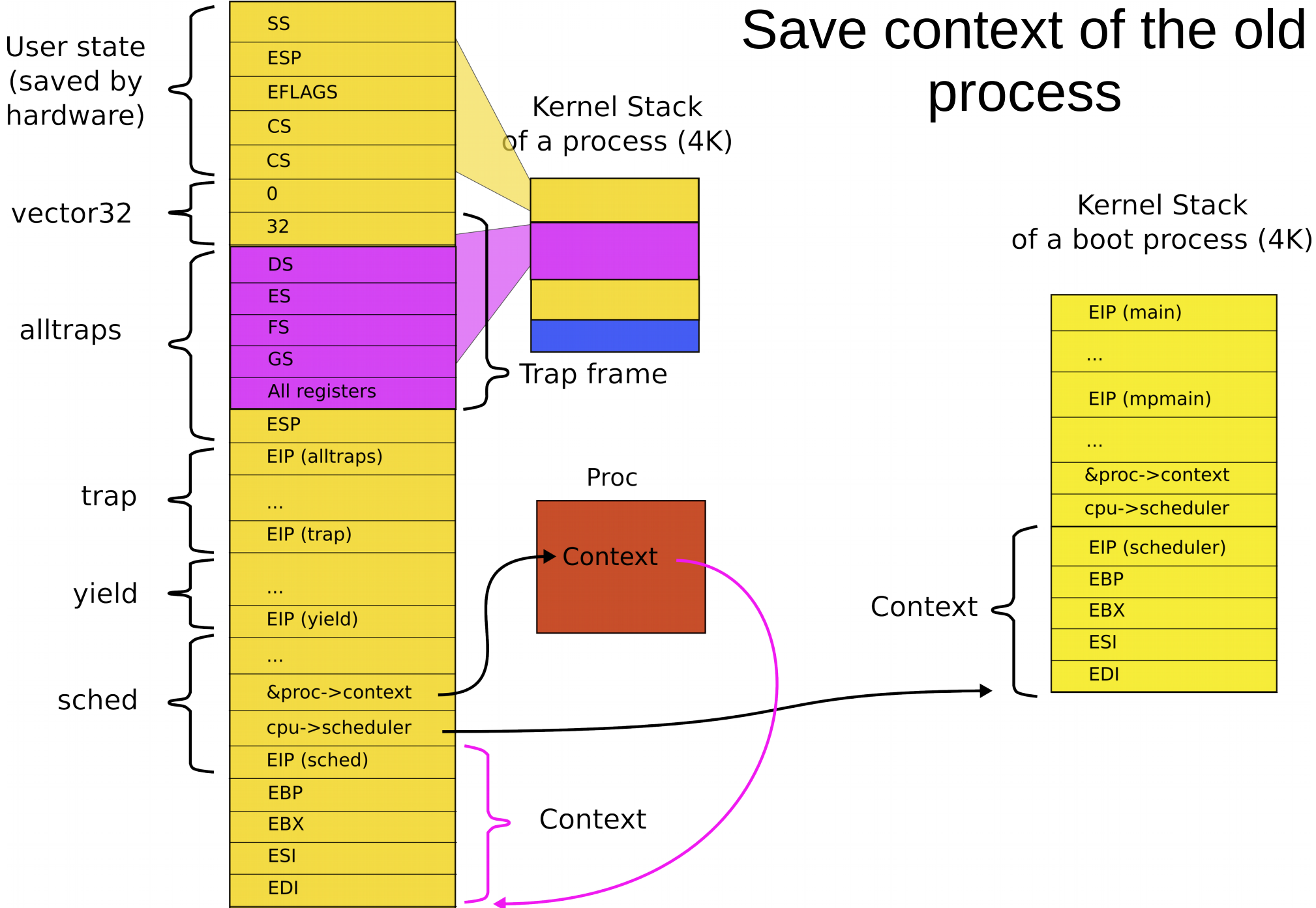
Main trick: context is always allocated on the top
of some stack

Context is always top of some stack



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

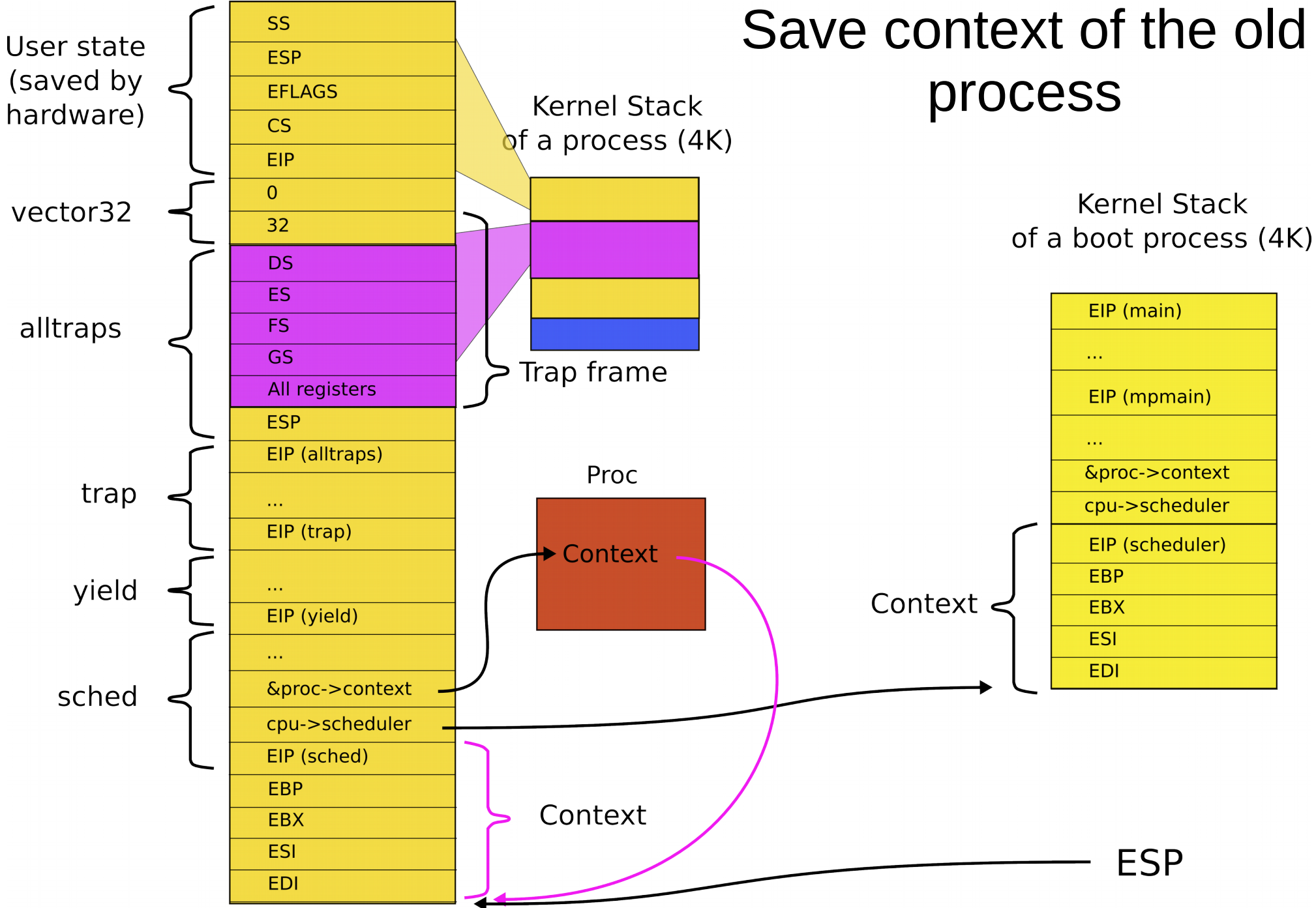
Save context of the old process



```
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)     // load current context (top of current stack) into
                        // the memory location pointed by *old
2970 movl %edx, %esp     // set stack to be equal to *new (the top of the new context)
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

swtch()

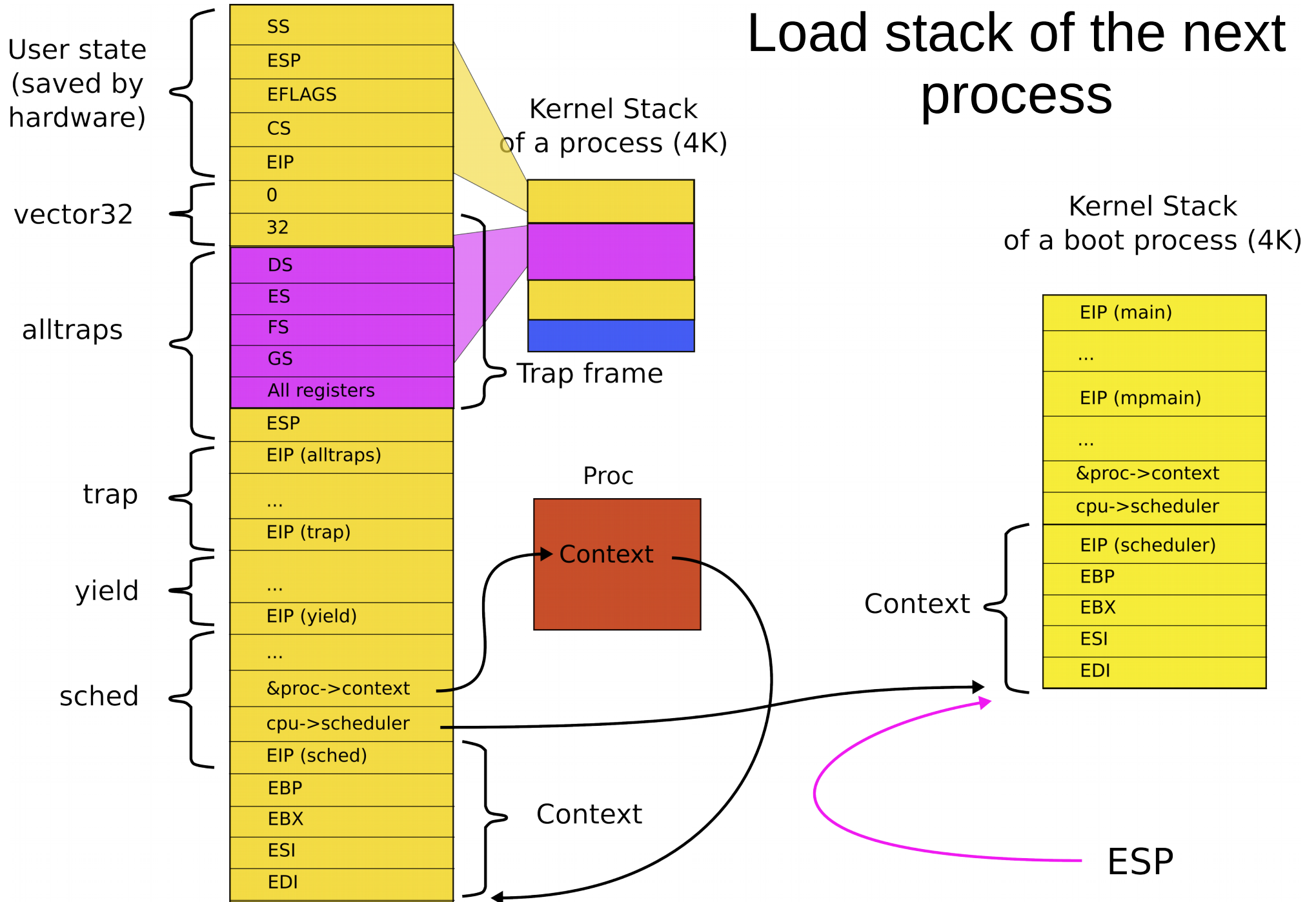
Save context of the old process



```
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)    // load current context (top of current stack) into
                        // the memory location pointed by *old
2970 movl %edx, %esp    // set stack to be equal to *new (the top of the new context)
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

swtch()

Load stack of the next process



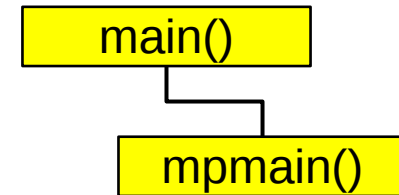
Context is always top of some stack...ok, but how?

- 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

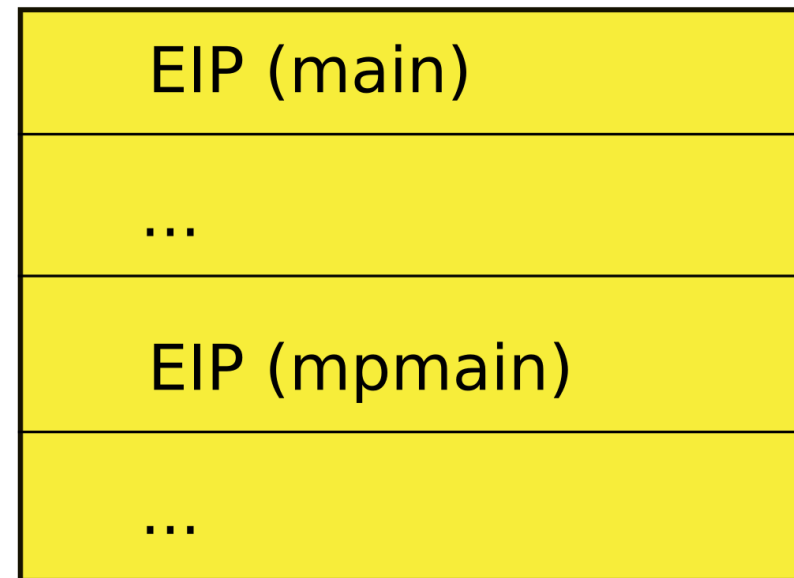
Scheduler()

```
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             switchvm(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

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);
```

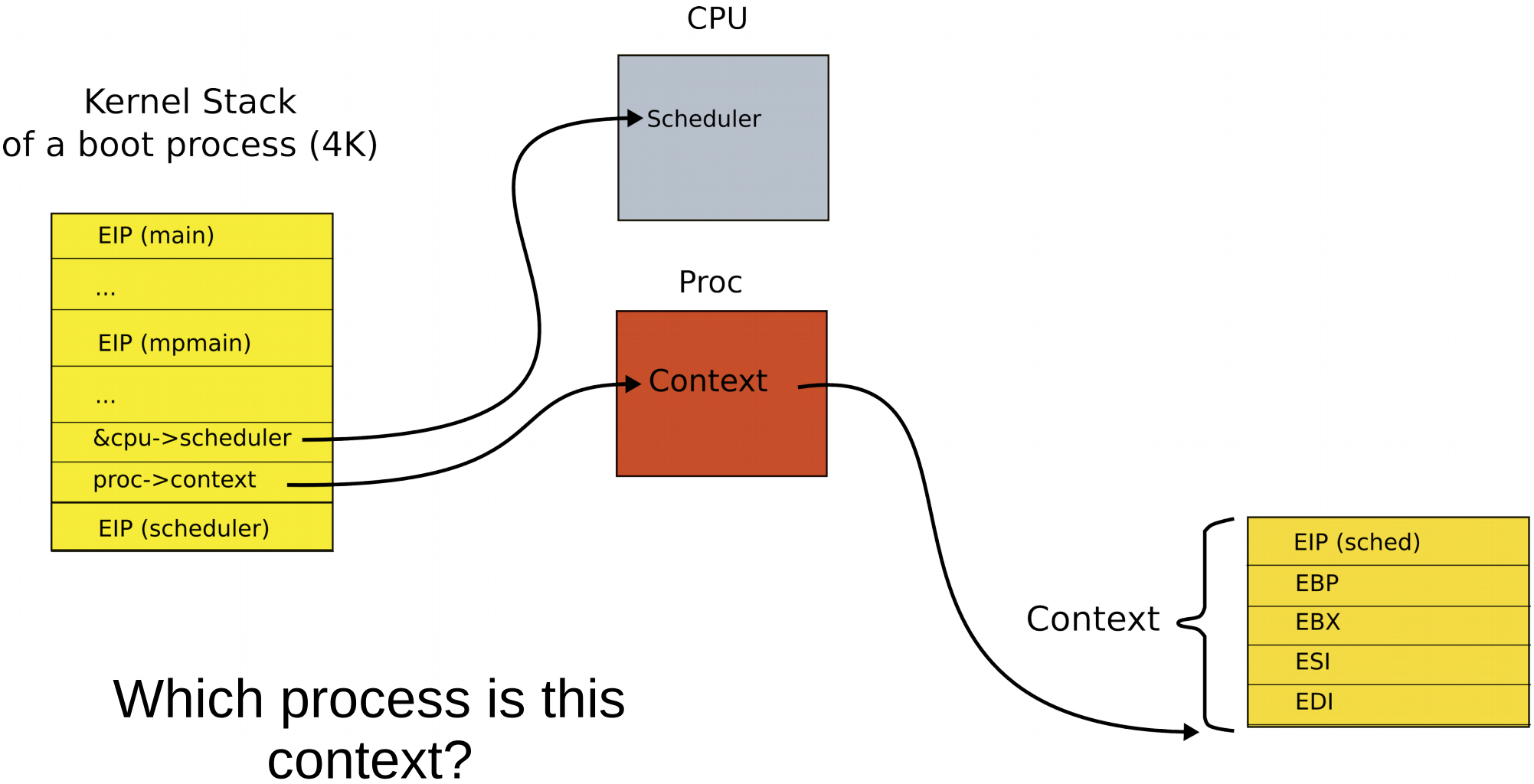
- We save the current context of the scheduler into:

```
&cpu->scheduler
```

- And restore the context of the first process

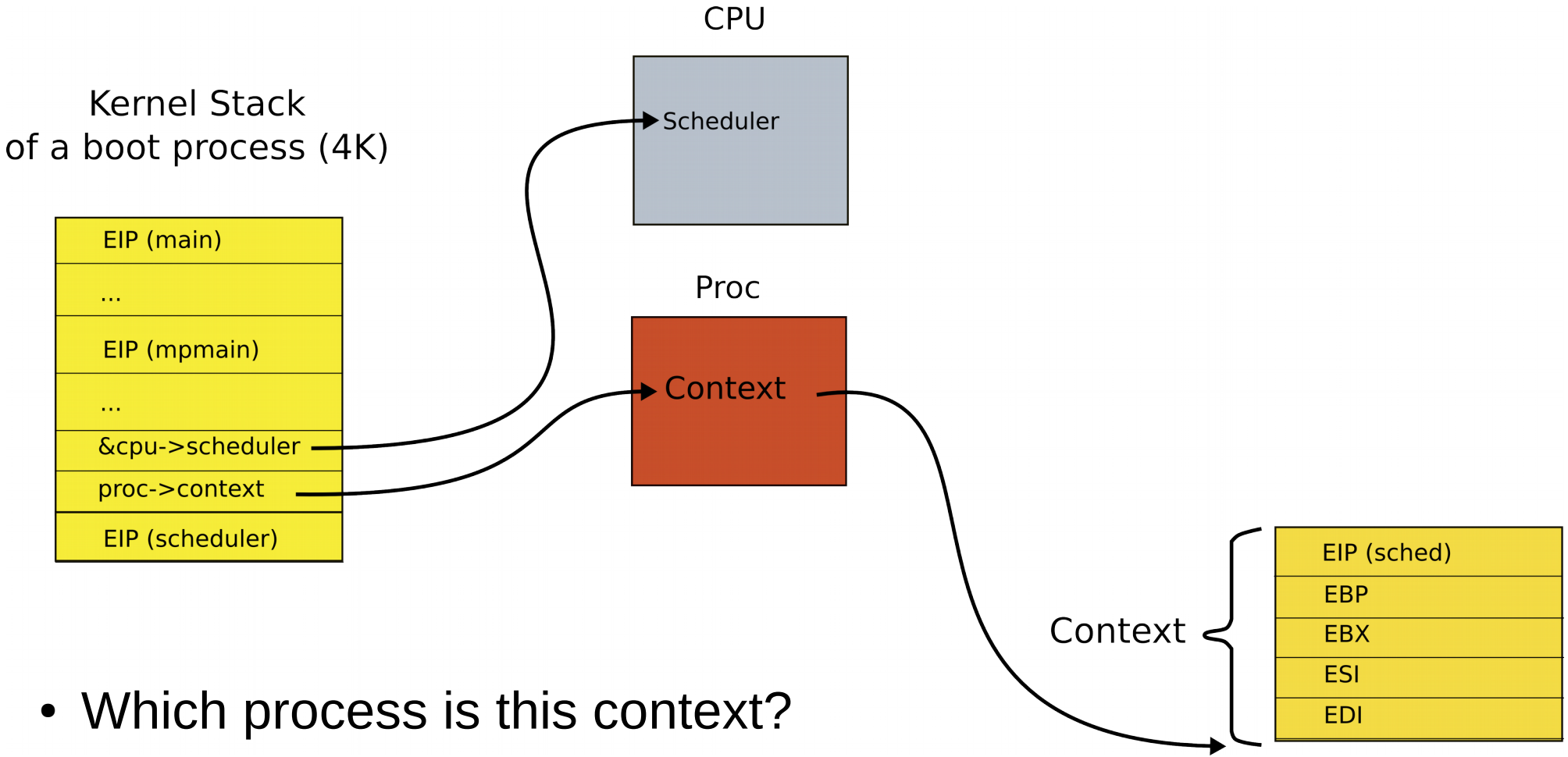
```
proc->context
```


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

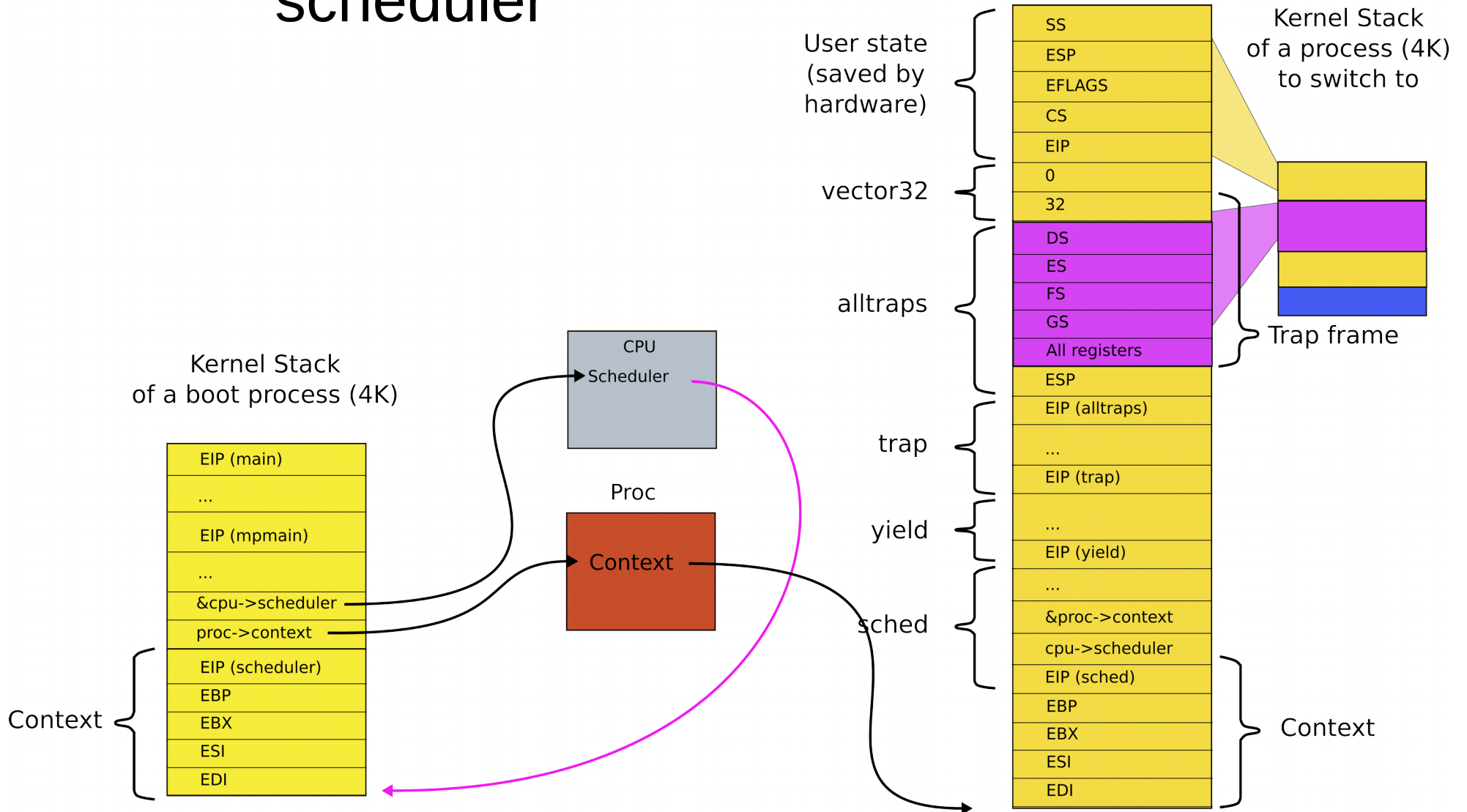


- Which process is this context?
- It's the context of the first process scheduler decides to run

```
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

Save context of the scheduler



```
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)    // load current context (top of current stack) into
                        // the memory location pointed by *old
2970 movl %edx, %esp    // set stack to be equal to *new (the top of the new context)
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

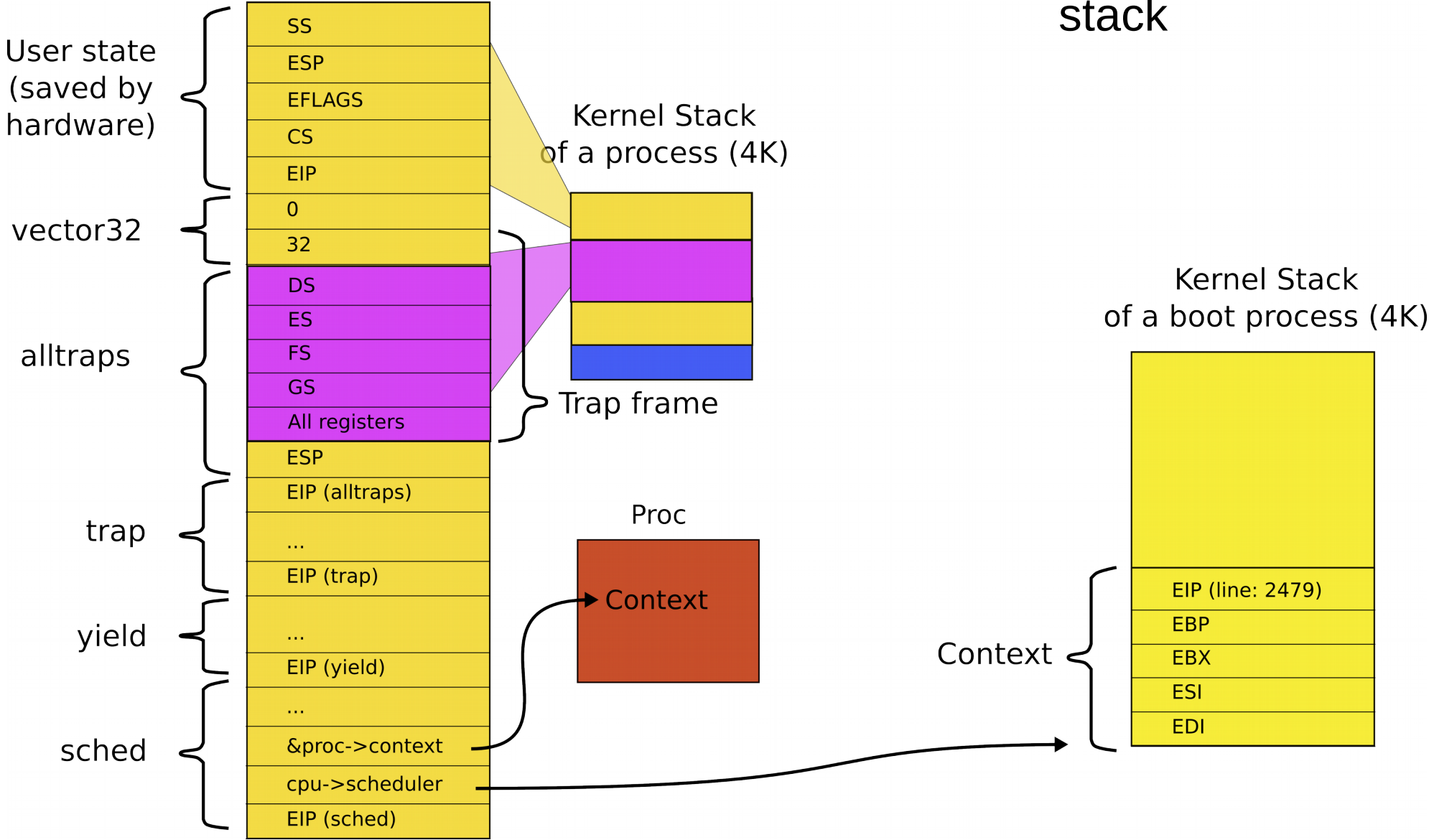
swtch()

The context is the top of some stack

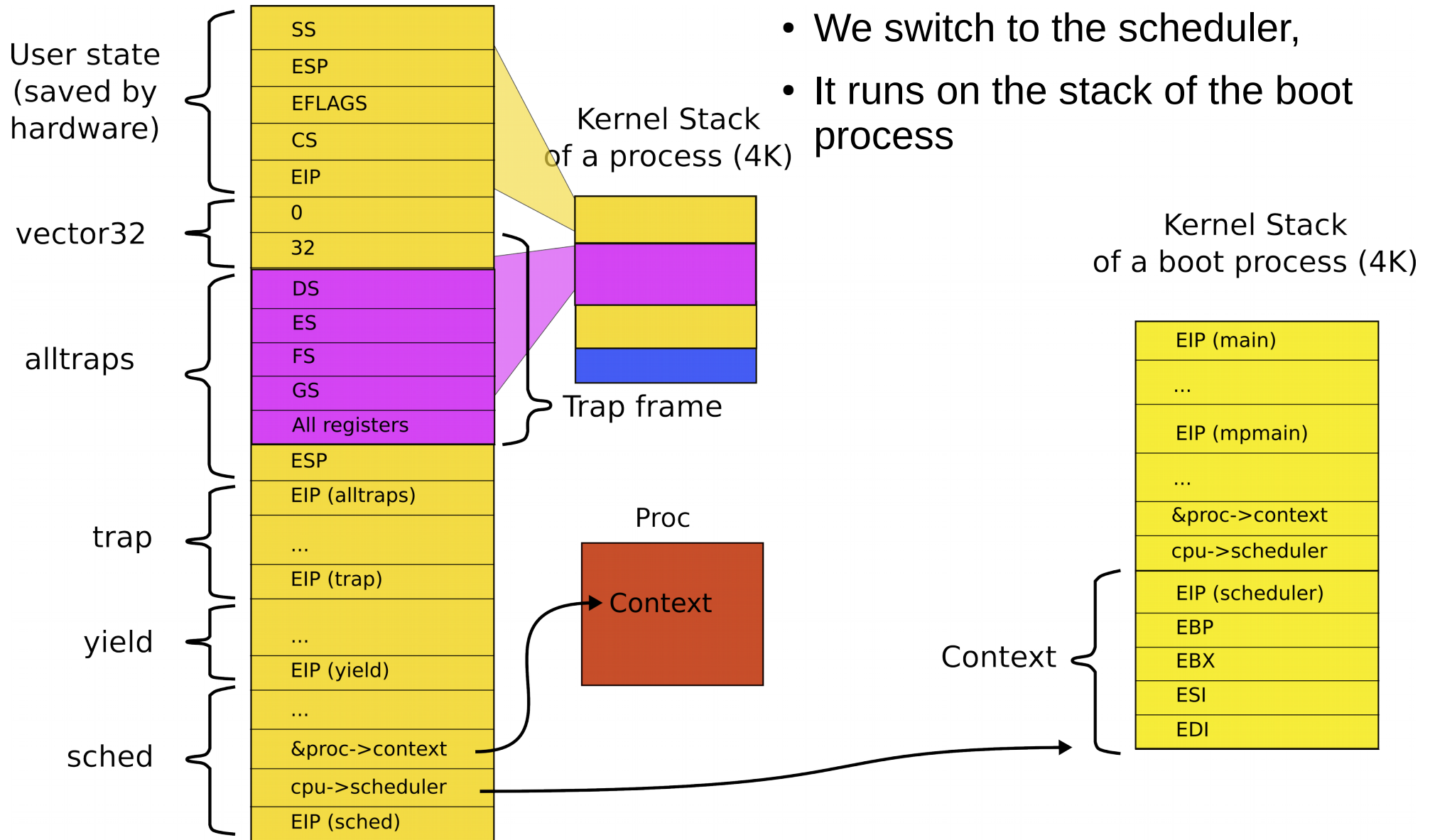
- Initially it was the stack of `mpenter()`
 - On which scheduler started
- Then first process...
 - Then scheduler again
 - And the next process...

Back to the original context switch
(end of the detour)

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



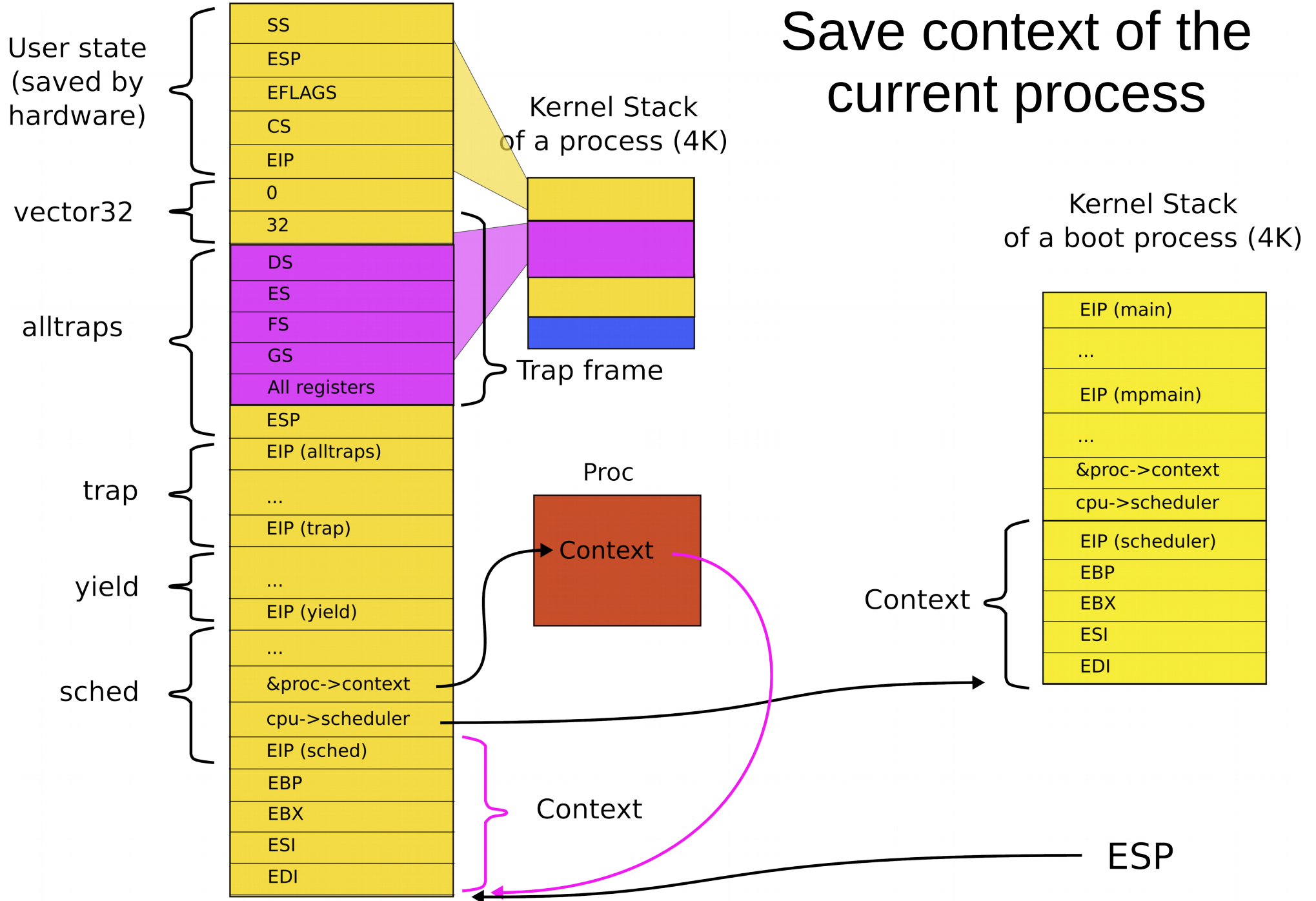
- Currently the ***new** context is the stack of the scheduler
- We switch to the scheduler,
- It runs on the stack of the boot process



```
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) into
                        // the memory location pointed by *old
2970 movl %edx, %esp     // set stack to be equal to *new (the top of the new context)
2971
2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

swtch()

Save context of the current process

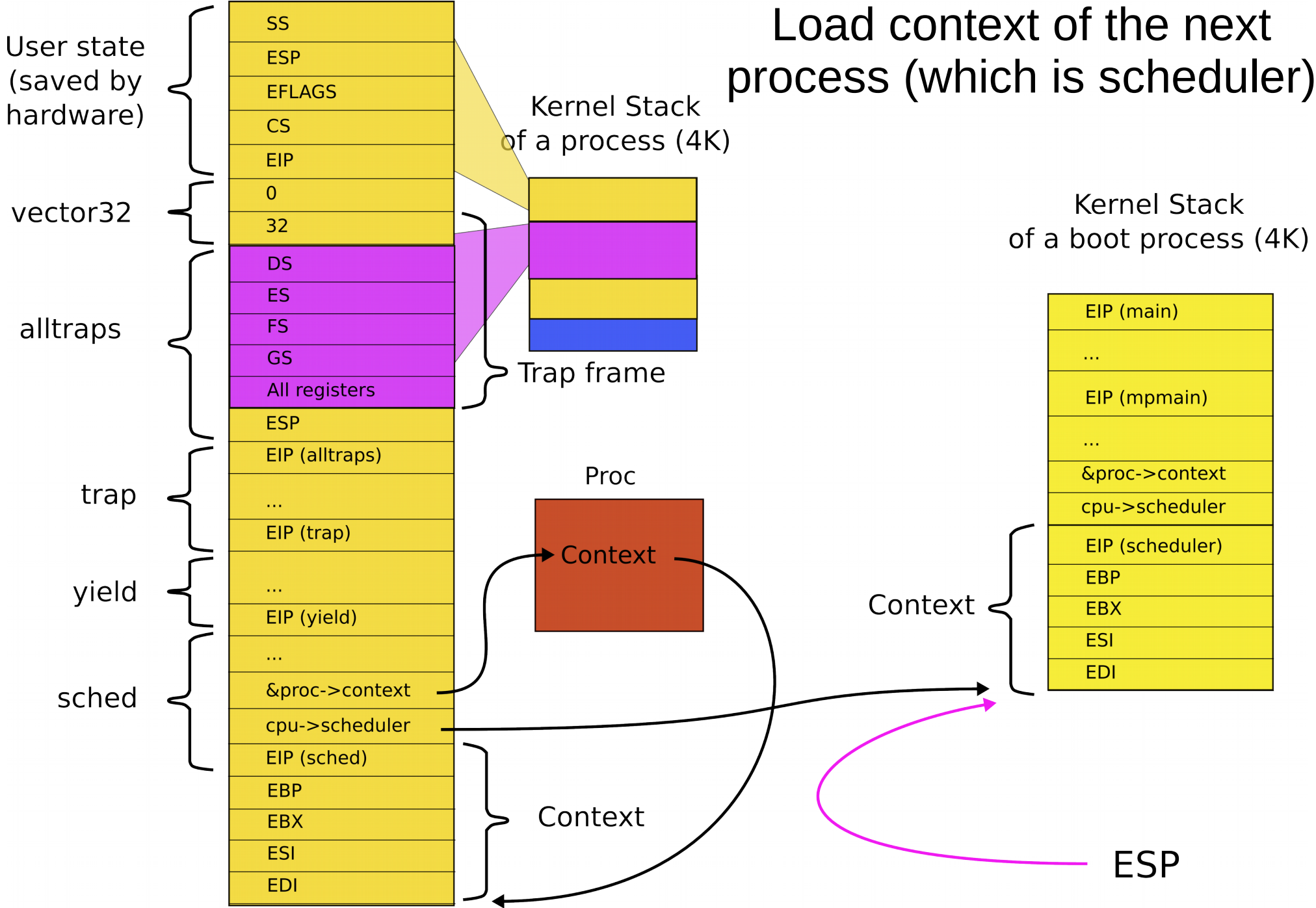


```
2958 swtch:
2959 movl 4(%esp), %eax    // struct context **old
2960 movl 8(%esp), %edx    // struct context *new
2961
2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi
2967
2968 # Switch stacks
2969 movl %esp, (%eax)    // save current context (top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

swtch(): load next context

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

Load context of the next process (which is scheduler)



And now: exit from swtch()

```
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
```

Where does this `switch()` return?

Where does this swtch() return?

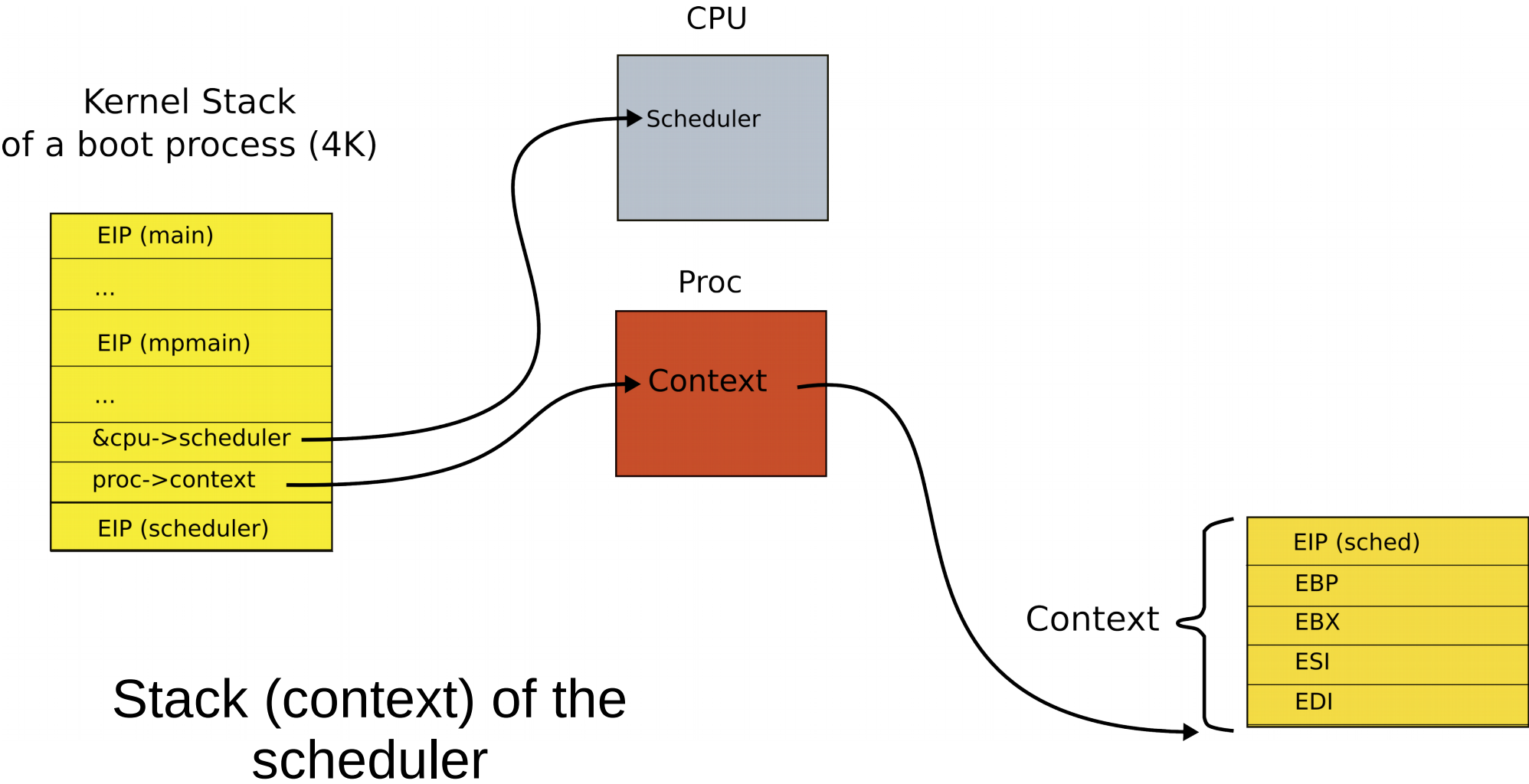
- Scheduler
- After all remember
 - We started with timer interrupt
 - Entered the kernel
 - Entered schedule()
 - Entered switch
- And are currently on our way from the process into the scheduler

What does scheduler do?

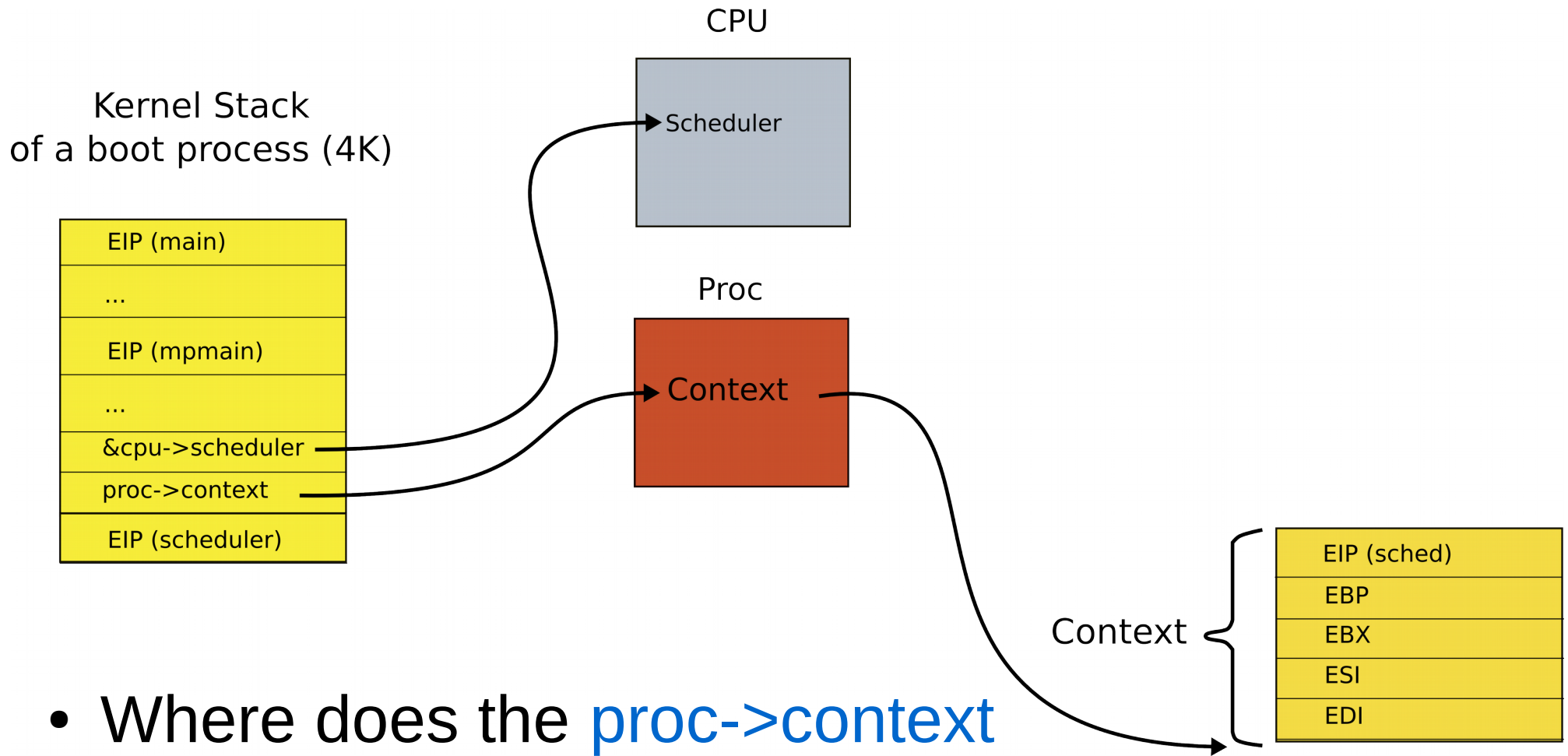
```
2458 scheduler(void)
2459 {
2462     for(;;){
2468         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2469             if(p->state != RUNNABLE)
2470                 continue;
2475             proc = p;
2476             switchvm(p);
2477             p->state = RUNNING;
2478             swtch(&cpu->scheduler,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

- Chooses next process to run
- Switches to it

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

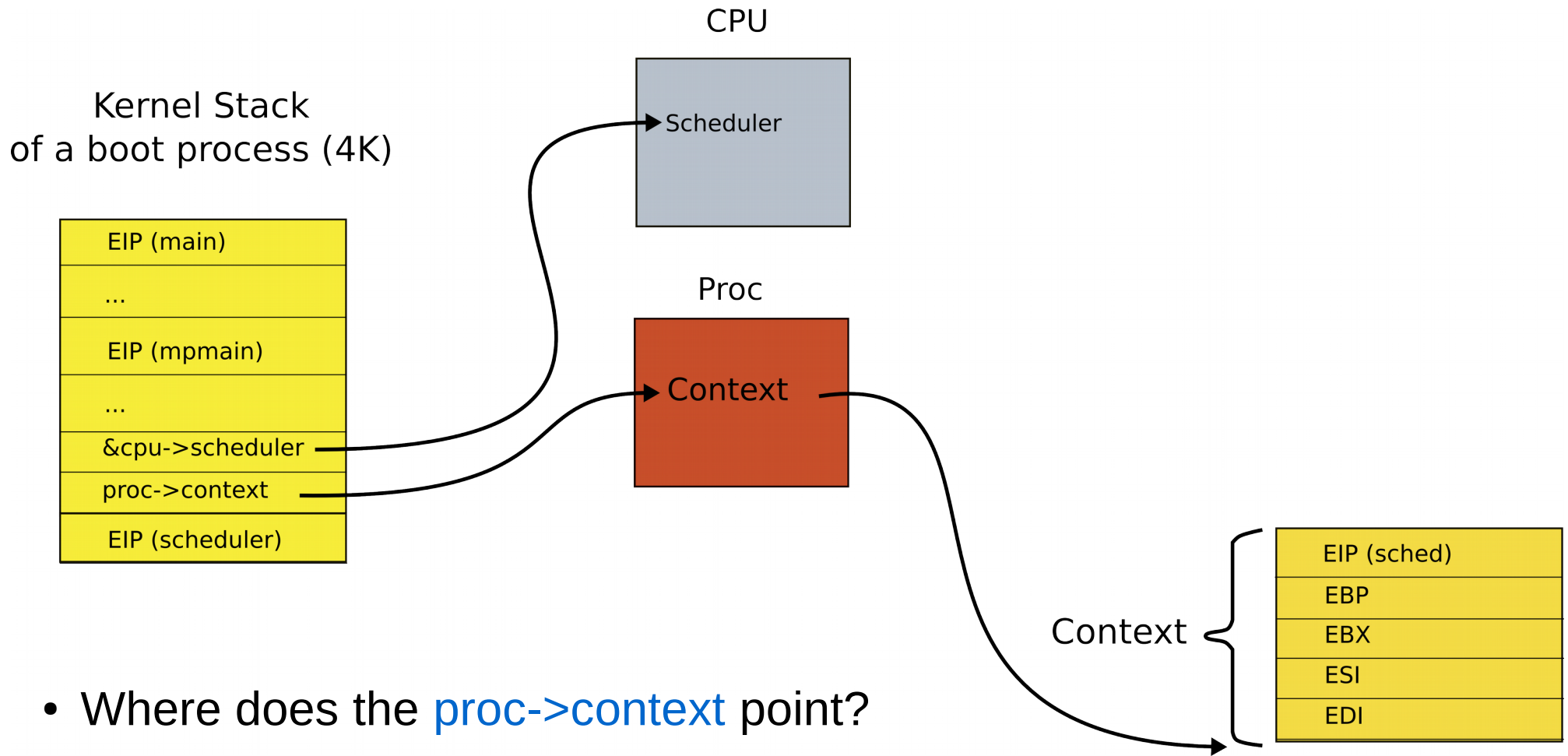


What does stack look like when scheduler() invokes switch()?



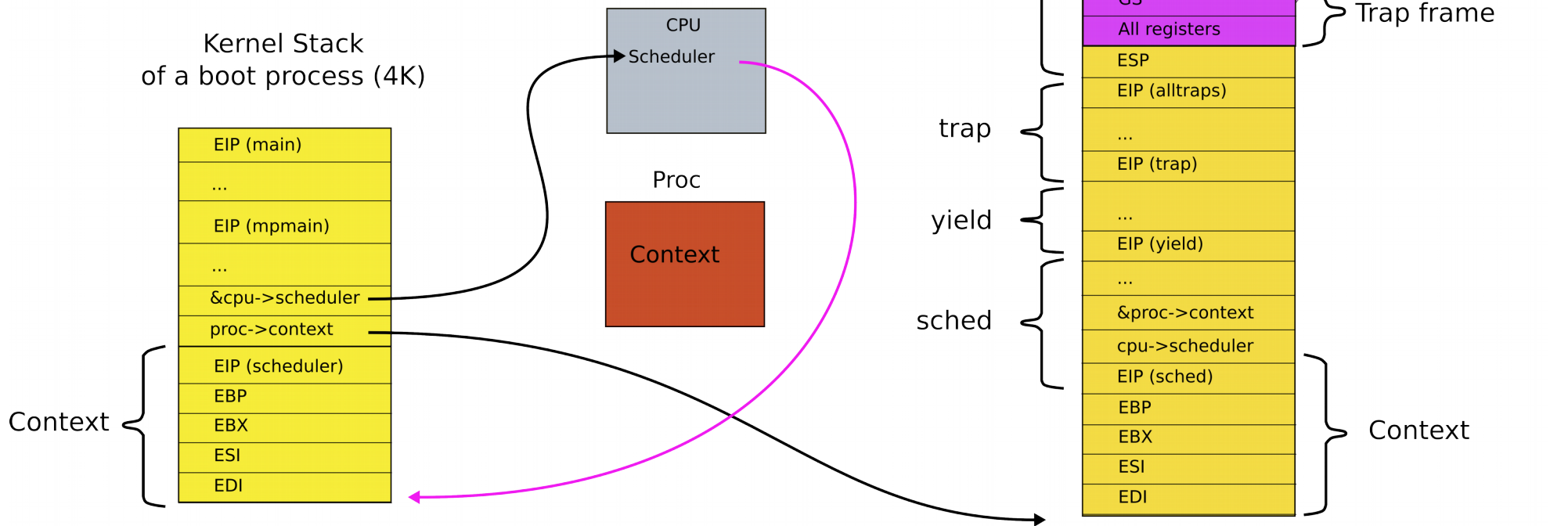
- Where does the `proc->context` point?

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



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

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



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

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

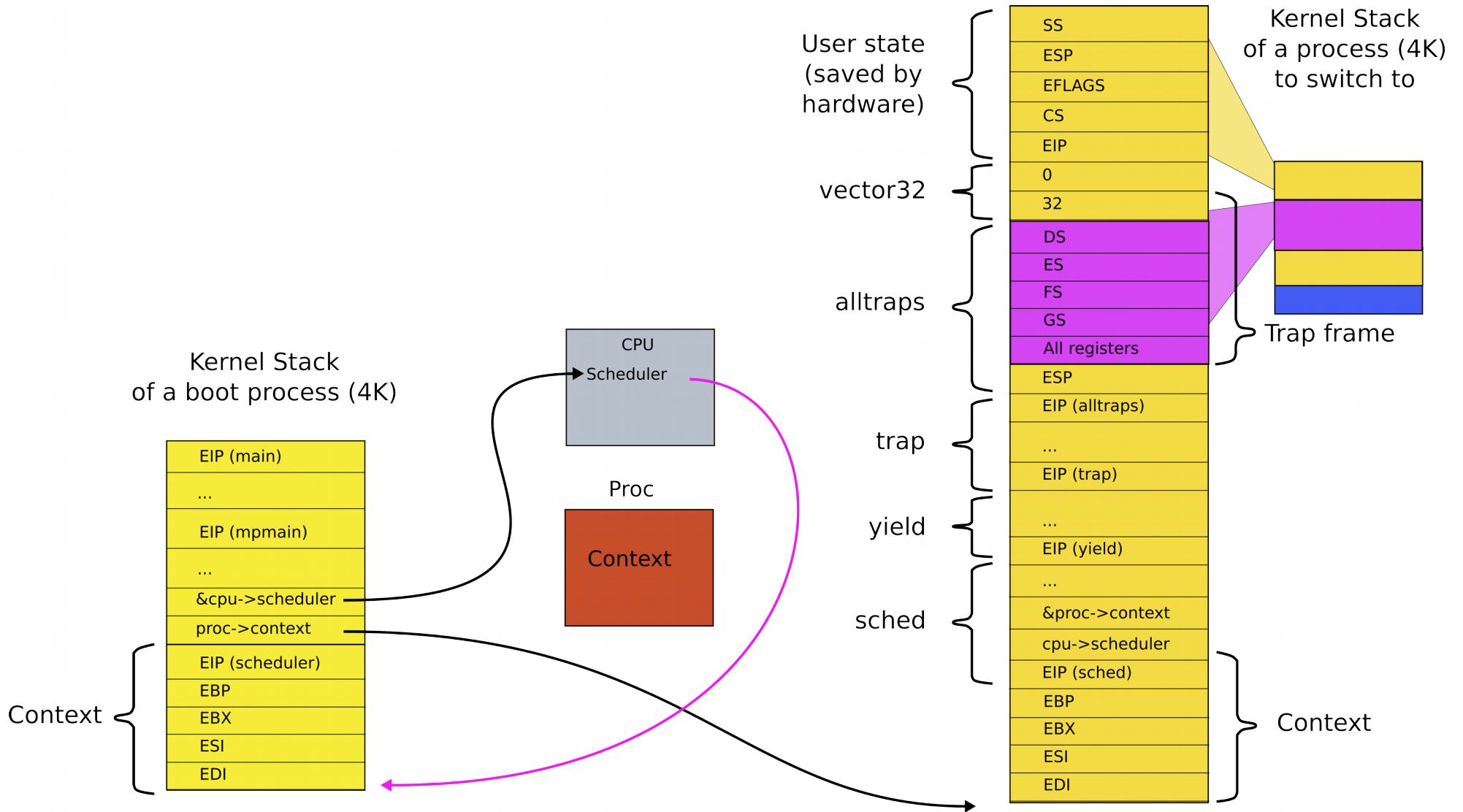
- Hence, we save context of the scheduler into

```
&cpu->scheduler
```

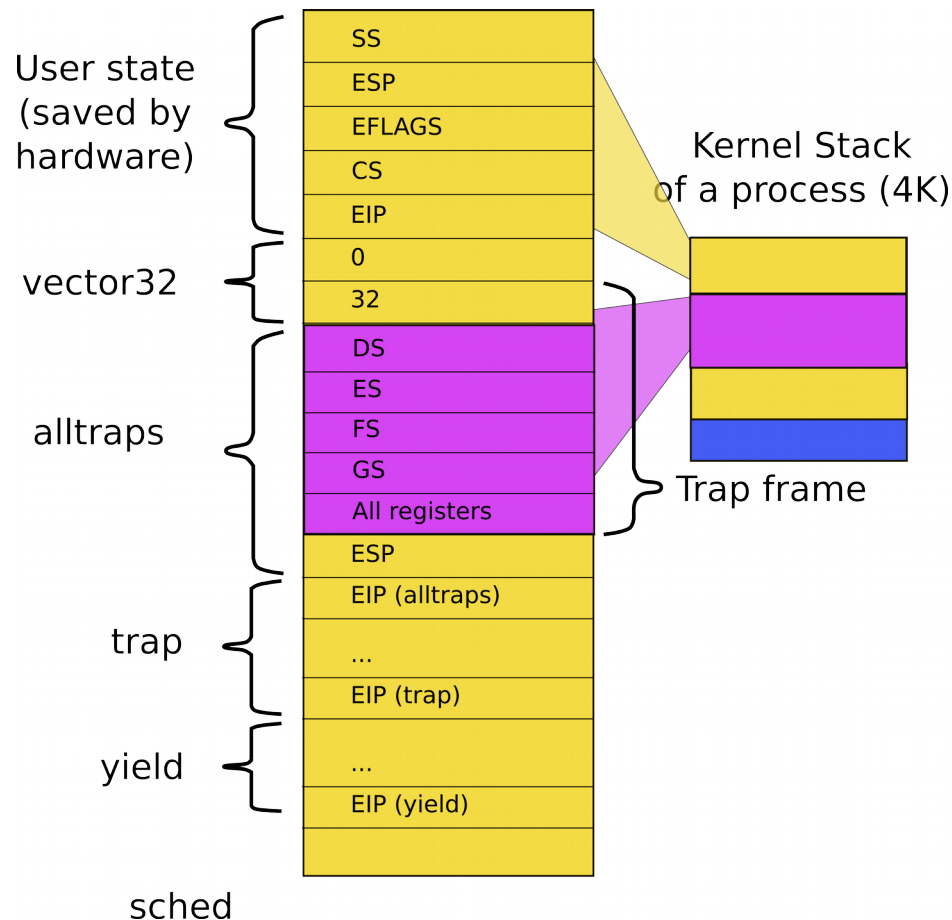
- And restore

```
proc->context
```

Stacks and contexts inside the switch()



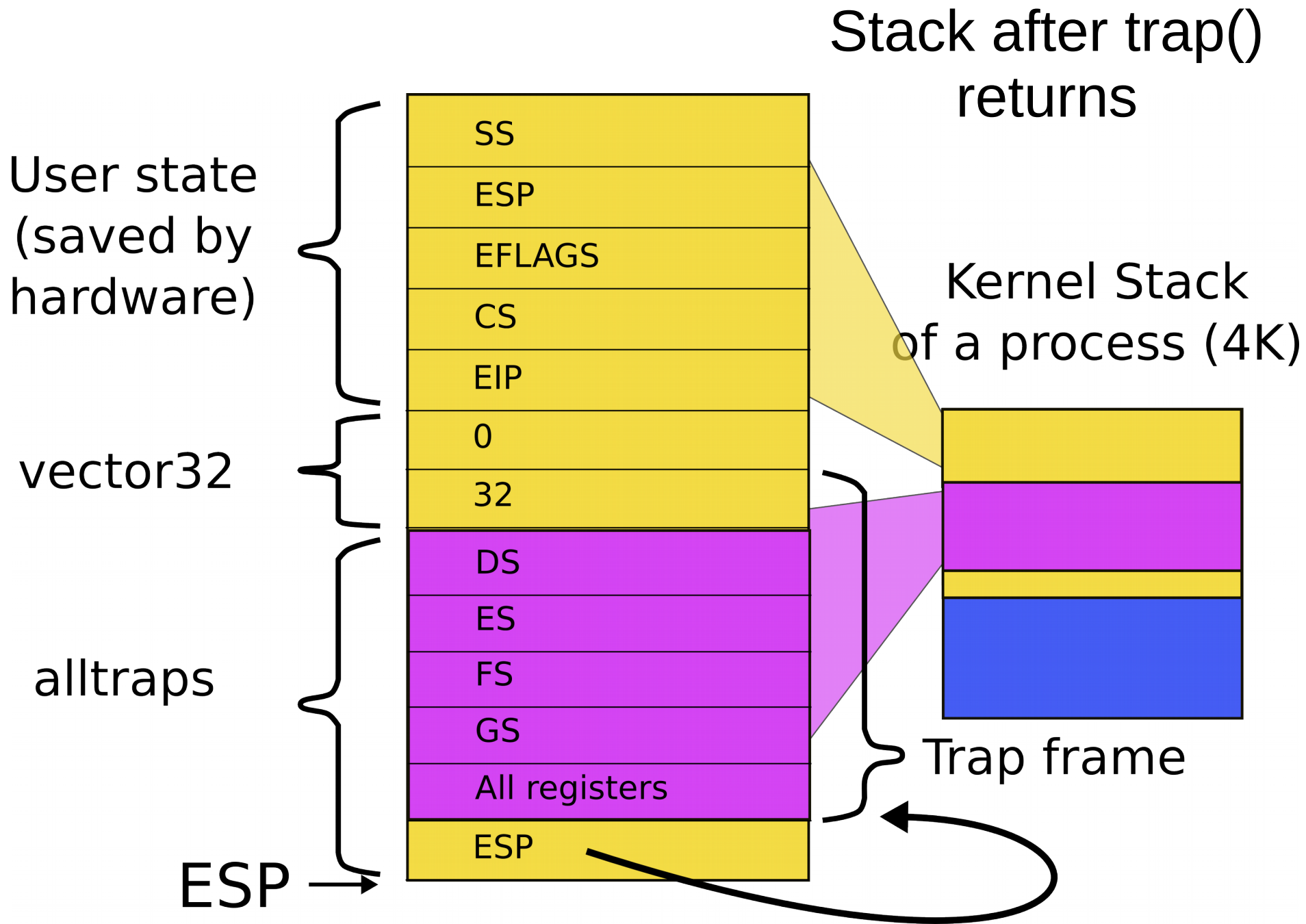
Exiting back to user-level



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


```
3004 alltraps:
...
3020 # Call trap(tf), where tf=%esp
3021 pushl %esp
3022 call trap
3023 addl $4, %esp
3024
3025 # Return falls through to trapret...
3026 .globl trapret
3027 trapret:
3028 popal
3029 popl %gs
3030 popl %fs
3031 popl %es
3032 popl %ds
3033 addl $0x8, %esp # trapno and errcode
3034 iret
```

alltraps(): exit into user-level

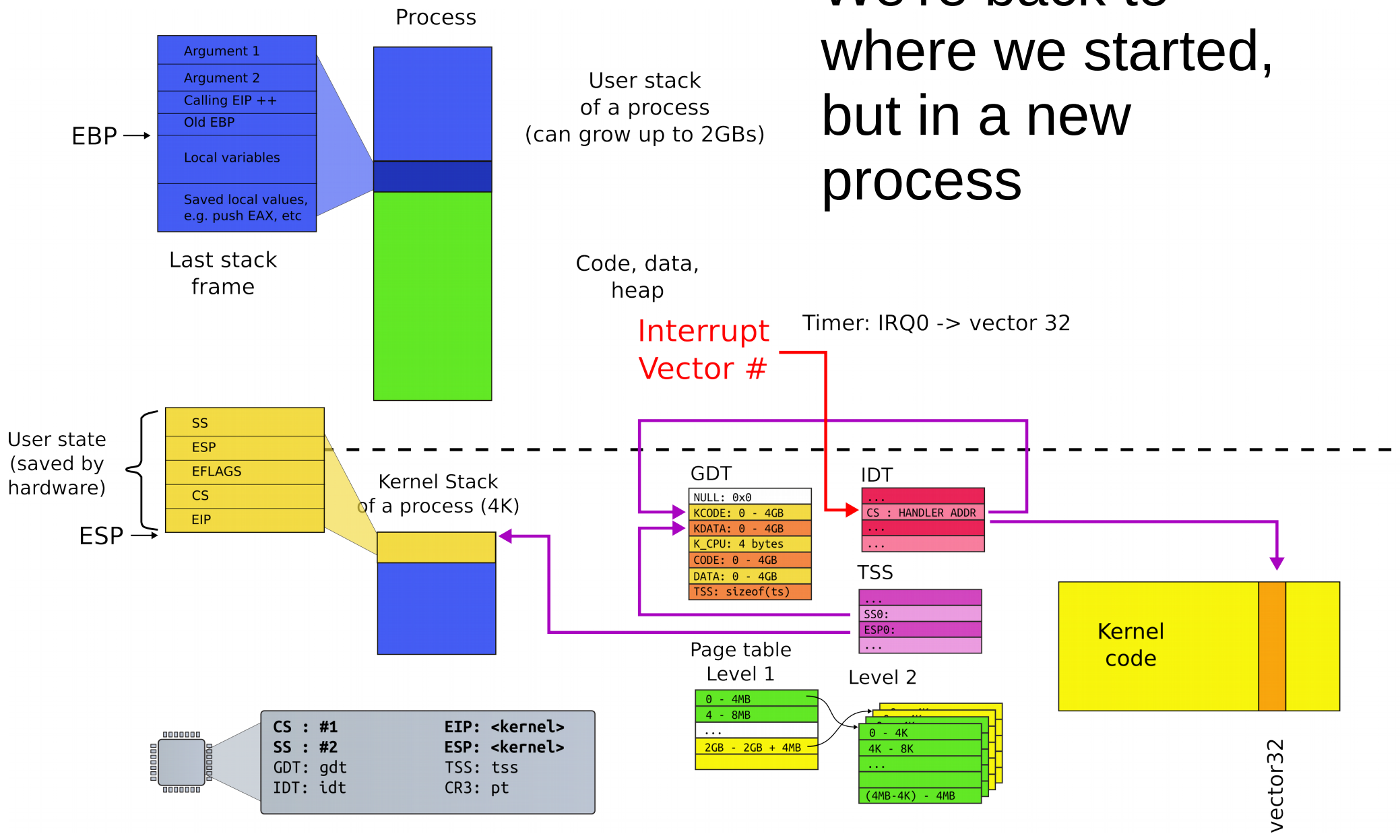


alltraps(): exiting

```
3004 alltraps:
...
3020 # Call trap(tf), where tf=%esp
3021 pushl %esp
3022 call trap
3023 addl $4, %esp
3024
3025 # Return falls through to trapret...
3026 .globl trapret
3027 trapret:
3028 popal
3029 popl %gs
3030 popl %fs
3031 popl %es
3032 popl %ds
3033 addl $0x8, %esp # trapno and errcode
3034 iret
```

- Restore all registers
- Exit into user
 - iret

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



Summary

- We switch between processes now

Creating the first process

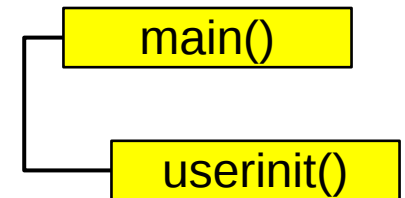
```
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
...
1323     seginit(); // segment descriptors
...
1330     tvinit(); // trap vectors
...
1338     userinit(); // first user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

main()

Userinit() – create first process

- Allocate process structure
 - Information about the process


```
2502 userinit(void)
2503 {
2504     struct proc *p;
2505     extern char _binary_initcode_start[],
                _binary_initcode_size[];
...
2509     p = allocproc();
2510     initproc = p;
2511     if((p->pgdir = setupkvm()) == 0)
2512         panic("userinit: out of memory?");
2513     inituvm(p->pgdir, _binary_initcode_start,
            (int)_binary_initcode_size);
2514     p->sz = PGSIZE;
2515     memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```

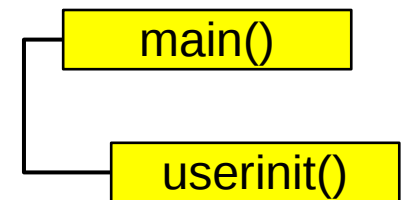


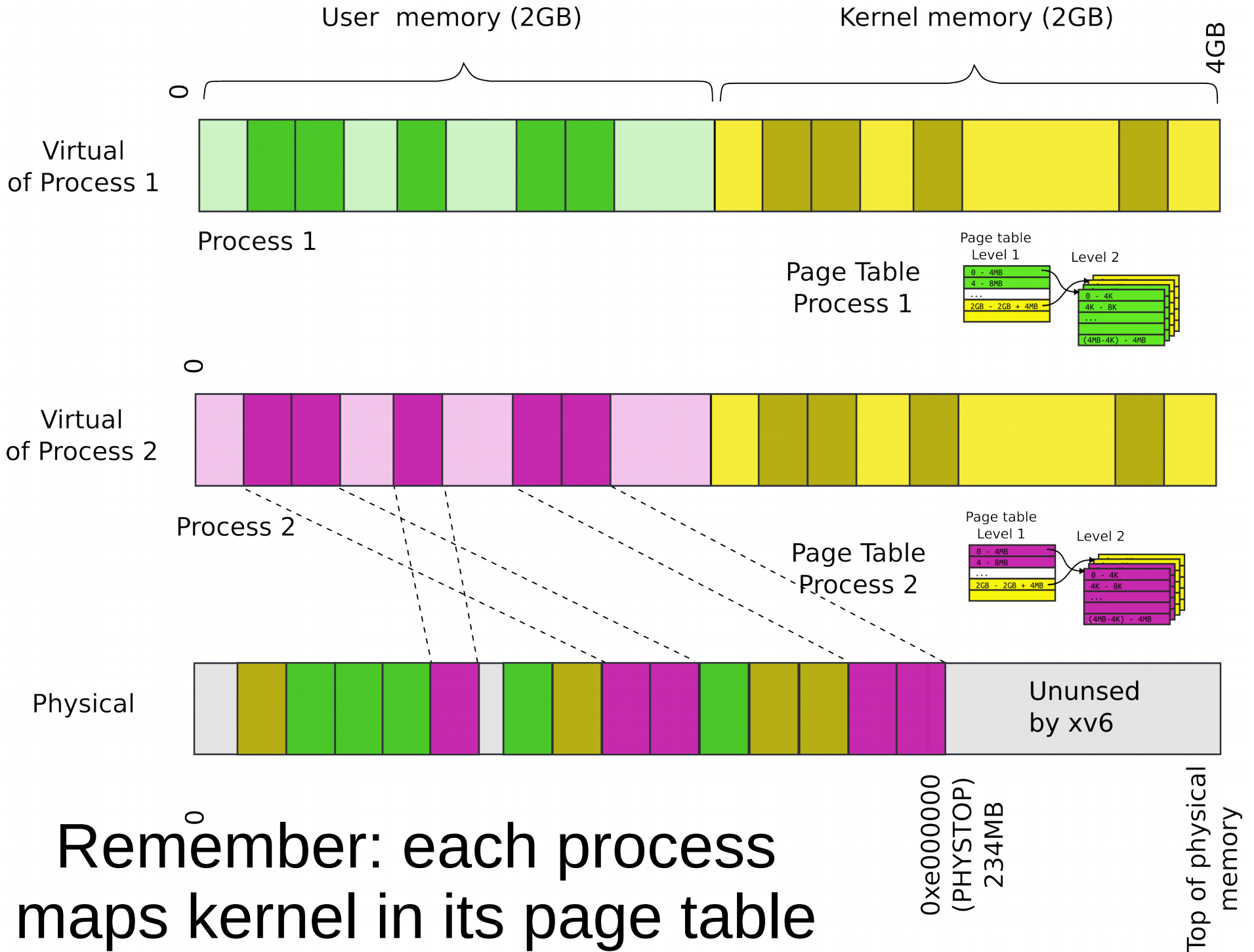
```
2103 struct proc {
2104     uint sz; // Size of process memory (bytes)
2105     pde_t* pgdir; // Page table
2106     char *kstack; // Bottom of kernel stack for this process
2107     enum procstate state; // Process state
2108     volatile int pid; // Process ID
2109     struct proc *parent; // Parent process
2110     struct trapframe *tf; // Trap frame for current syscall
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 };
```

Userinit() – create first process

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

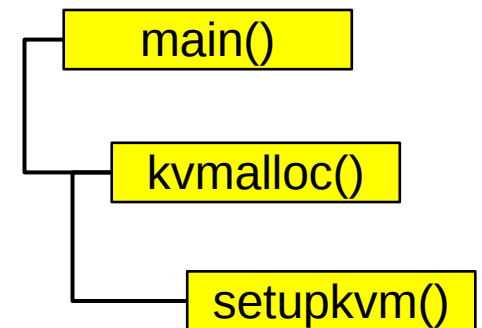
```
2502 userinit(void)
2503 {
2504     struct proc *p;
2505     extern char _binary_initcode_start[],
                _binary_initcode_size[];
...
2509     p = allocproc();
2510     initproc = p;
2511     if((p->pgdir = setupkvm()) == 0)
2512         panic("userinit: out of memory?");
2513     inituvm(p->pgdir, _binary_initcode_start,
            (int)_binary_initcode_size);
2514     p->sz = PGSIZE;
2515     memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```





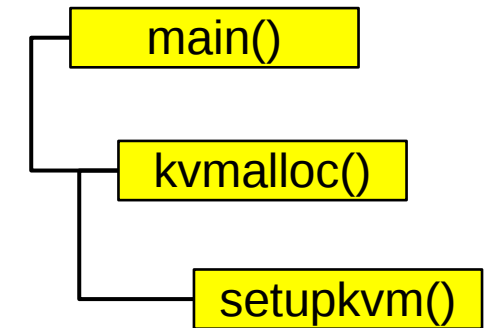
Recap: Allocate page table directory

```
1836 pde_t*
1837 setupkvm(void)
1838 {
1839     pde_t *pgdir;
1840     struct kmap *k;
1841
1842     if((pgdir = (pde_t*)kalloc()) == 0)
1843         return 0;
1844     memset(pgdir, 0, PGSIZE);
1845     ...
1847     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                     (uint)k->phys_start, k->perm) < 0)
1850             return 0;
1851     return pgdir;
1852 }
```



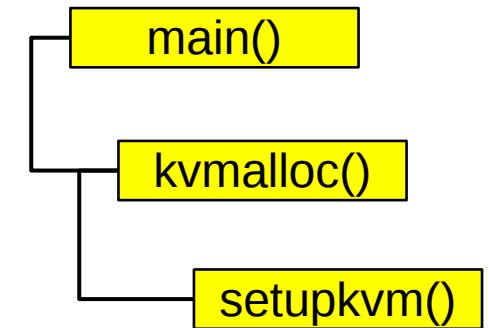
Recap: Iterate in a loop: remap physical pages

```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839     pde_t *pgdir;
1840     struct kmap *k;
1841
1842     if((pgdir = (pde_t*)kalloc()) == 0)
1843         return 0;
1844     memset(pgdir, 0, PGSIZE);
1845     ...
1847     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850             return 0;
1851     return pgdir;
1852 }
```

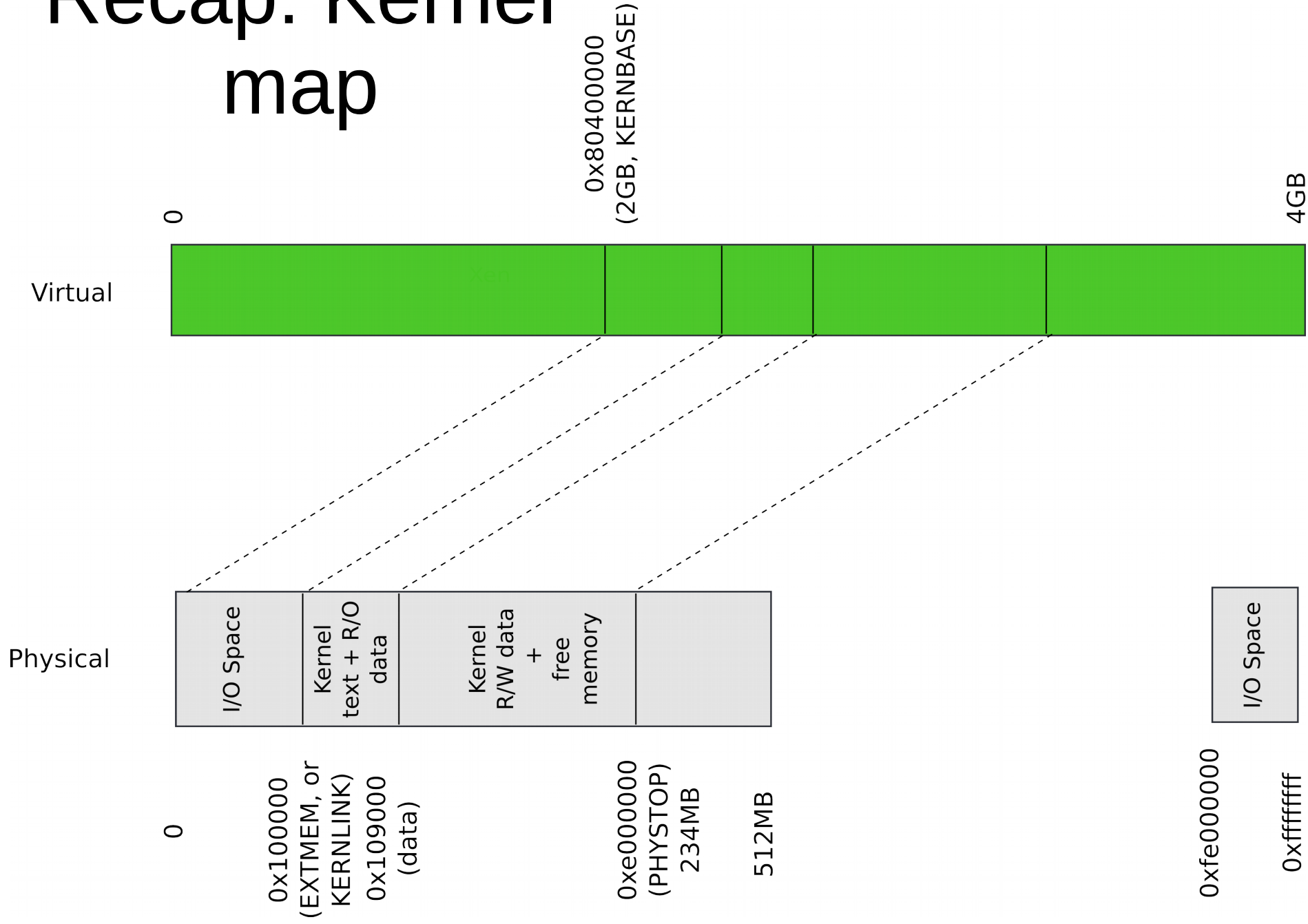


Recap: Iterate in a loop: remap physical pages

```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839     pde_t *pgdir;
1840     struct kmap *k;
1841
1842     if((pgdir = (pde_t*)kalloc()) == 0)
1843         return 0;
1844     memset(pgdir, 0, PGSIZE);
1845     ...
1847     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                     (uint)k->phys_start, k->perm) < 0)
1850             return 0;
1851     return pgdir;
1852 }
```



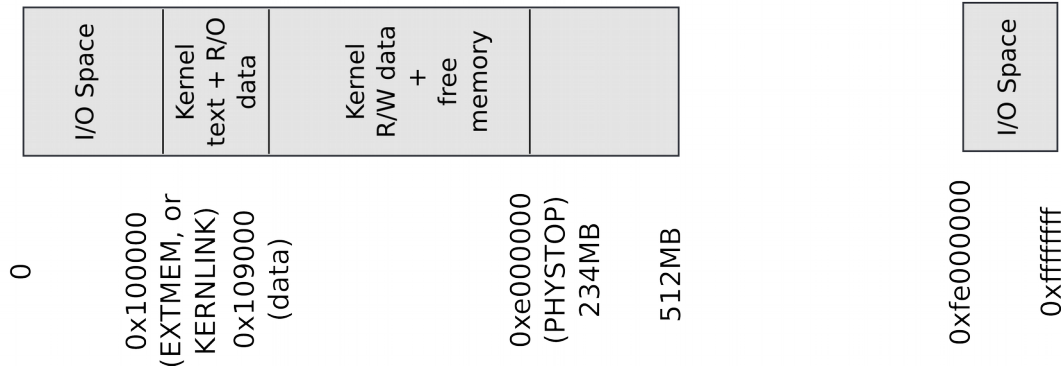
Recap: Kernel map



Recap: Kmap – kernel map

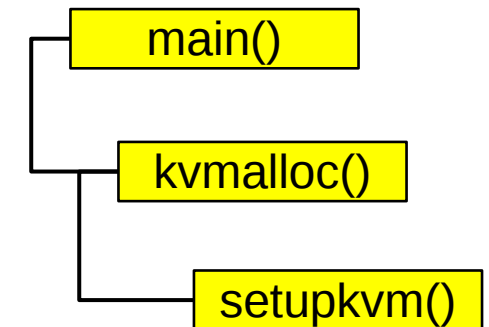
```
1823 static struct kmap {
1824     void *virt;
1825     uint phys_start;
1826     uint phys_end;
1827     int perm;
1828 } kmap[] = {
1829     { (void*)KERNBASE, 0, EXTMEM, PTE_W}, // I/O space
1830     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, //text+rodata
1831     { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern
data+memory
1832     { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
1833 };
```

Physical



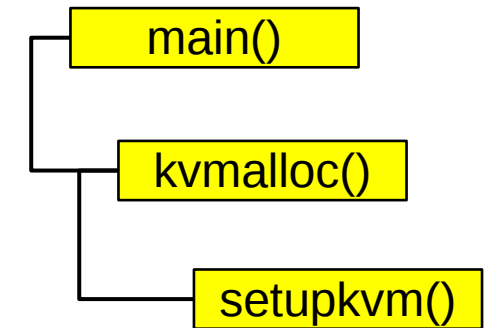
Recap: Iterate in a loop: remap physical pages

```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839     pde_t *pgdir;
1840     struct kmap *k;
1841
1842     if((pgdir = (pde_t*)kalloc()) == 0)
1843         return 0;
1844     memset(pgdir, 0, PGSIZE);
1845     ...
1847     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                    (uint)k->phys_start, k->perm) < 0)
1850             return 0;
1851     return pgdir;
1852 }
```



```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839     pde_t *pgdir;
1840     struct kmap *k;
1841
1842     if((pgdir = (pde_t*)kalloc()) == 0)
1843         return 0;
1844     memset(pgdir, 0, PGSIZE);
1845     ...
1847     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                    (uint)k->phys_start, k->perm) < 0)
1850             return 0;
1851     return pgdir;
1852 }
```

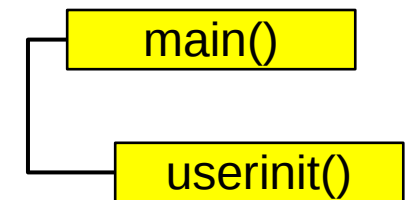
Recap: Remap physical pages



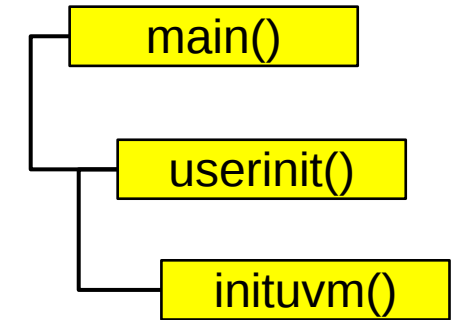
Userinit() – create first process

- 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**

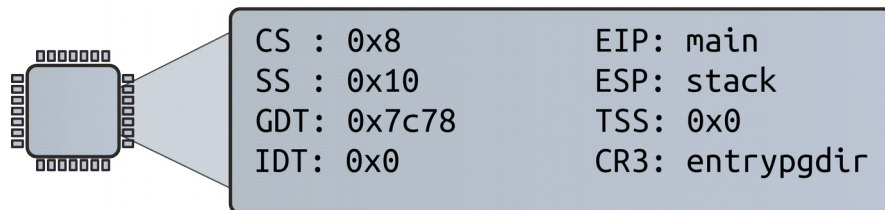
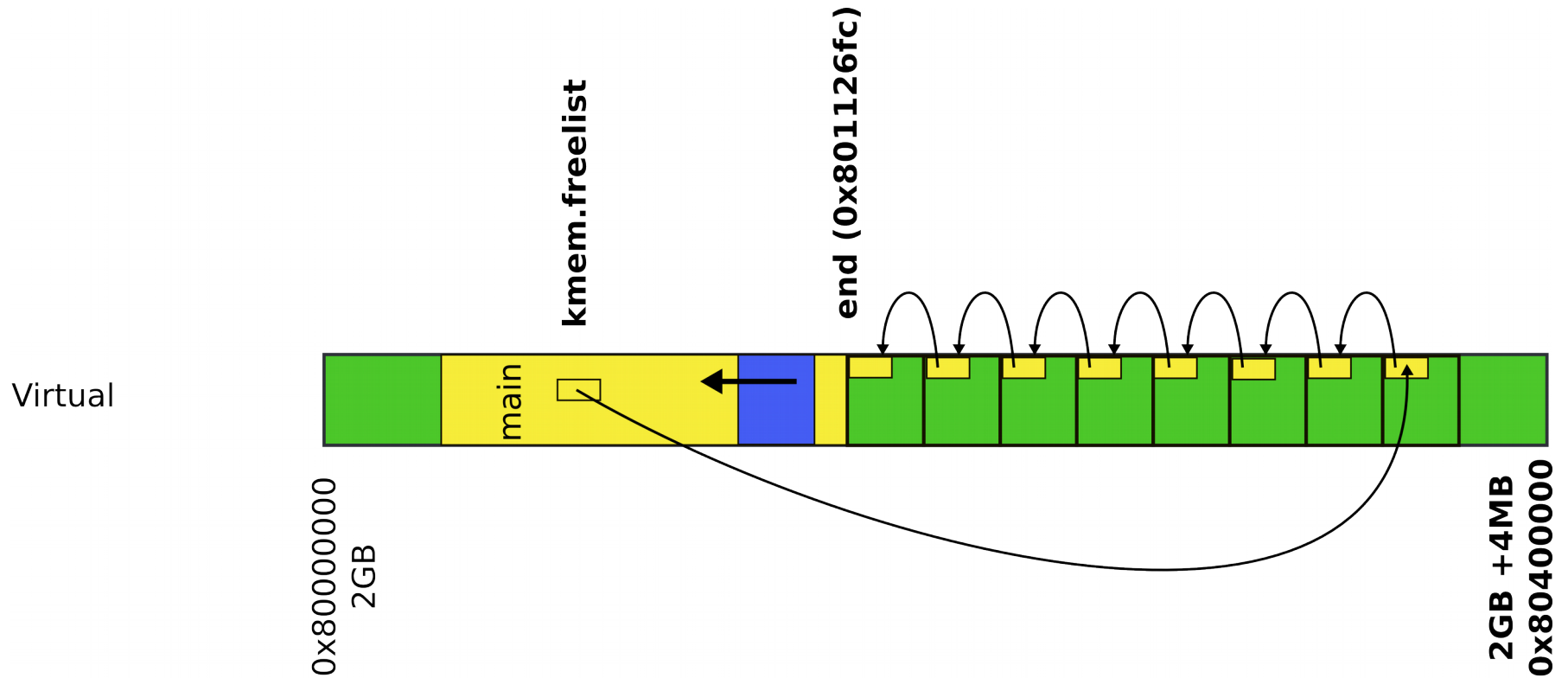
```
2502 userinit(void)
2503 {
2504     struct proc *p;
2505     extern char _binary_initcode_start[],
                _binary_initcode_size[];
...
2509     p = allocproc();
2510     initproc = p;
2511     if((p->pgdir = setupkvm()) == 0)
2512         panic("userinit: out of memory?");
2513     inituvm(p->pgdir, _binary_initcode_start,
            (int)_binary_initcode_size);
2514     p->sz = PGSIZE;
2515     memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```



```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905     char *mem;
1906
1907     if(sz >= PGSIZE)
1908         panic("inituvm: more than a page");
1909     mem = kalloc();
1910     memset(mem, 0, PGSIZE);
1911     mappages(pgdir, 0, PGSIZE, V2P(mem),
1912             PTE_W|PTE_U);
1913     memmove(mem, init, sz);
1914 }
```

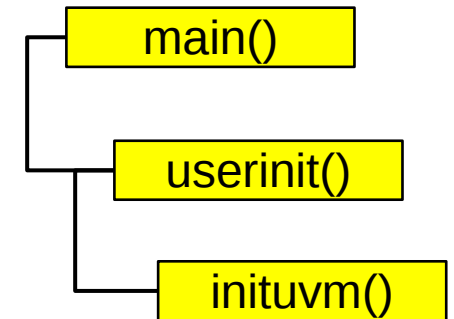


Recap: kalloc() – allocate page

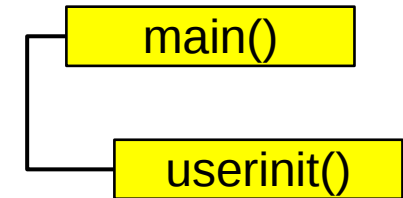


Protected Mode

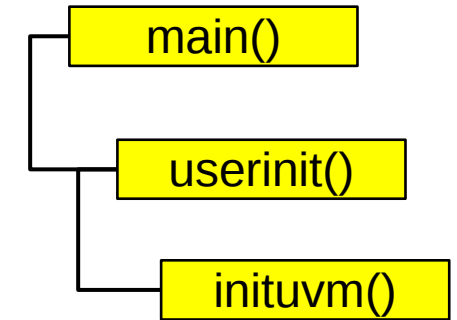

```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905     char *mem;
1906
1907     if(sz >= PGSIZE)
1908         panic("inituvm: more than a page");
1909     mem = kalloc();
1910     memset(mem, 0, PGSIZE);
1911     mappages(pgdir, 0, PGSIZE, V2P(mem),
1912             PTE_W|PTE_U);
1912     memmove(mem, init, sz);
1913 }
```



```
2502 userinit(void)
2503 {
2504     struct proc *p;
2505     extern char _binary_initcode_start[],
                _binary_initcode_size[];
...
2509     p = allocproc();
2510     initproc = p;
2511     if((p->pgdir = setupkvm()) == 0)
2512         panic("userinit: out of memory?");
2513     inituvm(p->pgdir, _binary_initcode_start,
                (int)_binary_initcode_size);
2514     p->sz = PGSIZE;
2515     memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```



```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905     char *mem;
1906
1907     if(sz >= PGSIZE)
1908         panic("inituvm: more than a page");
1909     mem = kalloc();
1910     memset(mem, 0, PGSIZE);
1911     mappages(pgdir, 0, PGSIZE, V2P(mem),
1912             PTE_W|PTE_U);
1912     memmove(mem, init, sz);
1913 }
```



```
8409 start:
8410     pushl $argv
8411     pushl $init
8412     pushl $0 // where caller pc would be
8413     movl $SYS_exec, %eax
8414     int $T_SYSCALL
8415
...
8422 # char init[] = "/init\0";
8423 init:
8424     .string "/init\0"
8425
8426 # char *argv[] = { init, 0 };
8427 .p2align 2
8428 argv:
8429     .long init
8430     .long 0
```

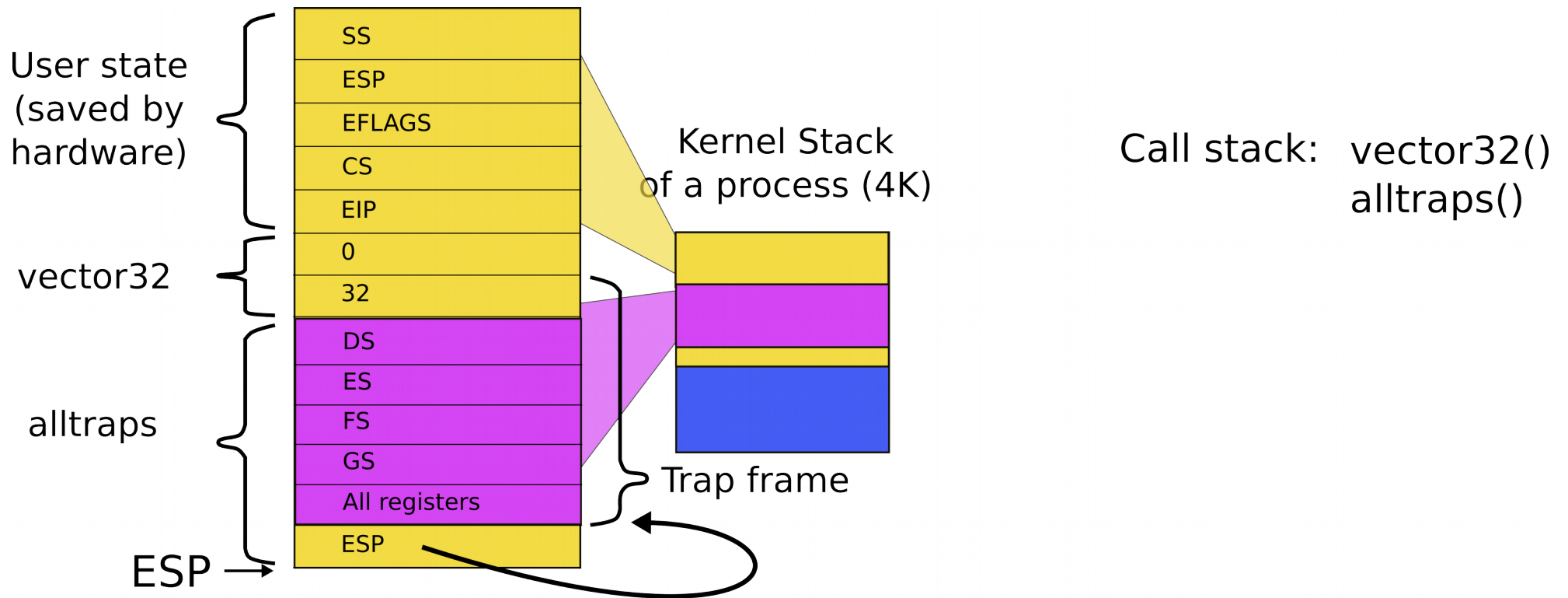
initcode.S: call
exec("/init", argv);

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



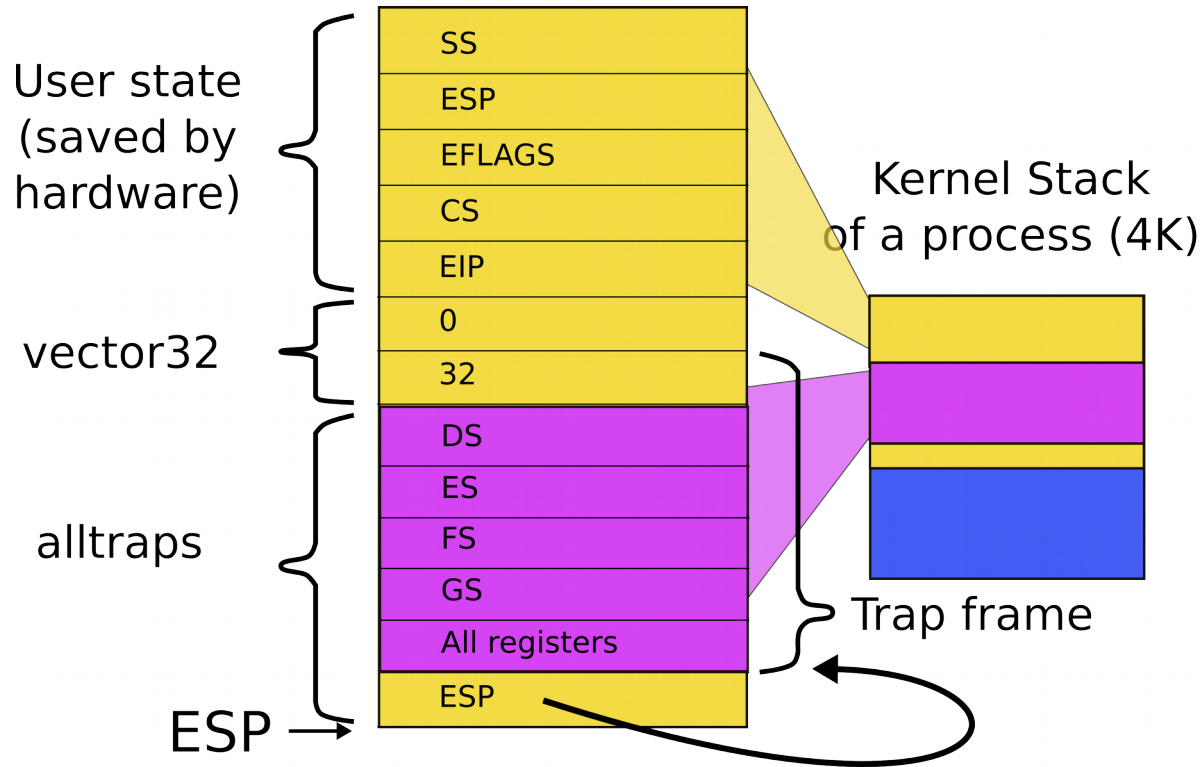
```
2103 struct proc {
2104     uint sz; // Size of process memory (bytes)
2105     pde_t* pgdir; // Page table
2106     char *kstack; // Bottom of kernel stack for this process
2107     enum procstate state; // Process state
2108     volatile int pid; // Process ID
2109     struct proc *parent; // Parent 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 };
```

```
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 }
```

Trap frame is on the
kernel stack of the process


```
2502 userinit(void)
2503 {
...
2513     inituvm(p->pgdir, _binary_initcode_start,
             (int)_binary_initcode_size);
2514     p->sz = PGSIZE;
2515     memset(p->tf, 0, sizeof(*p->tf));
2516     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2517     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2518     p->tf->es = p->tf->ds;
2519     p->tf->ss = p->tf->ds;
2520     p->tf->eflags = FL_IF;
2521     p->tf->esp = PGSIZE;
2522     p->tf->eip = 0; // beginning of initcode.S
...
2530 }
```

Kernel stack after interrupt/syscall

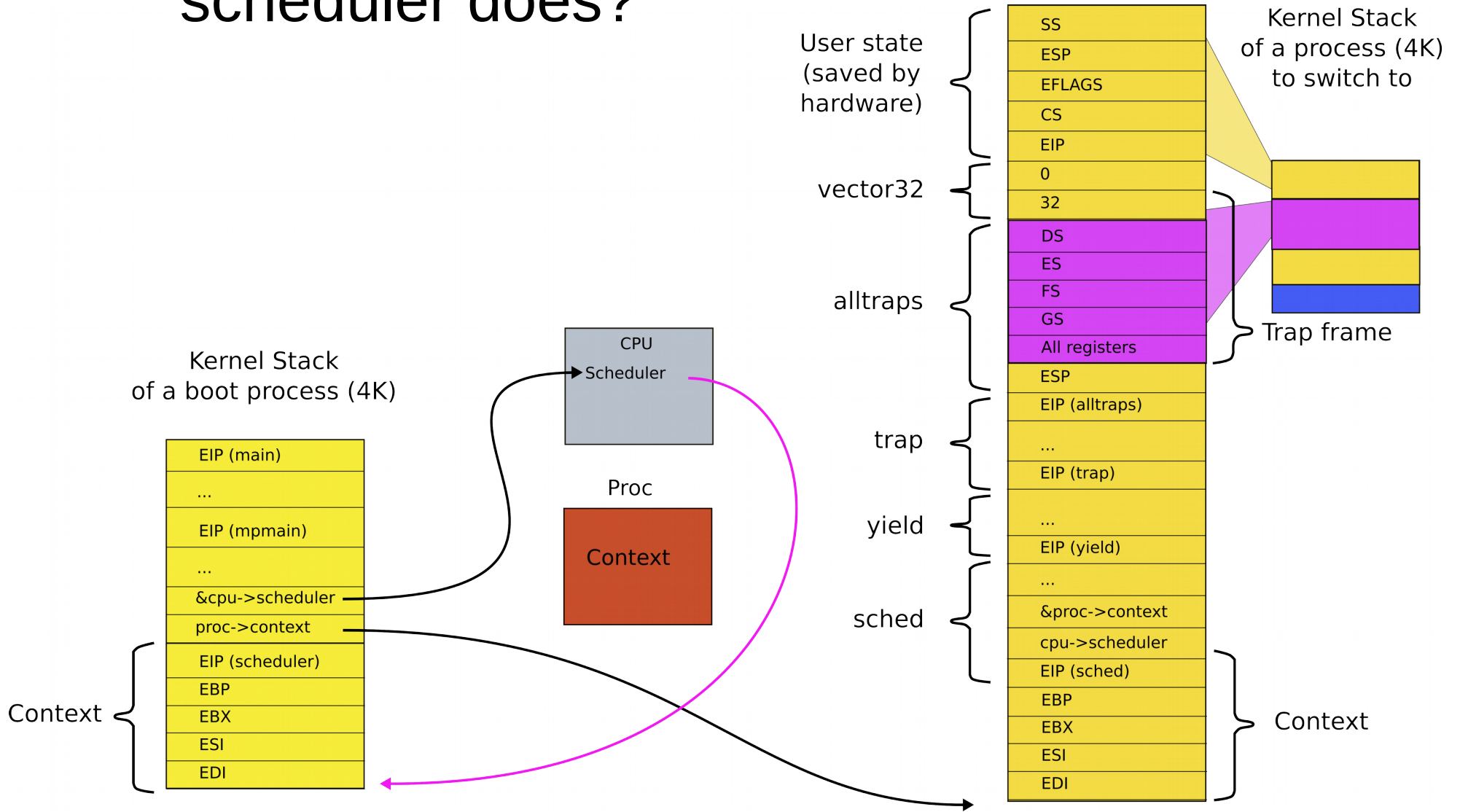


Call stack: `vector32()`
`alltraps()`

```
2502 userinit(void)
2503 {
...
2515  memset(p->tf, 0, sizeof(*p->tf));
2516  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2517  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2518  p->tf->es = p->tf->ds;
2519  p->tf->ss = p->tf->ds;
2520  p->tf->eflags = FL_IF;
2521  p->tf->esp = PGSIZE;
2522  p->tf->eip = 0; // beginning of initcode.S
2523
2524  safestrcpy(p->name, "initcode", sizeof(p->name));
2525  p->cwd = namei("/");
2526
2527  p->state = RUNNABLE;
...
2530 }
```

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

Remember what scheduler does?



Trap frame is on the kernel stack of the process

```
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 }
```

forkret(): just returns

```
2788 forkret(void)
```

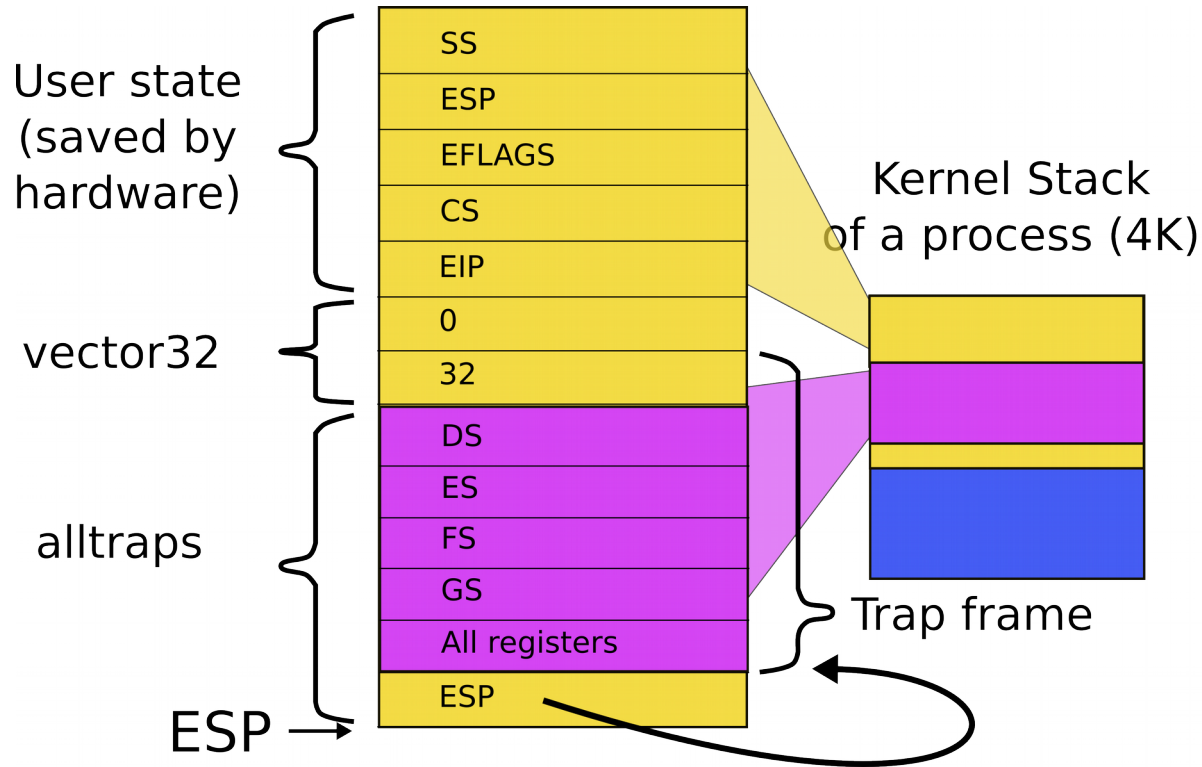
```
2789 {
```

```
...
```

```
2803 // Return to "caller", actually trapret (see allocproc).
```

```
2804 }
```

Kernel stack after interrupt/syscall



Call stack: vector32()
alltraps()


```
3276 .globl trapret
```

trapret(): just returns

```
3277 trapret:
```

```
3278     popal
```

```
3279     popl %gs
```

```
3280     popl %fs
```

```
3281     popl %es
```

```
3282     popl %ds
```

```
3283     addl $0x8, %esp # trapno and  
                               errcode
```

```
3284     iret
```

```
8510 main(void)
8511 {
...
8514     if(open("console", O_RDWR) < 0){
8515         mknod("console", 1, 1);
8516         open("console", O_RDWR);
8517     }
8518     dup(0); // stdout
8519     dup(0); // stderr
8520
8521     for(;;){
8522         printf(1, "init: starting sh\n");
8523         pid = fork();
8524         if(pid < 0){
8525             printf(1, "init: fork failed\n");
8526             exit();
8527         }
8528         if(pid == 0){
8529             exec("sh", argv);
8530             printf(1, "init: exec sh failed\n");
8531             exit();
8532         }
8533         while((wpid=wait()) >= 0 && wpid != pid)
8534             printf(1, "zombie!\n");
8535     }
8536 }
```

- 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

Thank you