CS5460/6460: Operating Systems

Lecture 9: First process

Anton Burtsev
January, 2014

Recap from last time

- Jumped to main()
- Allocated physical memory allocator
  - kalloc() – allocates a page of virtual memory
- Initialized kernel page tables
- Initialized segment registers
- Initialized interrupt descriptor table

Starting other CPUs

```
1217 main(void)
1218 {
...
1237   startothers(); // start other processors
1238   kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
1239   userinit(); // first user process
1240   // Finish setting up this processor
1241   mpmain();
1242 }
```

Starting other CPUs

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

```
1274 startothers(void)
1275 {
1284   code = p2v(0x7000);
1285   memmove(code, _binary_entryother_start,
               (uint)_binary_entryother_size);
1286
1287   for(c = cpus; c < cpus+ncpu; c++){
1288     if(c == cpus+cpunum()) // We've started already.
1289       continue;
...
1294     stack = kalloc();
1295     *(void**)(code-4) = stack + KSTACKSIZE;
1296     *(void**)(code-8) = mpenter;
1297     *(int**)(code-12) = (void *) v2p(entrypgdir);
1298
1299     lapicstartap(c->id, v2p(code));
```

entryother.S

```
1123 .code16
1124 .globl start
1125 start:
1126 cli
1127
1128 xorw %ax,%ax
1129 movw %ax,%ds
1130 movw %ax,%es
1131 movw %ax,%ss
1132
```

- Disable interrupts
- Init segments with 0

entryother.S

```
1133 lgdt gdtdesc
1134 movl %cr0, %eax
1135 orl $CR0_PE, %eax
1136 movl %eax, %cr0
1150 ljmpl $(SEG_KCODE<<3), $(start32)
1151
1152 .code32
1153 start32:
1154 movw $(SEG_KDATA<<3), %ax
1155 movw %ax, %ds
1156 movw %ax, %es
1157 movw %ax, %ss
1158 movw $0, %ax
1159 movw %ax, %fs
1160 movw %ax, %gs
```

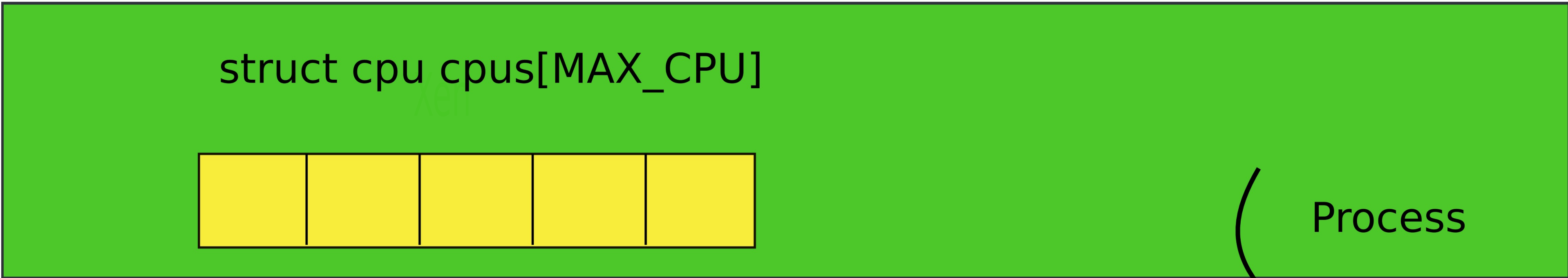
- Load GDT
- Switch to 32bit mode
  - Long jump to start32
- Load segments

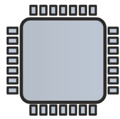
```
1162 # Turn on page size extension for 4Mbyte pages
1163 movl %cr4, %eax
1164 orl $(CR4_PSE), %eax
1165 movl %eax, %cr4
1166 # Use enterpgdir as our initial page table
1167 movl (start-12), %eax
1168 movl %eax, %cr3
1169 # Turn on paging.
1170 movl %cr0, %eax
1171 orl $(CR0_PE|CR0_PG|CR0_WP), %eax
1172 movl %eax, %cr0
1173
1174 # Switch to the stack allocated by startothers()
1175 movl (start-4), %esp
1176 # Call mpenter()
1177 call *(start-8)
```

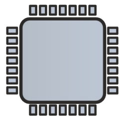
entryother.S

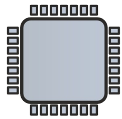
```
1251 static void
1252 mpenter(void)
1253 {
1254     switchkvm();
1255     seginit();
1256     lapicinit();
1257     mpmain();
1258 }
```

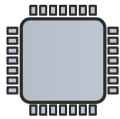
```
1616 seginit(void)
1617 {
1618   struct cpu *c;
...
1624   c = &cpus[cpunum()];
1625   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
1626   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
1627   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0x8000000, DPL_USER);
1628   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
1629
1630   // Map cpu, and curproc
1631   c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
1632
1633   lgdt(c->gdt, sizeof(c->gdt));
1634   loadgs(SEG_KCPU << 3);
1635
1636   // Initialize cpu-local storage.
1637   cpu = c;
1638   proc = 0;
1639 }
```

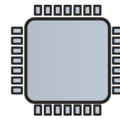

struct cpu cpus[MAX_CPU]
Process





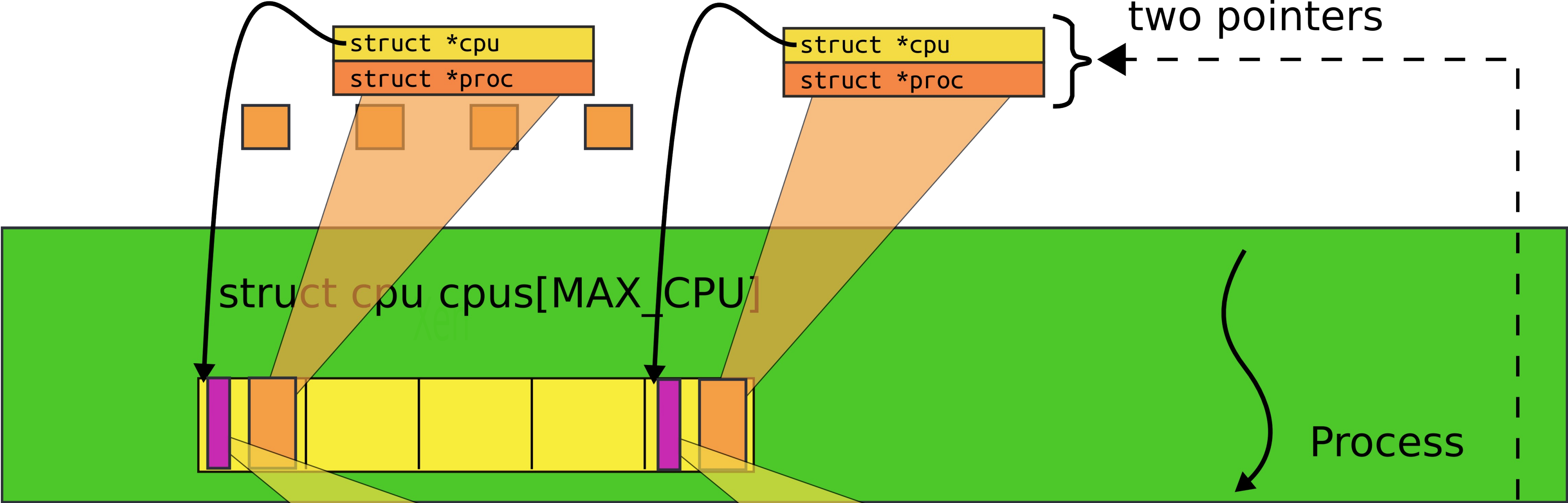


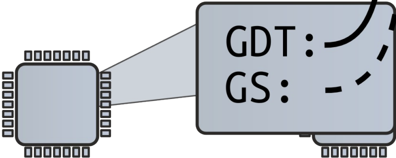


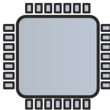


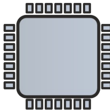
APIC
lapic[ID]

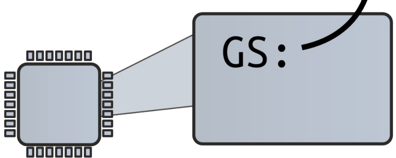
Tiny segment (8 bytes),
two pointers

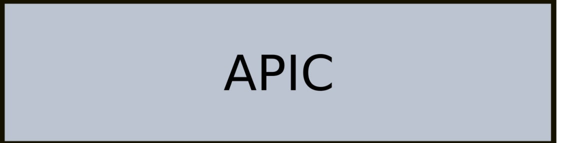












```
1260 // Common CPU setup code.
1261 static void
1262 mpmain(void)
1263 {
1264   cprintf("cpu%d: starting\n", cpu->id);
1265   idtinit(); // load idt register
1266   xchg(&cpu->started, 1);
1267   scheduler(); // start running processes
1268 }
```

First process

```
1216 int
1217 main(void) {
...
1223   seginit(); // set up segments
1238   kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
1239   userinit(); // first user process
1240
1241   mpmain();
1242 }
```

Userinit() – create first process

- Allocate process structure
  - Information about 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 for current syscall
2111   struct context *context; // swtch() here to run process
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**

```
2251 void
2252 userinit(void)
2253 {
2254   struct proc *p;
2255   extern char _binary_initcode_start[],
                _binary_initcode_size[];
2256
2257   p = allocproc();
2258   initproc = p;
2259   if((p->pgdir = setupkvm()) == 0)
2260     panic("userinit: out of memory?");
```

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**

```
1802 void
1803 inituvm(pde_t *pgdir, char *init, uint sz)
1804 {
1805   char *mem;
1806
1807   if(sz >= PGSIZE)
1808     panic("inituvm: more than a page");
1809   mem = kalloc();
1810   memset(mem, 0, PGSIZE);
1811   mappages(pgdir,0,PGSIZE,v2p(mem), PTE_W|PTE_U);
1812   memmove(mem, init, sz);
1813 }
```

Recap: kalloc() – allocate page

kmem.freelist

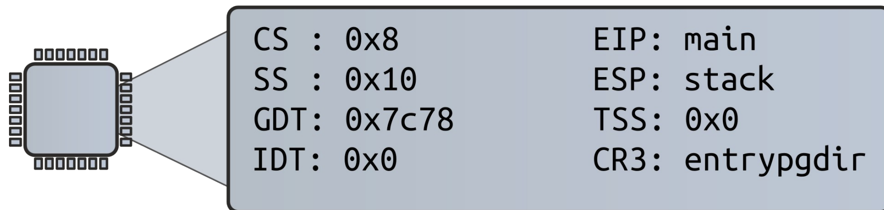
end (0x801126fc)

Virtual

main

0x80000000
2GB

2GB +4MB
0x80400000



Protected Mode

```
1802 void
1803 inituvm(pde_t *pgdir, char *init, uint sz)
1804 {
1805   char *mem;
1806
1807   if(sz >= PGSIZE)
1808     panic("inituvm: more than a page");
1809   mem = kalloc();
1810   memset(mem, 0, PGSIZE);
1811   mappages(pgdir,0,PGSIZE,v2p(mem), PTE_W|PTE_U);
1812   memmove(mem, init, sz);
1813 }
```

Recap: mappages()

```
1679 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1680 {
1681   char *a, *last;
1682   pte_t *pte;
1683
1684   a = (char*)PGROUNDDOWN((uint)va);
1685   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1686   for(;;){
1687     if((pte = walkpgdir(pgdir, a, 1)) == 0)
1688       return -1;
1689     if(*pte & PTE_P)
1690       panic("remap");
1691     *pte = pa | perm | PTE_P;
1692     if(a == last)
1693       break;
1694     a += PGSIZE;
1695     pa += PGSIZE;
1696   }
1697   return 0;
1698 }
```

Recap: walkpgdir()

```
1654 walkpgdir(pde_t *pgdir, const void *va, int alloc)
1655 {
1656   pde_t *pde;
1657   pte_t *pgtab;
1658
1659   pde = &pgdir[PDX(va)];
1660   if(*pde & PTE_P){
1661     pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
1662   } else {
1663     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1664       return 0;
1665     // Make sure all those PTE_P bits are zero.
1666     memset(pgtab, 0, PGSIZE);
…
1670     *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
1671   }
1672   return &pgtab[PTX(va)];
1673 }
```

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”**

```
...
2261   inituvm(p->pgdir, _binary_initcode_start,
              (int)_binary_initcode_size);
2262   p->sz = PGSIZE;
2263   memset(p->tf, 0, sizeof(*p->tf));
2264   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2265   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2266   p->tf->es = p->tf->ds;
2267   p->tf->ss = p->tf->ds;
2268   p->tf->eflags = FL_IF;
2269   p->tf->esp = PGSIZE;
2270   p->tf->eip = 0; // beginning of initcode.S
2271
2272   safestrcpy(p->name, "initcode", sizeof(p->name));
2273   p->cwd = namei("/");
2274
2275   p->state = RUNNABLE;
2276 }
```

scheduler()

Per−CPU process scheduler

- Each CPU calls scheduler() after setting itself up
- Scheduler never returns
- It loops, doing:
  - Choose a process to run
  - Switch to start running that process
  - Eventually that process transfers control
    - Via switching back to the scheduler

```
2458 scheduler(void)
2459 {
2462   for(;;){
2464     sti();
2467     acquire(&ptable.lock);
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     }
2485     release(&ptable.lock);
2487   }
2488 }
```

switchuvm()

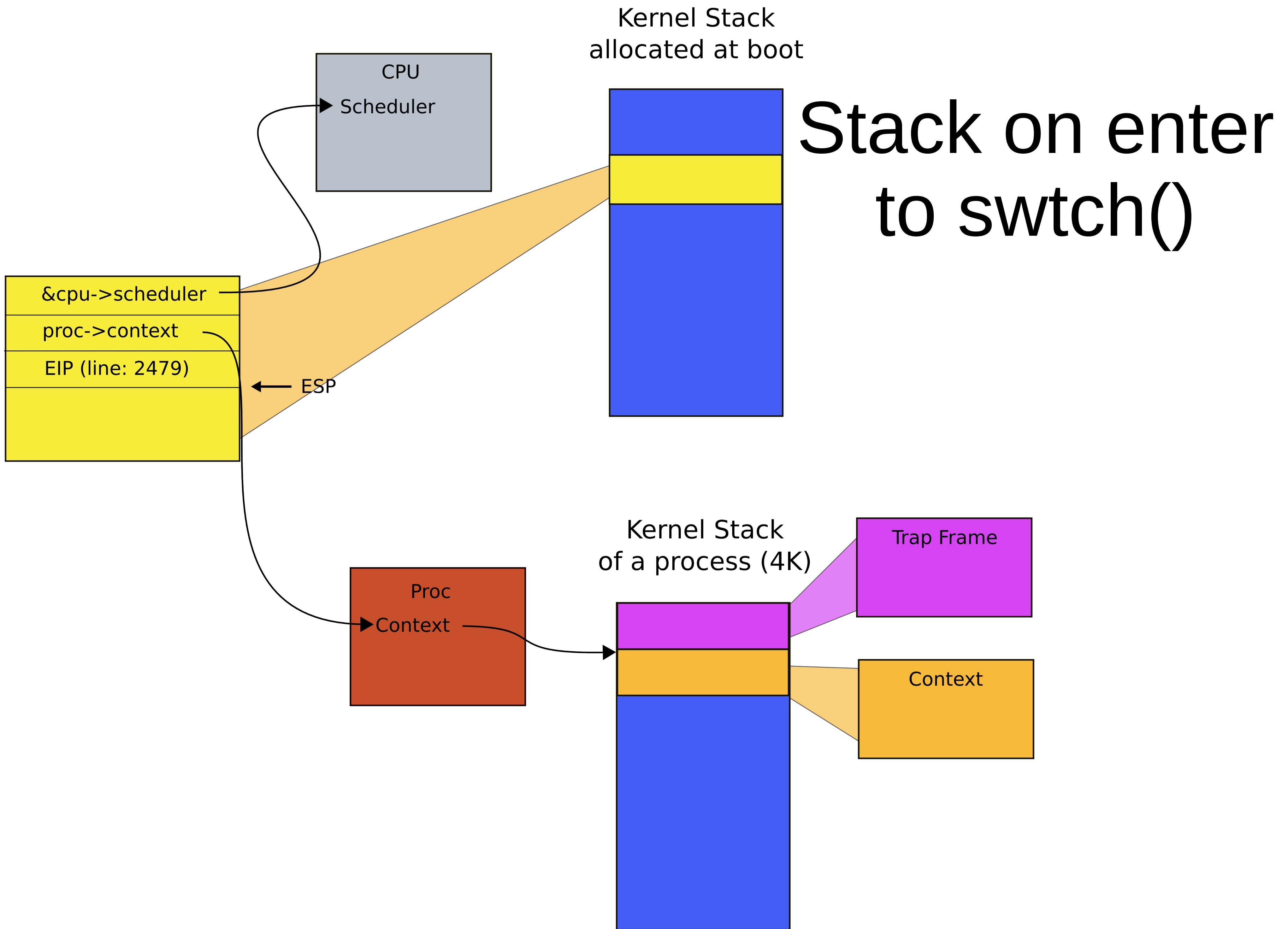
- Switch TSS
- Switch page table

```
1772 void
1773 switchuvm(struct proc *p)
1774 {
1775   pushcli();
1776   cpu->gdt[SEG_TSS]=SEG16(STS_T32A,&cpu->ts,sizeof(cpu->ts)-1,0);
1777   cpu->gdt[SEG_TSS].s = 0;
1778   cpu->ts.ss0 = SEG_KDATA << 3;
1779   cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1780   ltr(SEG_TSS << 3);
1781   if(p->pgdir == 0)
1782     panic("switchuvm: no pgdir");
1783   lcr3(v2p(p->pgdir)); // switch to new address space
1784   popcli();
1785 }
```

```
2458 scheduler(void)
2459 {
2462   for(;;){
2464     sti();
2467     acquire(&ptable.lock);
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     }
2485     release(&ptable.lock);
2487   }
2488 }
```

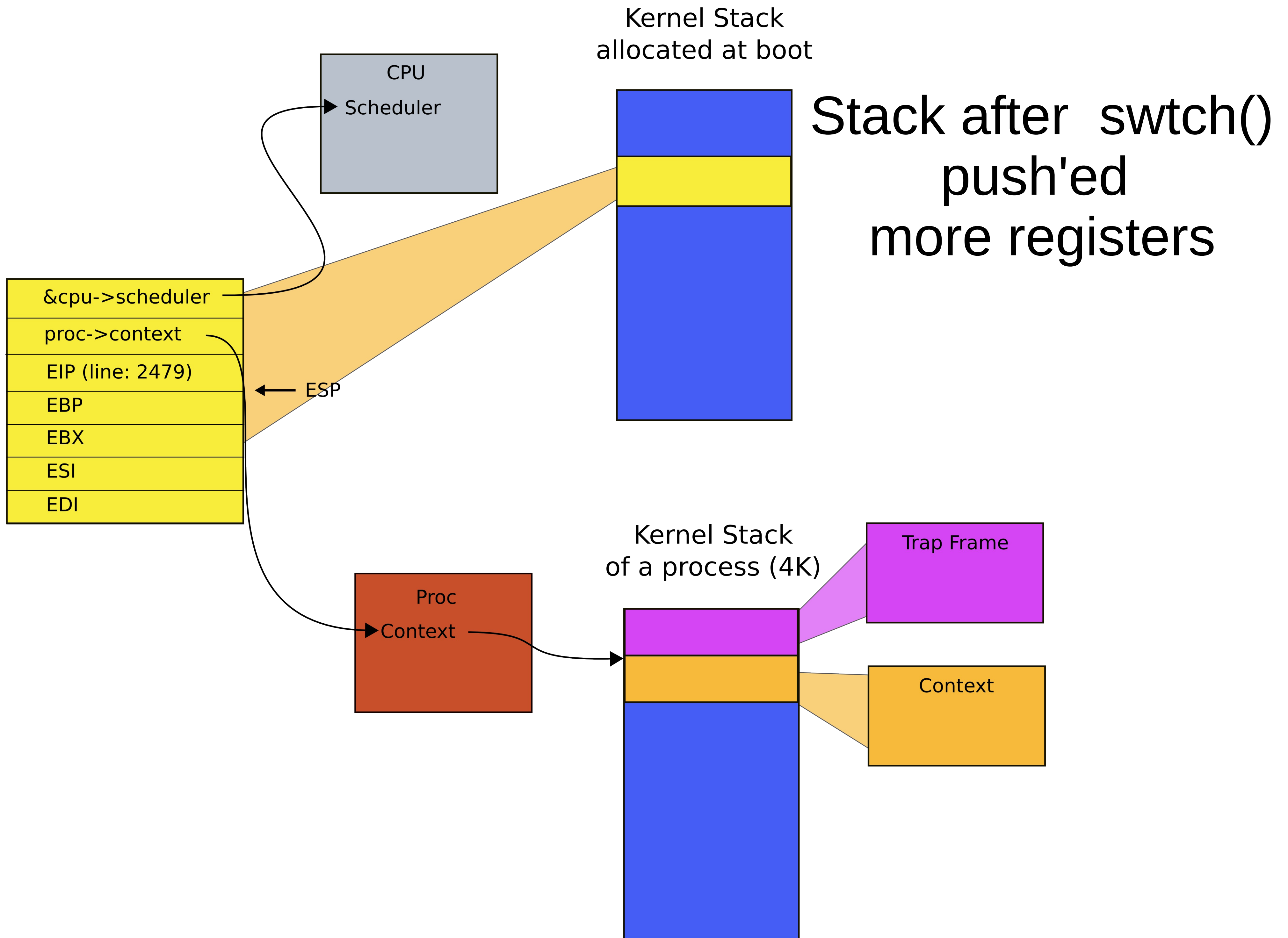
swtch()

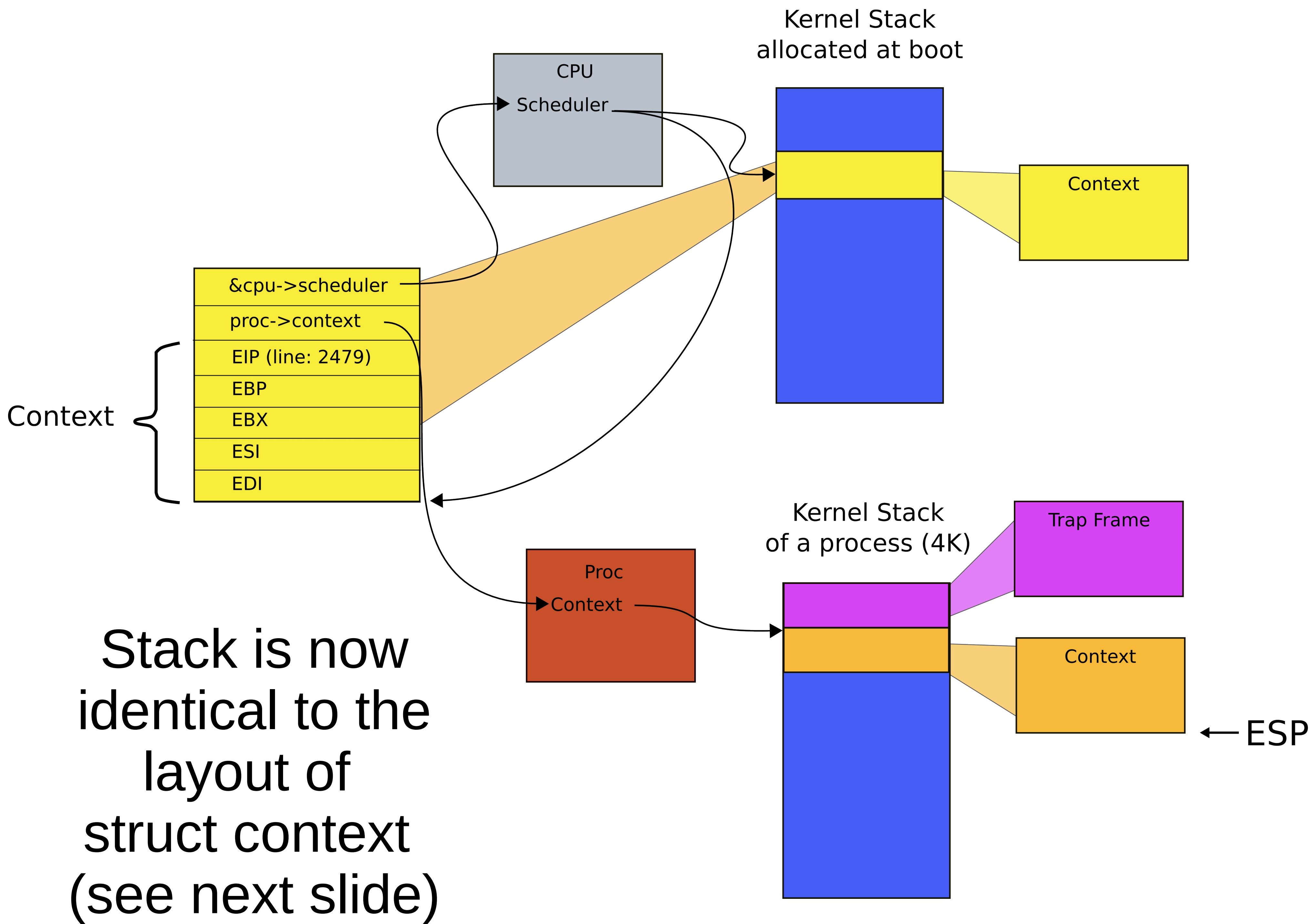
```
2707 .globl swtch
2708 swtch:
2709 movl 4(%esp), %eax
2710 movl 8(%esp), %edx
2711
2712 # Save old callee-save registers
2713 pushl %ebp
2714 pushl %ebx
2715 pushl %esi
2716 pushl %edi
2717
2718 # Switch stacks
2719 movl %esp, (%eax)
2720 movl %edx, %esp
2721
2722 # Load new callee-save registers
2723 popl %edi
2724 popl %esi
2725 popl %ebx
2726 popl %ebp
2727 ret
```


Stack on enter to swtch()
CPU
Scheduler
Kernel Stack
allocated at boot
&cpu->scheduler
proc->context
EIP (line: 2479)
ESP
Kernel Stack
of a process (4K)
Trap Frame
Proc
Context
Context

swtch()

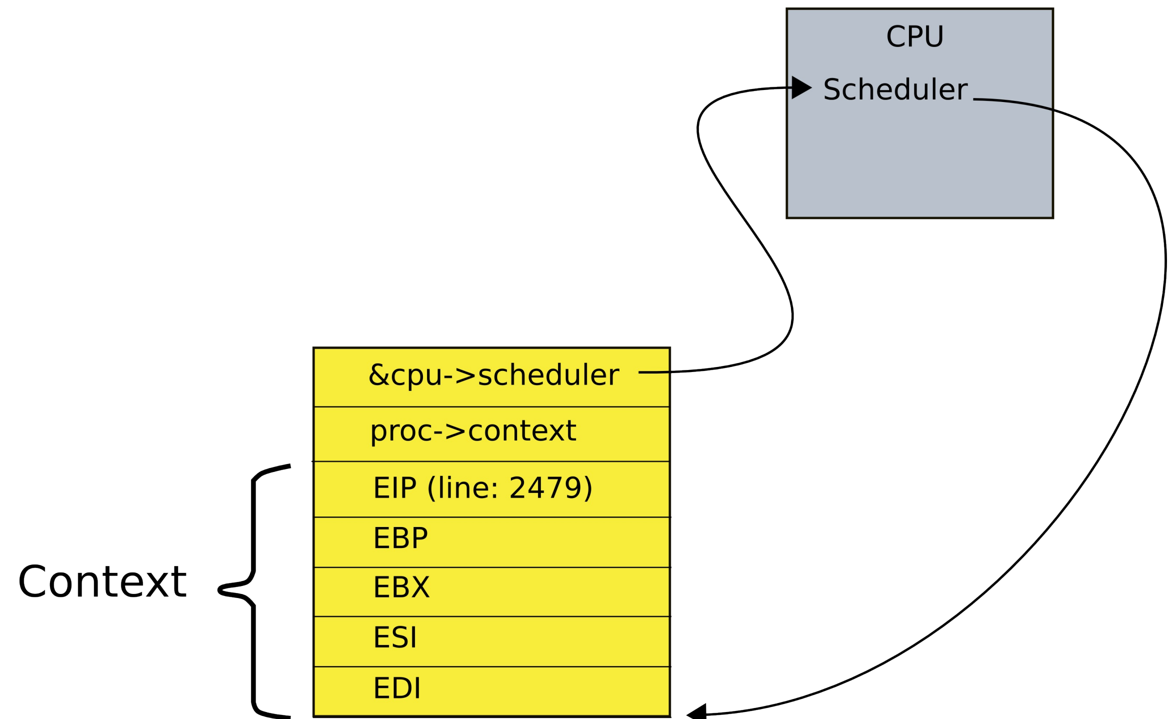
```
2707 .globl swtch
2708 swtch:
2709 movl 4(%esp), %eax
2710 movl 8(%esp), %edx
2711
2712 # Save old callee-save registers
2713 pushl %ebp
2714 pushl %ebx
2715 pushl %esi
2716 pushl %edi
2717
2718 # Switch stacks
2719 movl %esp, (%eax)
2720 movl %edx, %esp
2721
2722 # Load new callee-save registers
2723 popl %edi
2724 popl %esi
2725 popl %ebx
2726 popl %ebp
2727 ret
```


Kernel Stack
allocated at boot
CPU
Scheduler
Stack after swtch()
push'ed
more registers
&cpu->scheduler
proc->context
EIP (line: 2479)
EBP
EBX
ESI
EDI
ESP
Kernel Stack
of a process (4K)
Trap Frame
Proc
Context
Context


Kernel Stack
allocated at boot
CPU
Scheduler
Context
&cpu->scheduler
proc->context
EIP (line: 2479)
EBP
EBX
ESI
EDI
Context
Kernel Stack
of a process (4K)
Trap Frame
Proc
Context
Context
ESP
Stack is now identical to the layout of struct context (see next slide)

Context structure

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



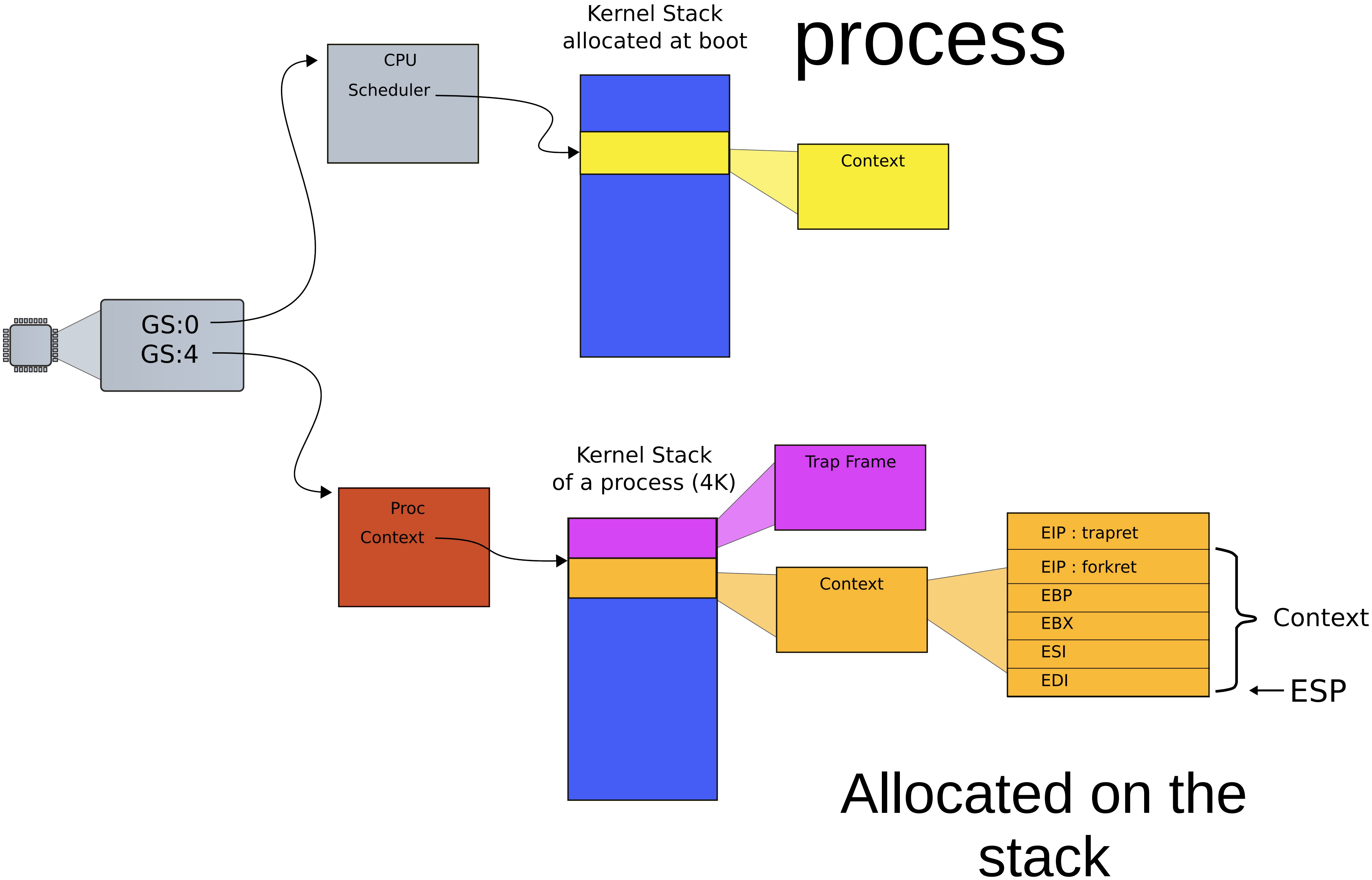
swtch(): switch stacks/contexts

```
2707 .globl swtch
2708 swtch:
2709 movl 4(%esp), %eax
2710 movl 8(%esp), %edx
2711
2712 # Save old callee-save registers
2713 pushl %ebp
2714 pushl %ebx
2715 pushl %esi
2716 pushl %edi
2717
2718 # Switch stacks
2719 movl %esp, (%eax)
2720 movl %edx, %esp
2721
2722 # Load new callee-save registers
2723 popl %edi
2724 popl %esi
2725 popl %ebx
2726 popl %ebp
2727 ret
```

Which registers need to be saved?

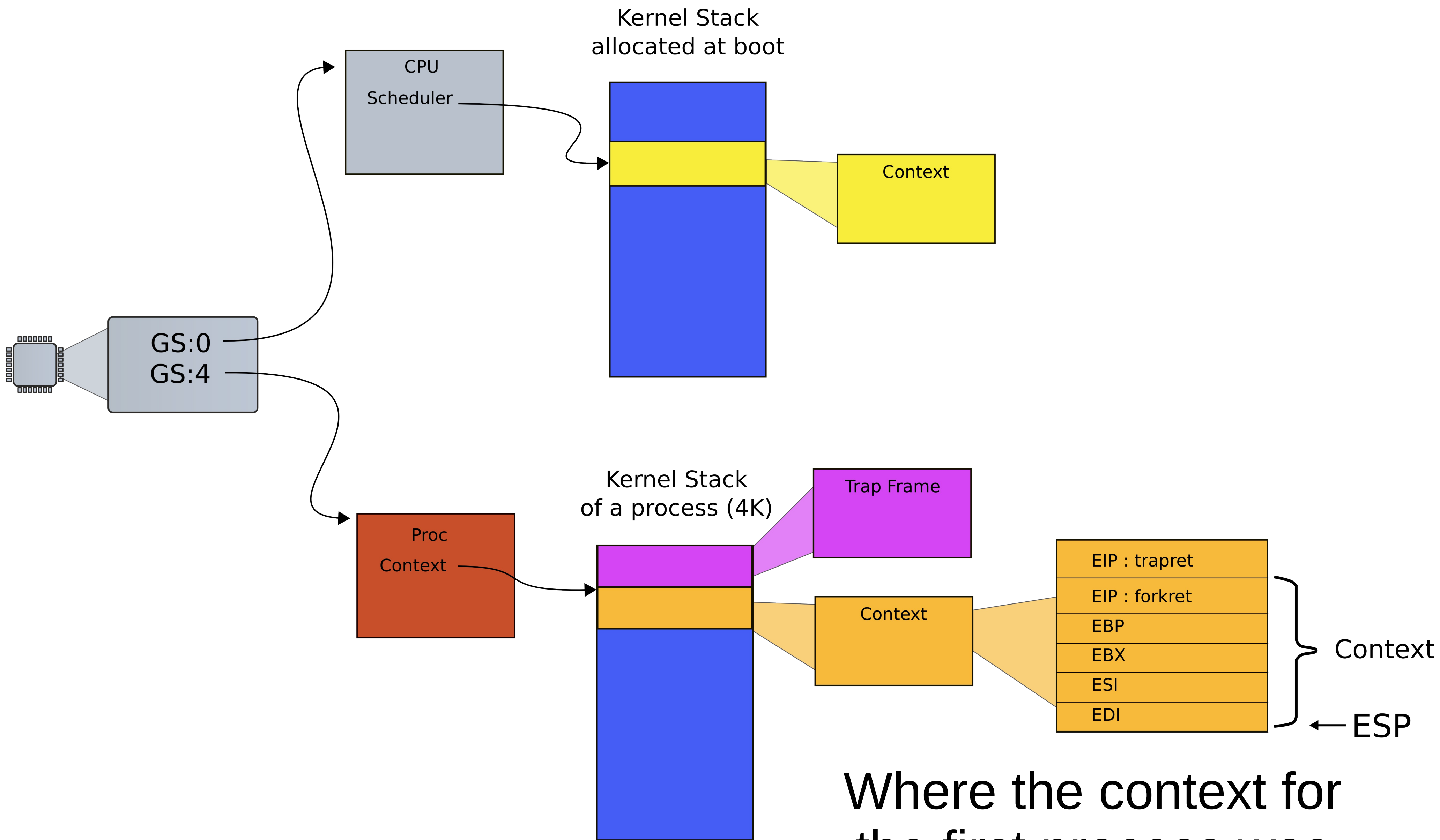
- Do not need to save all the segment registers (%cs, etc)
  - Constant across kernel contexts
- Do not need to save %eax, %ecx, %edx
  - The x86 convention is that the caller has saved them
- The stack pointer is the address of the context
- Do not save eip explicitly
  - But it is on the stack and allocproc() manipulates it

Context of the first process

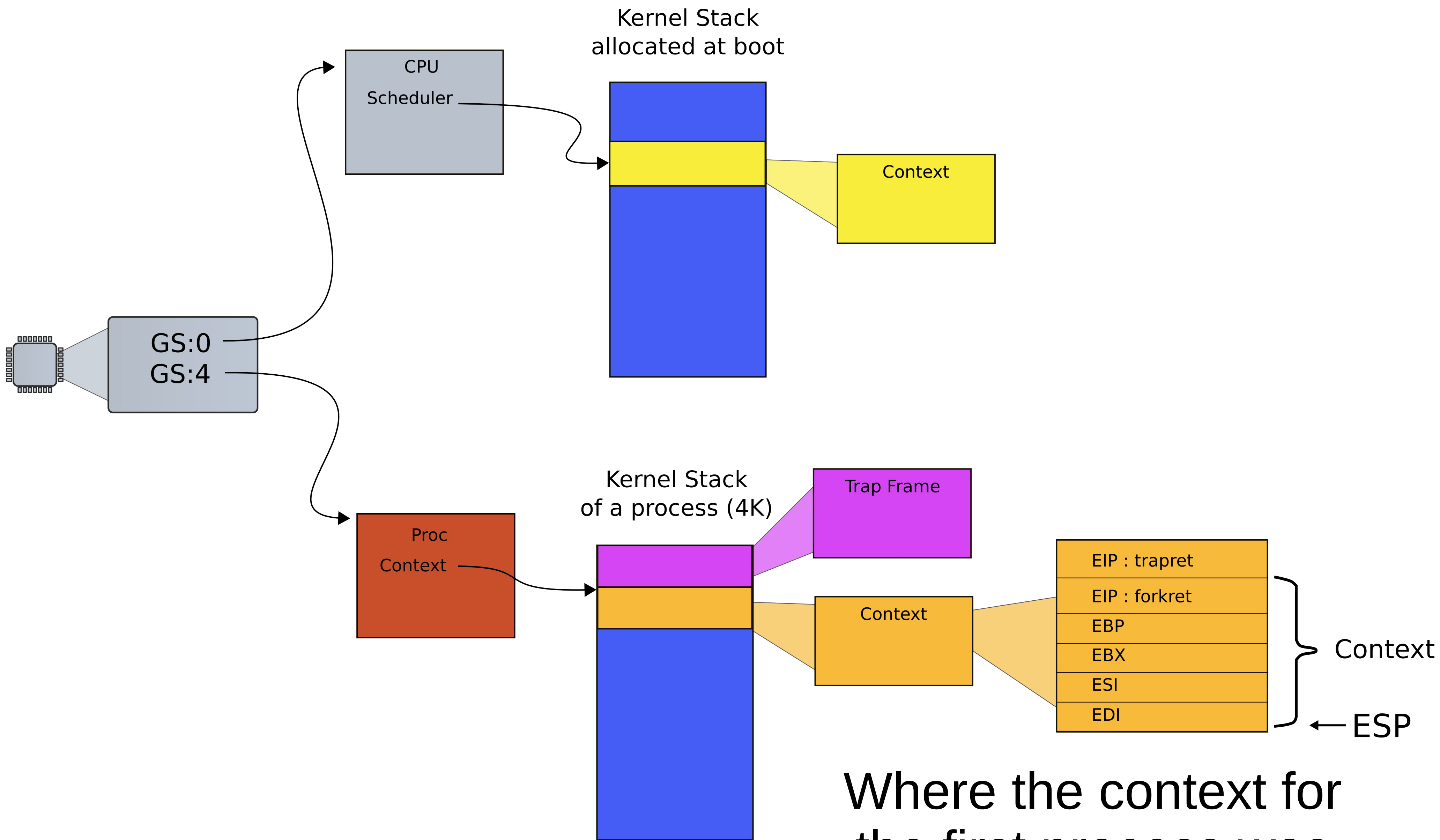


swtch(): load registers of the first process

```
2707 .globl swtch
2708 swtch:
2709 movl 4(%esp), %eax
2710 movl 8(%esp), %edx
2711
2712 # Save old callee-save registers
2713 pushl %ebp
2714 pushl %ebx
2715 pushl %esi
2716 pushl %edi
2717
2718 # Switch stacks
2719 movl %esp, (%eax)
2720 movl %edx, %esp
2721
2722 # Load new callee-save registers
2723 popl %edi
2724 popl %esi
2725 popl %ebx
2726 popl %ebp
2727 ret
```



Where the context for the first process was allocated?



Where the context for the first process was allocated?

allocproc()

```
2205 allocproc(void)
2206 {
...
2222   // Allocate kernel stack.
2223   if((p->kstack = kalloc()) == 0){
...
2227   sp = p->kstack + KSTACKSIZE;
...
2230   sp -= sizeof *p->tf;                // Leave room for trap frame.
2231   p->tf = (struct trapframe*)sp;
...
2235   sp -= 4;                            // Set up new context to start executing at forkret,
2236   *(uint*)sp = (uint)trapret;         // which returns to trapret.
2237
2238   sp -= sizeof *p->context;
2239   p->context = (struct context*)sp;
2240   memset(p->context, 0, sizeof *p->context);
2241   p->context->eip = (uint)forkret;
...
2244 }
```

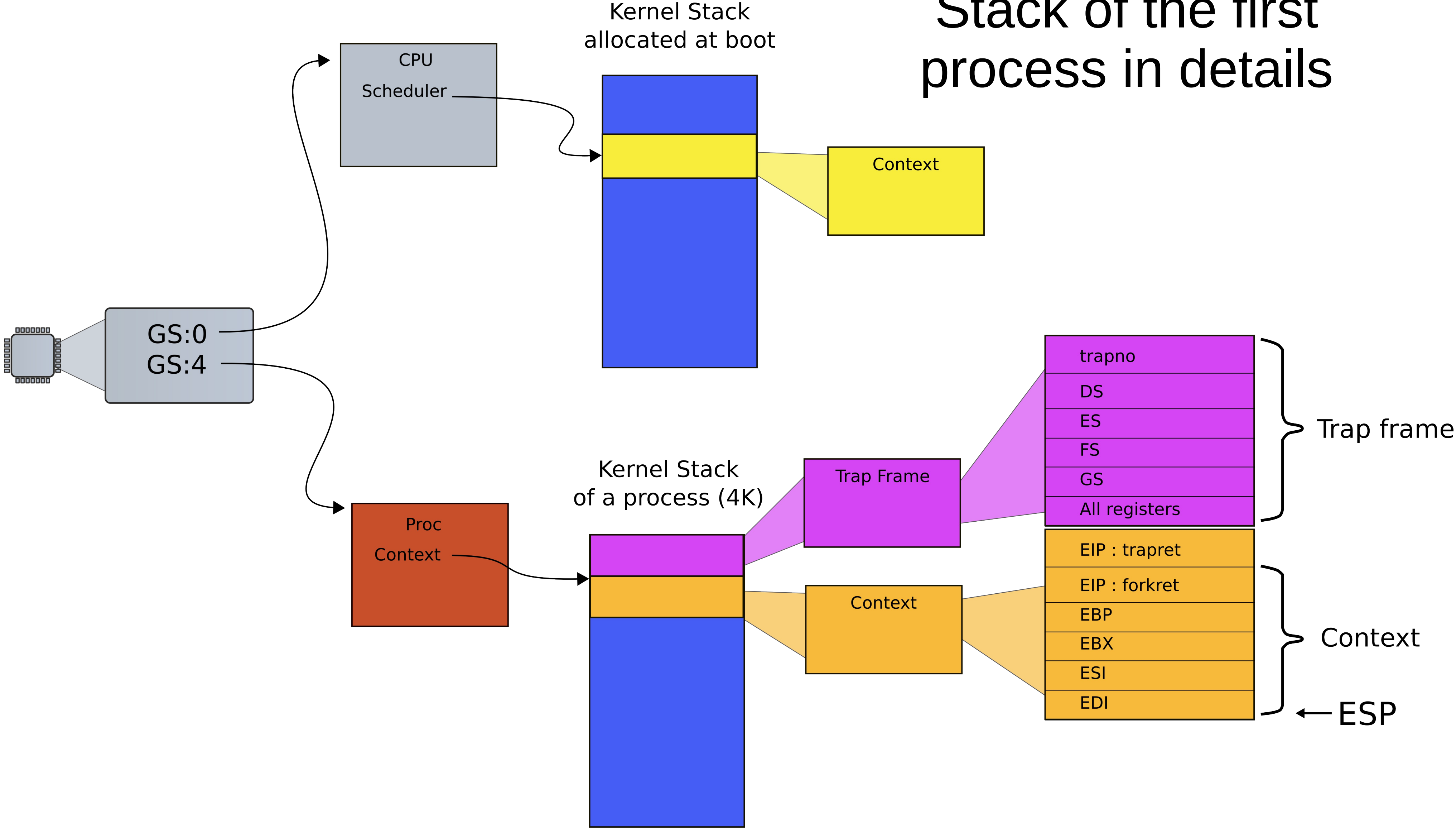
Stack of the first process

- The goal:
  - Encode exit to the user-space on the stack
  - Use two functions:
    - forkret()
    - trapret()

forkret()

```
2533 forkret(void)
2534 {
2535   static int first = 1;
...
2539   if (first) {
...
2543     first = 0;
2544     initlog();
2545   }
2546
2547   // Return to "caller", actually trapret
2548 }
```

Stack of the first process in details



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
```

Who creates the trapframe?

```
2252 userinit(void)
2253 {
...
2257   p = allocproc();
...
2263   memset(p->tf, 0, sizeof(*p->tf));
2264   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2265   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2266   p->tf->es = p->tf->ds;
2267   p->tf->ss = p->tf->ds;
2268   p->tf->eflags = FL_IF;
2269   p->tf->esp = PGSIZE;
2270   p->tf->eip = 0; // beginning of initcode.S
...
2276 }
```

Summary

- Initialized all CPUs
- Created the first process
- Finished initialization
- Entered the scheduler
- Started running the first process

Thank you!