Operativni sistemi

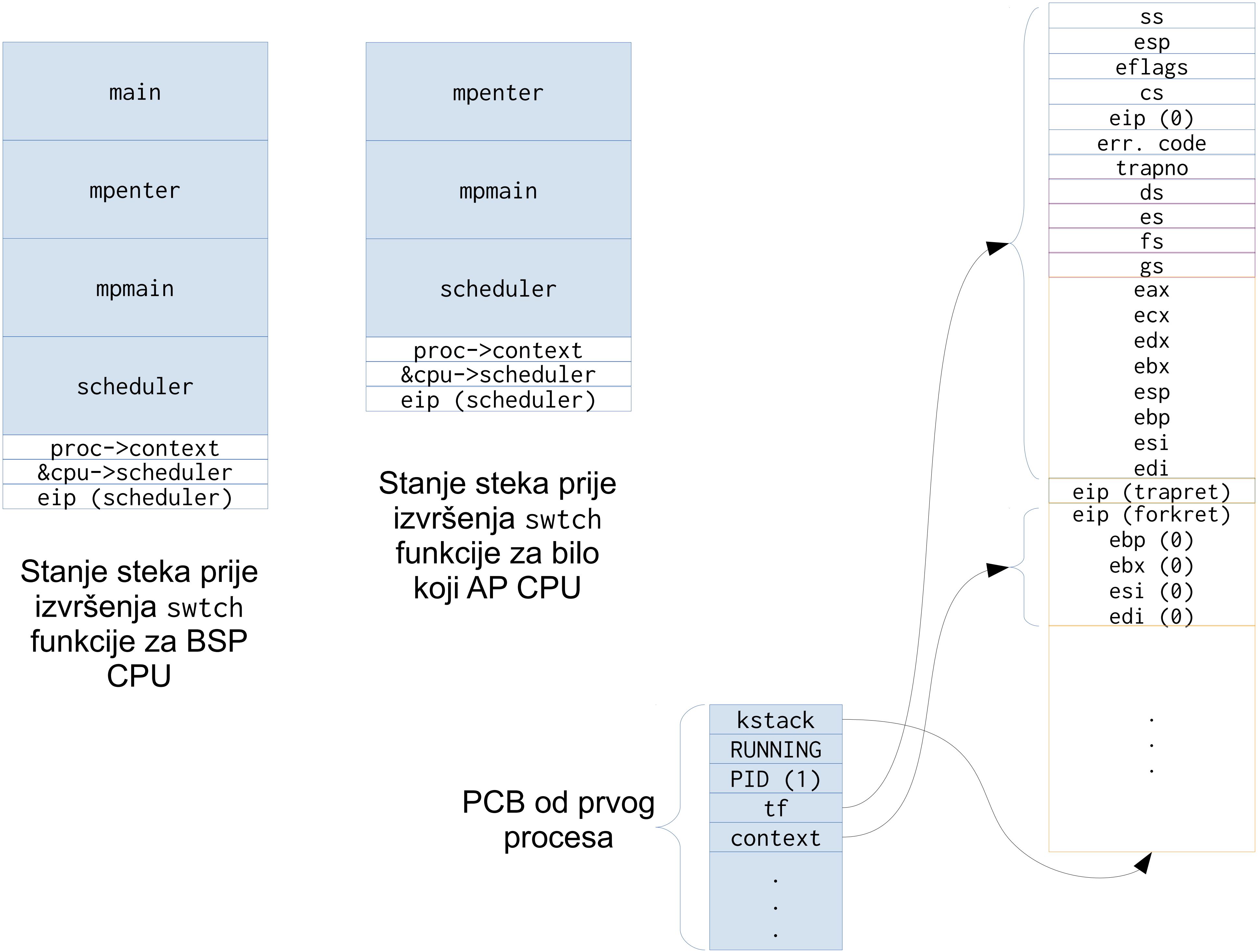
dr.sc. Amer Hasanović

Promjena konteksta

- AP procesori:
  - po pokretanju izvršavaju kod funkcije `scheduler`;
  - svaki AP ima svoj stek koje BSP alocira pozivom `kalloc` u funkciji `startothers`.
- BSP procesor:
  - nakon što pokrene AP-ove, izvršava kod funkcije `scheduler`;
  - ima svoj stek odvojen u BSS sekciji kernela u procesu linkanja.

- `scheduler` funkcija u beskonačnoj petlji:
  - pronalazi PCB prvog procesa koji je u stanju `RUNNABLE`;
  - sa `switchuvm`, aktivira adresni prostor odabranog procesa i postavlja TSS registar tako da kernel stek tokom servisiranja prekida bude `kstack` polje PCB-a odabranog procesa;
  - postavlja odabrani proces u stanje `RUNNING`, sa `swtch` prelazi na kernel stek od odabranog procesa, te nakon izvršenja `trapret` počinje u user modu izvršavati kod od procesa.

```
void scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.

    sti();

    // Loop over process table looking for process to run.

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.

      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

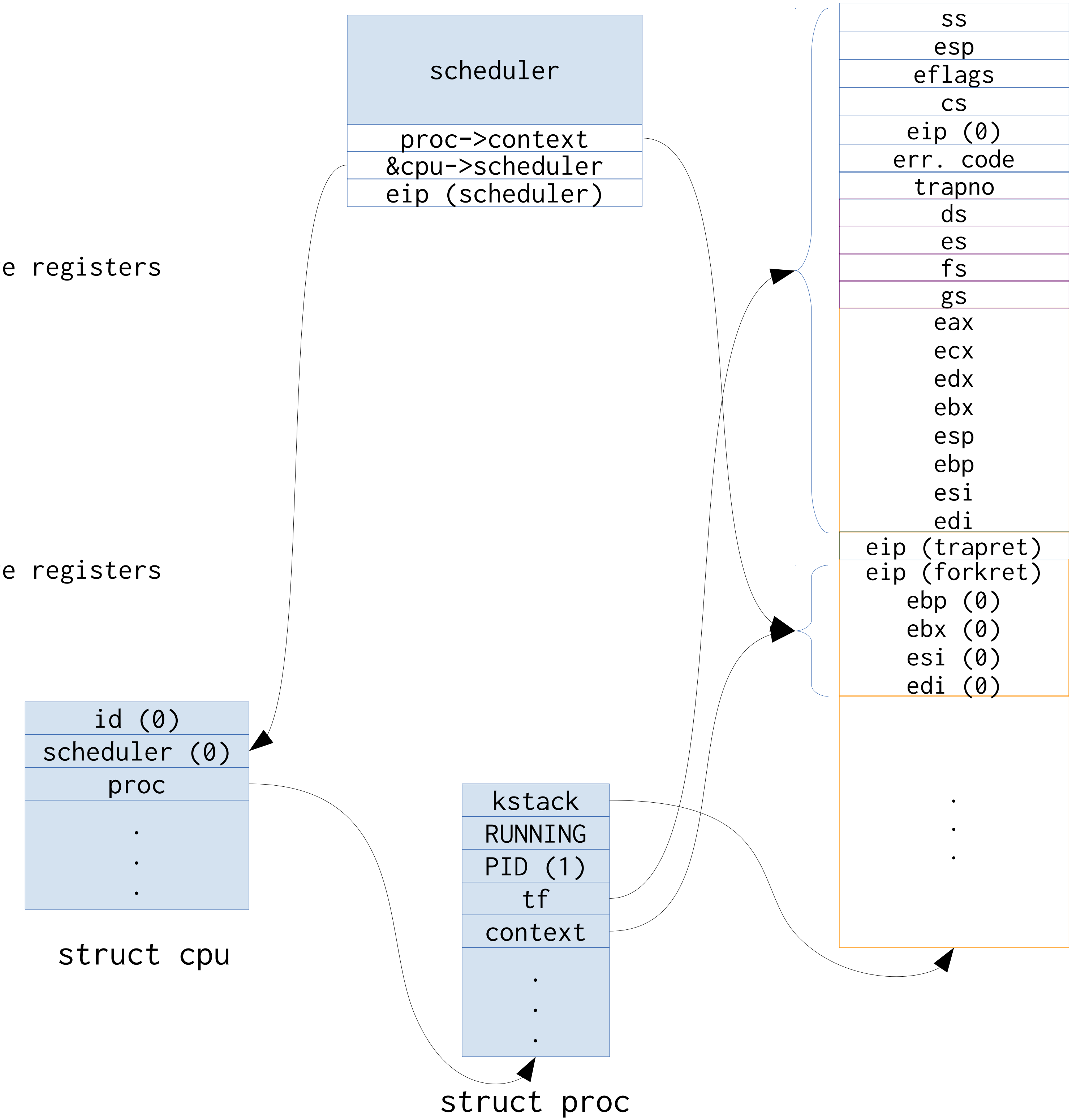


Stanje steka prije izvršenja swtch funkcije za BSP CPU

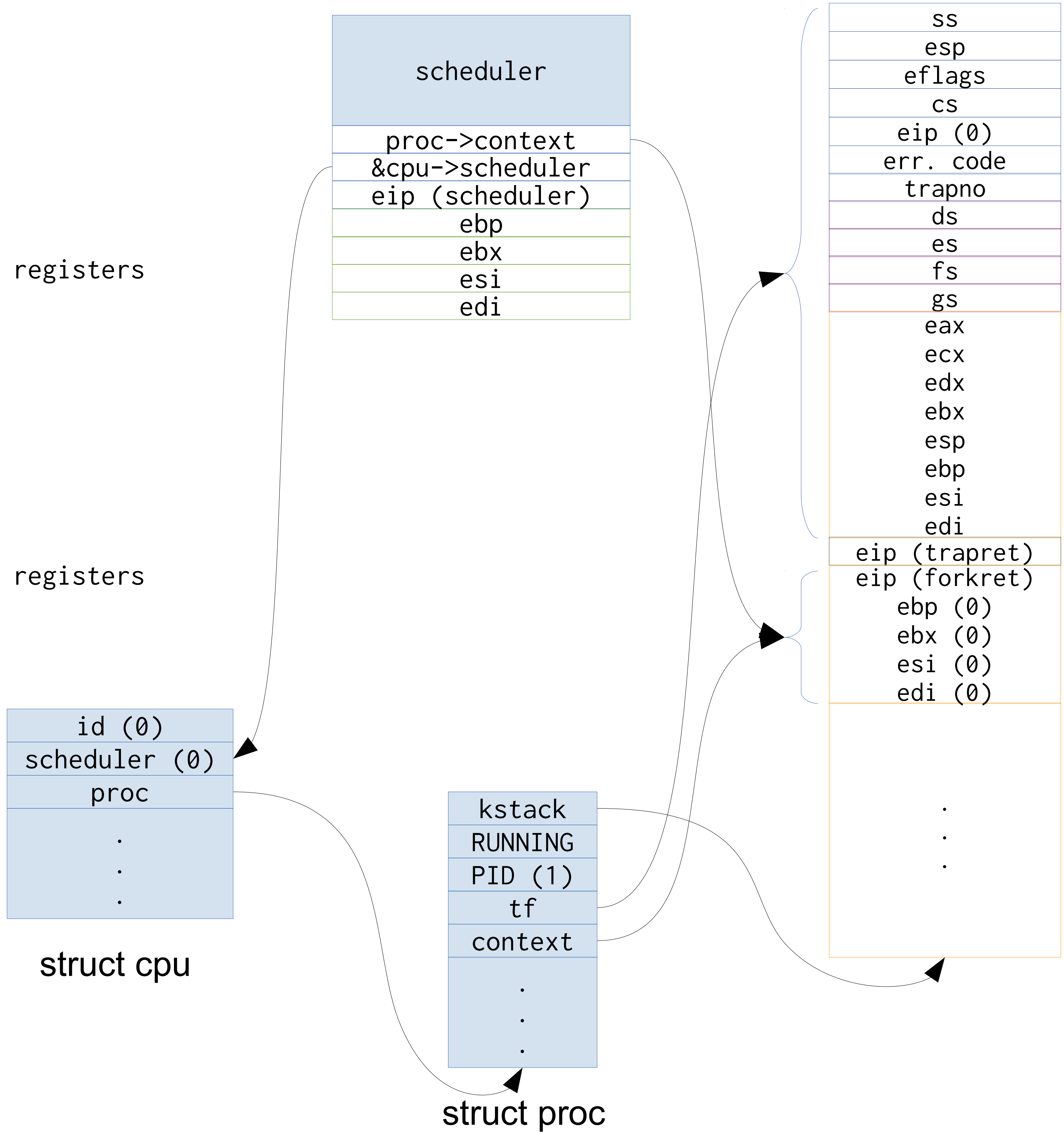
Stanje steka prije izvršenja swtch funkcije za bilo koji AP CPU

PCB od prvog procesa

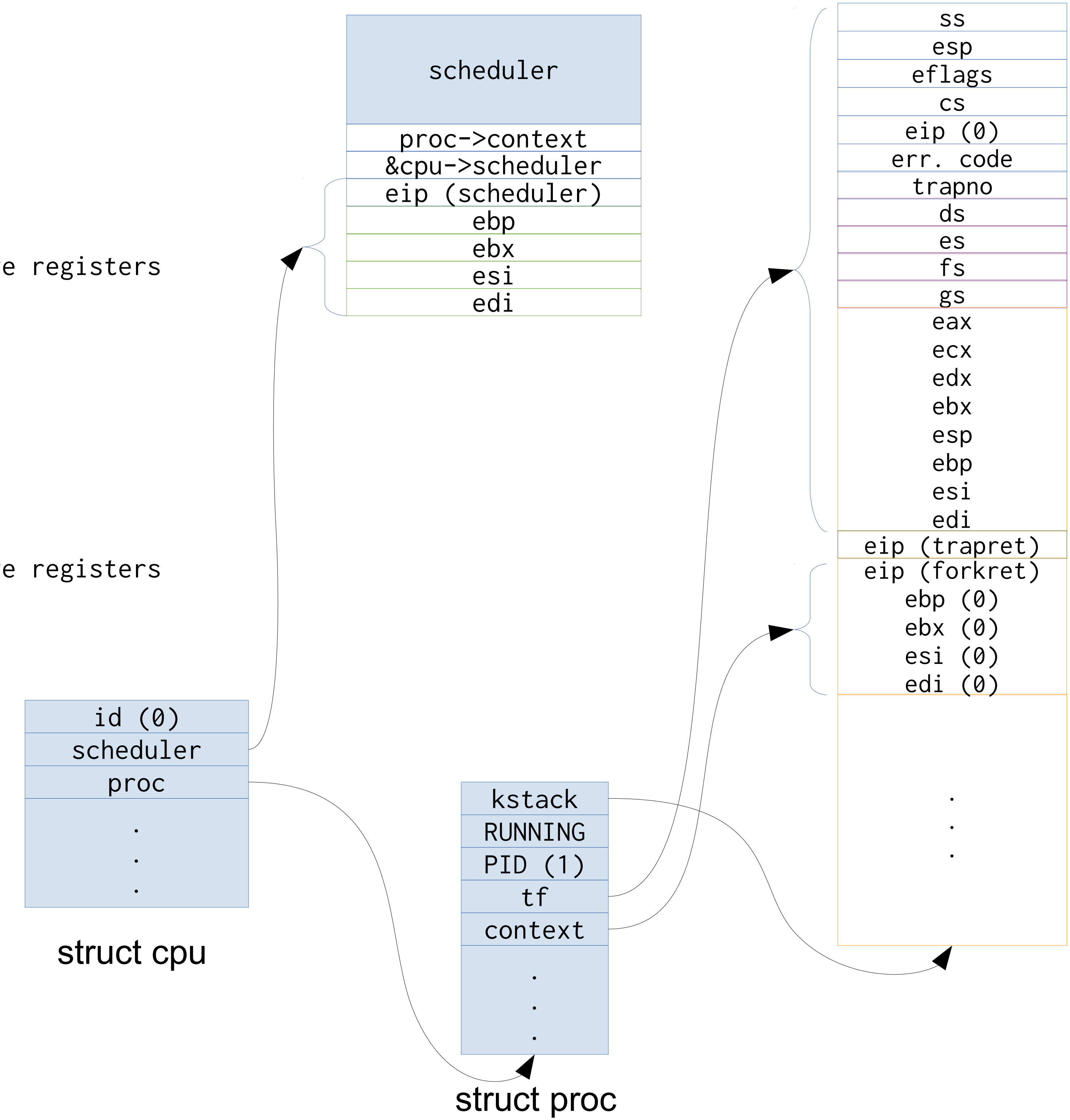
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # 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
```



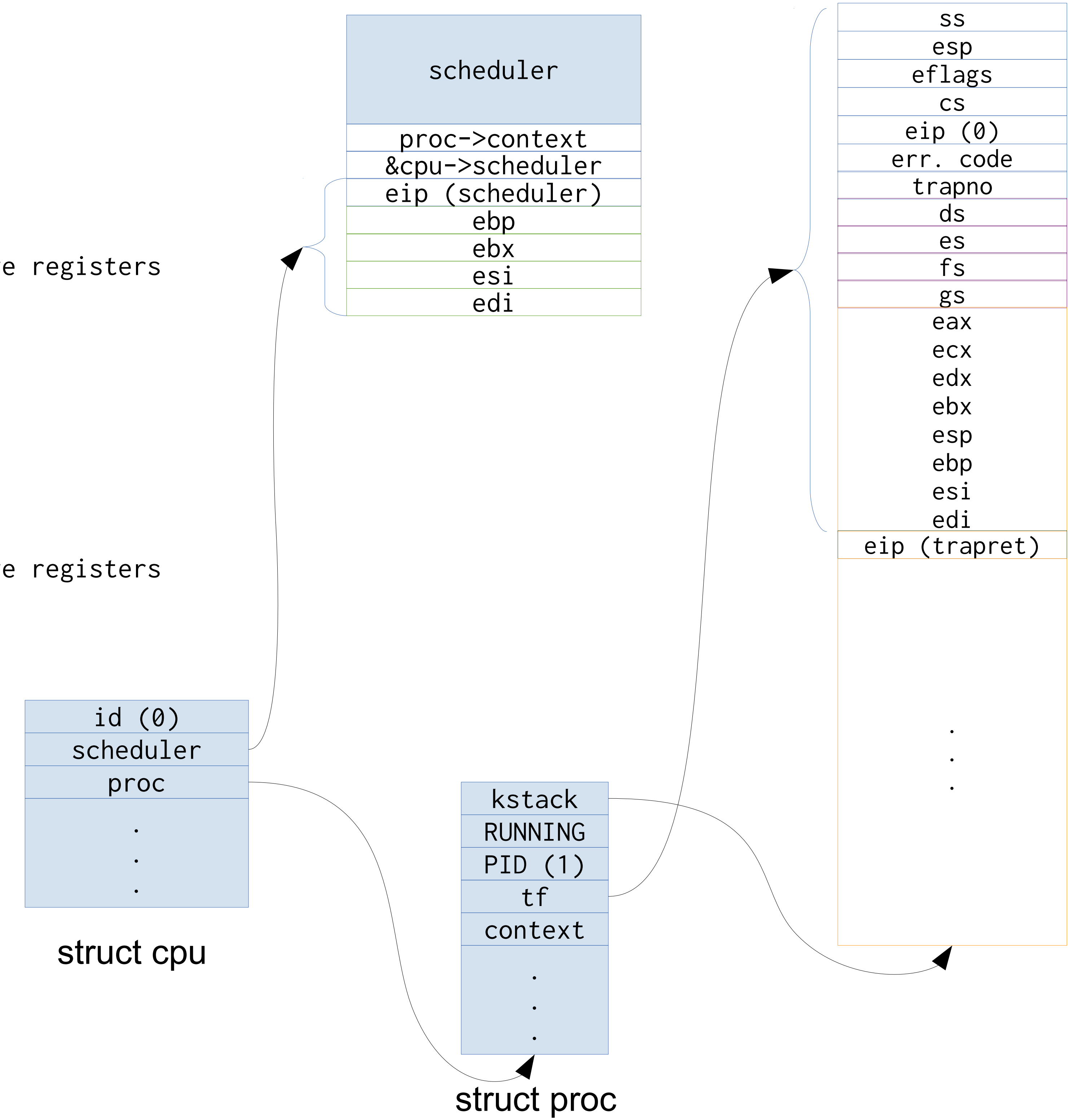
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # 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
```



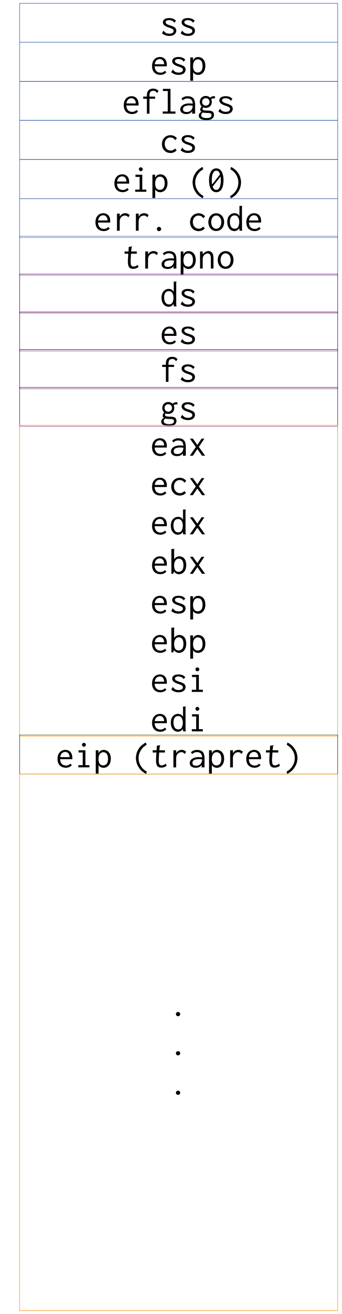
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # 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
```



```
.globl 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
```



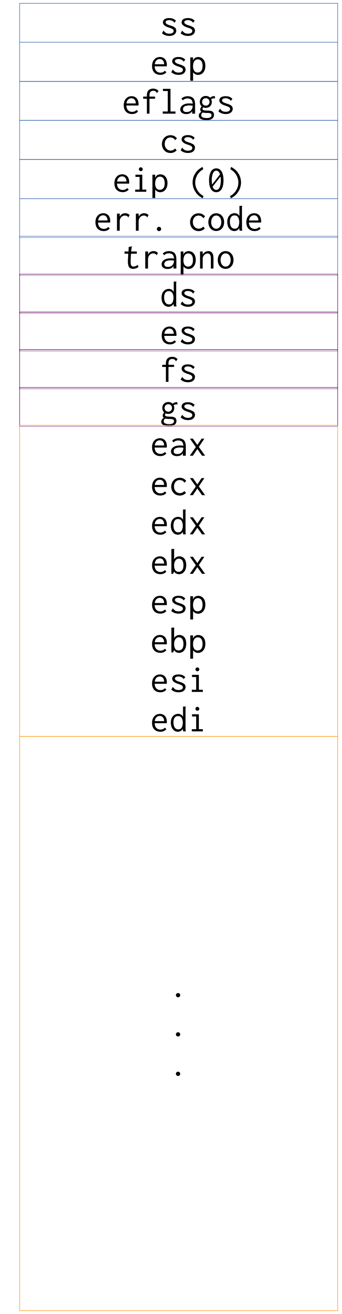
```c
void forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0;
    initlog();
  }
  // Return to "caller", actually trapret (see allocproc).
}
```



```c
void forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0;
    initlog();
  }
  // Return to "caller", actually trapret (see allocproc).
}
```

ss
esp
eflags
cs
eip (0)
err. code
trapno
ds
es
fs
gs
eax
ecx
edx
ebx
esp
ebp
esi
edi
.
.
.

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



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

ss
esp
eflags
cs
eip (0)
err. code
trapno
ds
es
fs
gs
. . .

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

ss
esp
eflags
cs
eip (0)
err. code
trapno
. . .

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

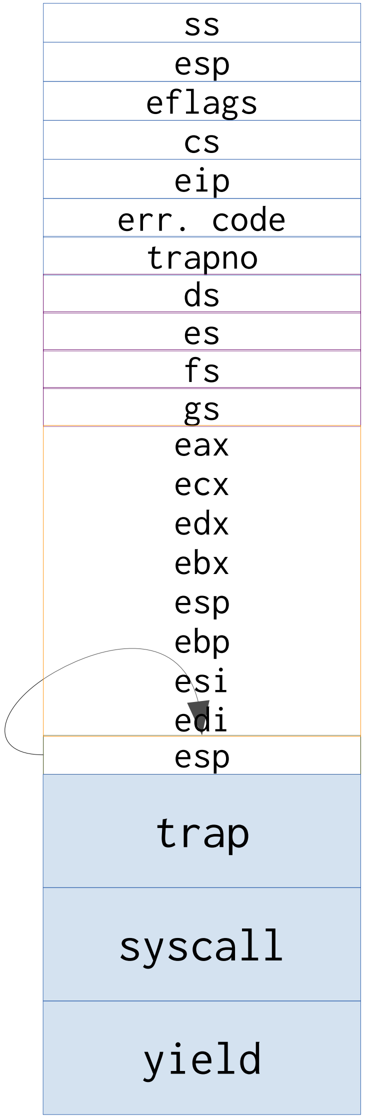
Kernel stek procesa ostaje prazan:
Izvršen prelazak u user mode:

- `esp` postavljen na vrh user steka
- `eip` postavljen na prvu instrukciju (adresa `0`) u funkciji `main`

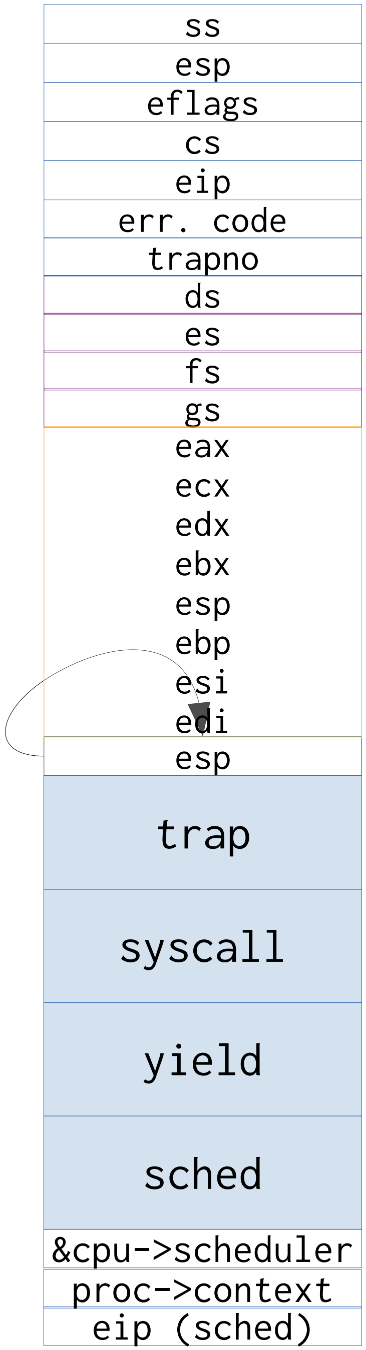
Povratak u scheduler

- Predpostavimo da je prilikom tretmana nekog sistemskog poziva procesa pozvana funkcija `yield`

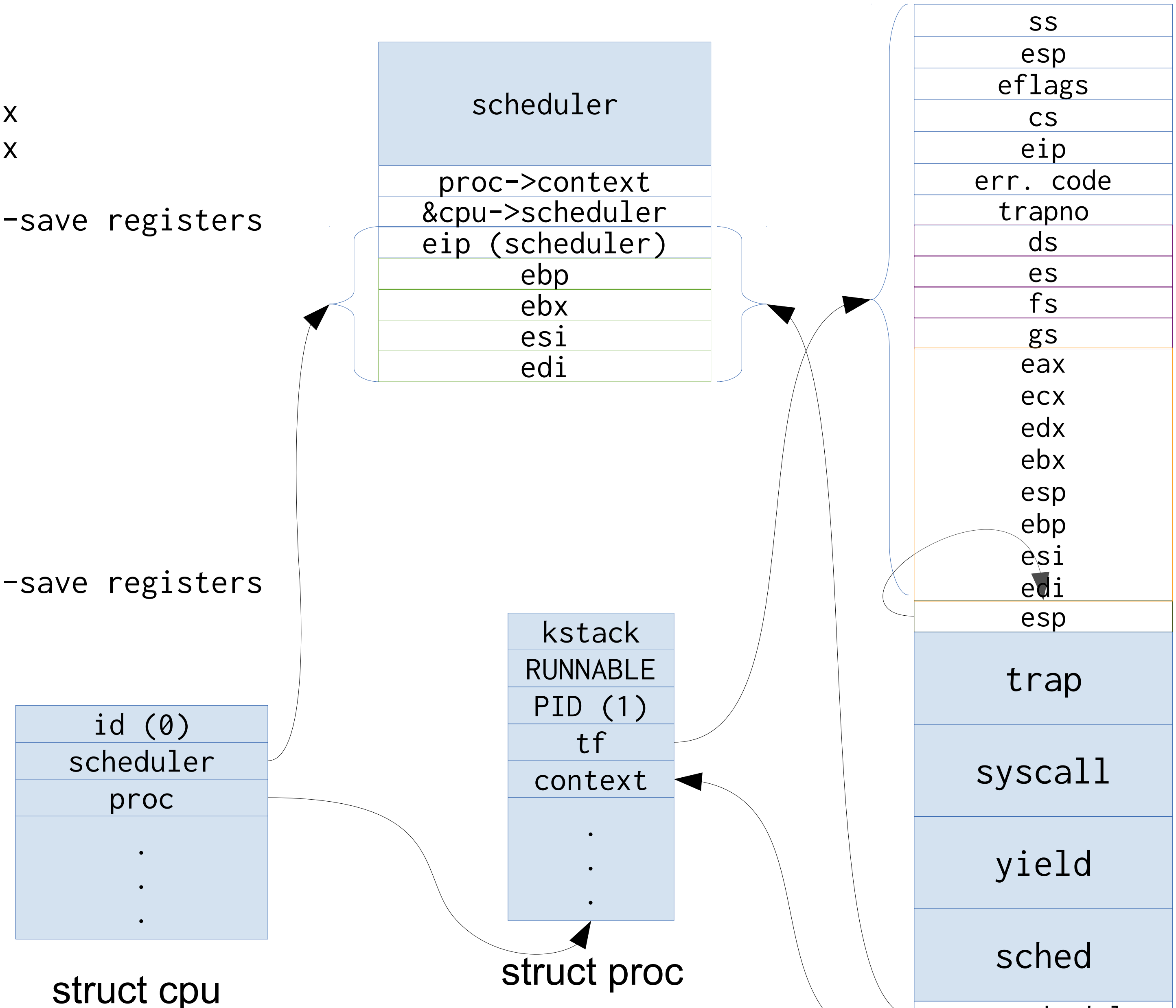
```
void yield(void)
{
  acquire(&ptable.lock);
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```



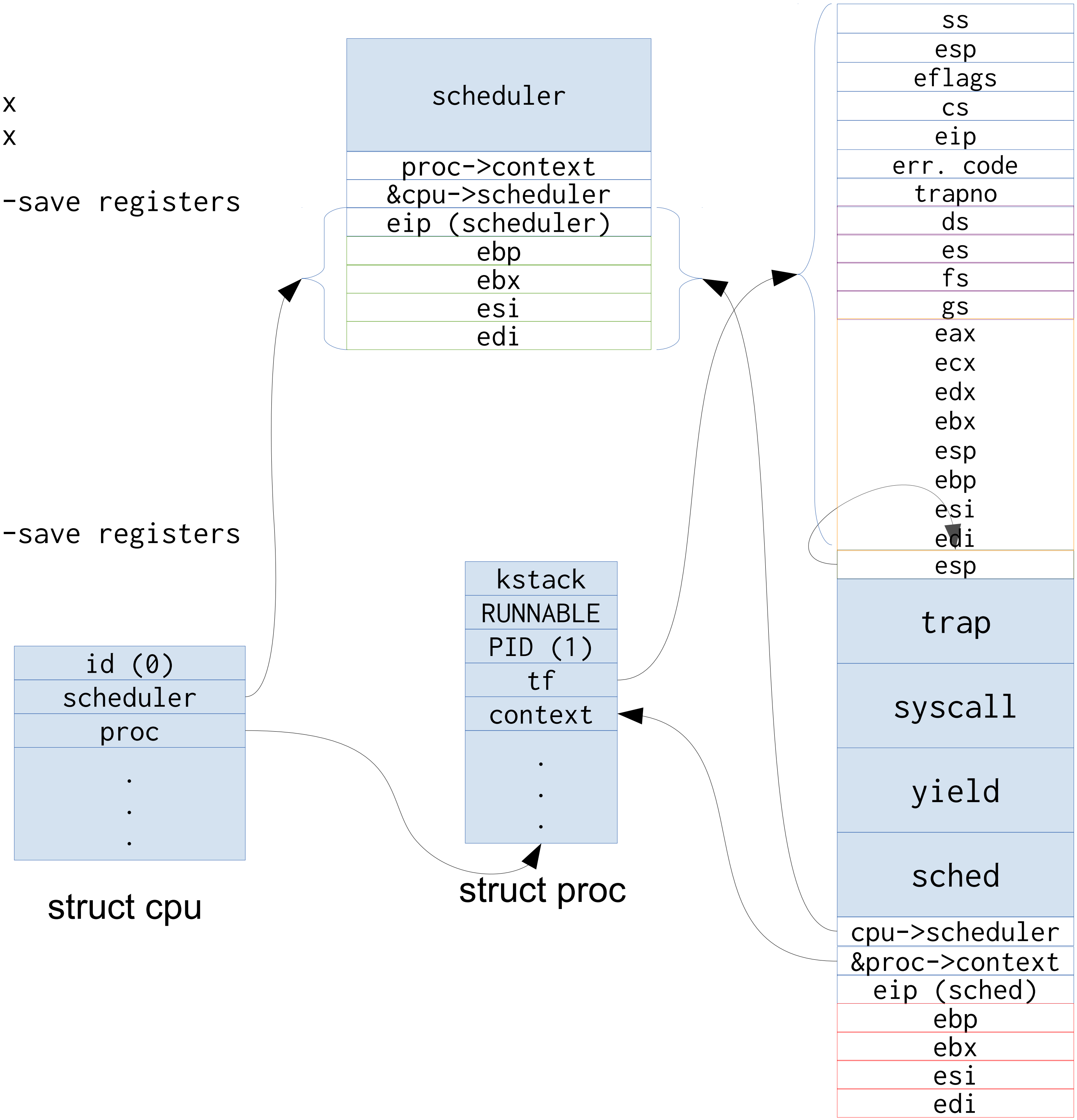
```
void sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}
```



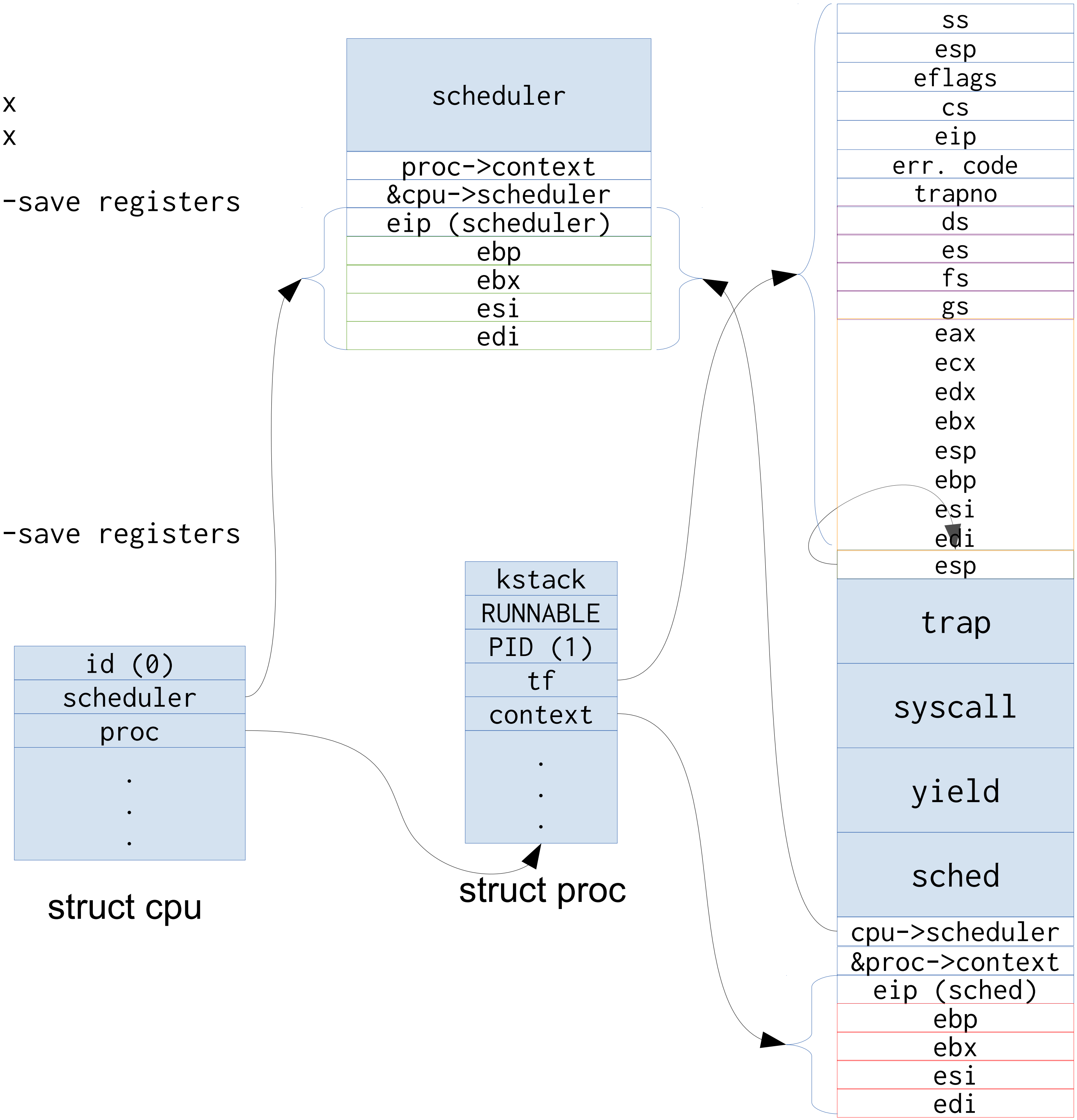
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # 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
```



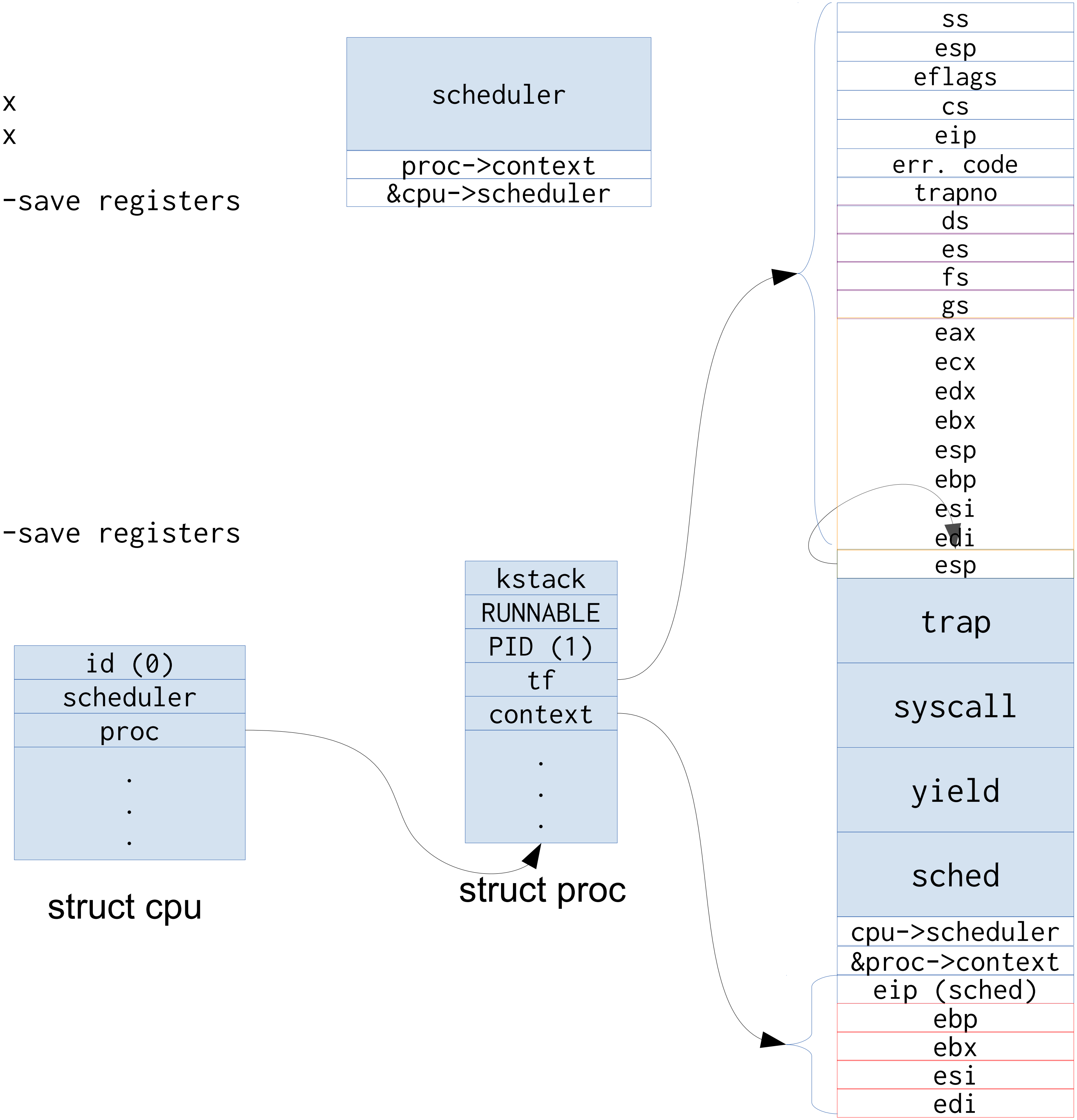
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # 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
```



```
.globl 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
```



```
.globl 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
```



```c
void scheduler(void)
{
  struct proc *p;

  for(;;){
    sti();
    acquire(&ptable.lock);
    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;
    }
    release(&ptable.lock);
  }
}
```

scheduler

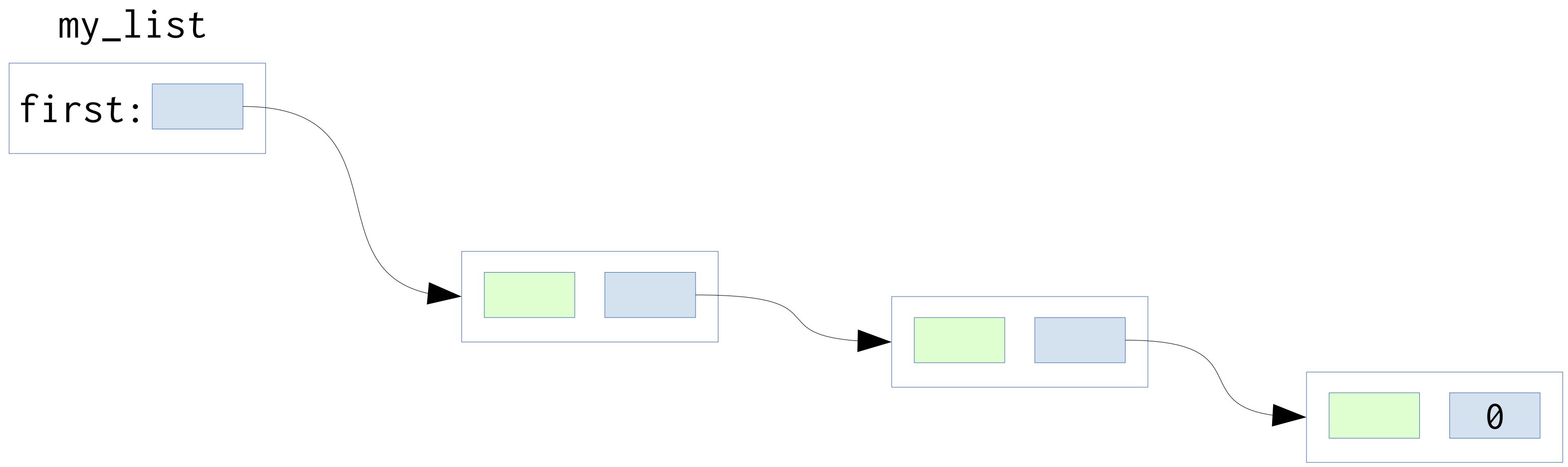
id (0)
scheduler
proc (0)
. . .

struct cpu

Stanja utrke i sinhronizacija

```c
struct node {
  int data;
  struct node *next;
};

struct list {
  struct node *first;
} my_list;

void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

- Neka je data definicija jednostruko povezane liste struktura tipa node
- Neka su aktivna dva procesorska jezgra
  - Neka jezgro `id=1` izvodi operaciju `insert(10);`
  - Neka jezgro `id=2` izvodi operaciju `insert(20);`

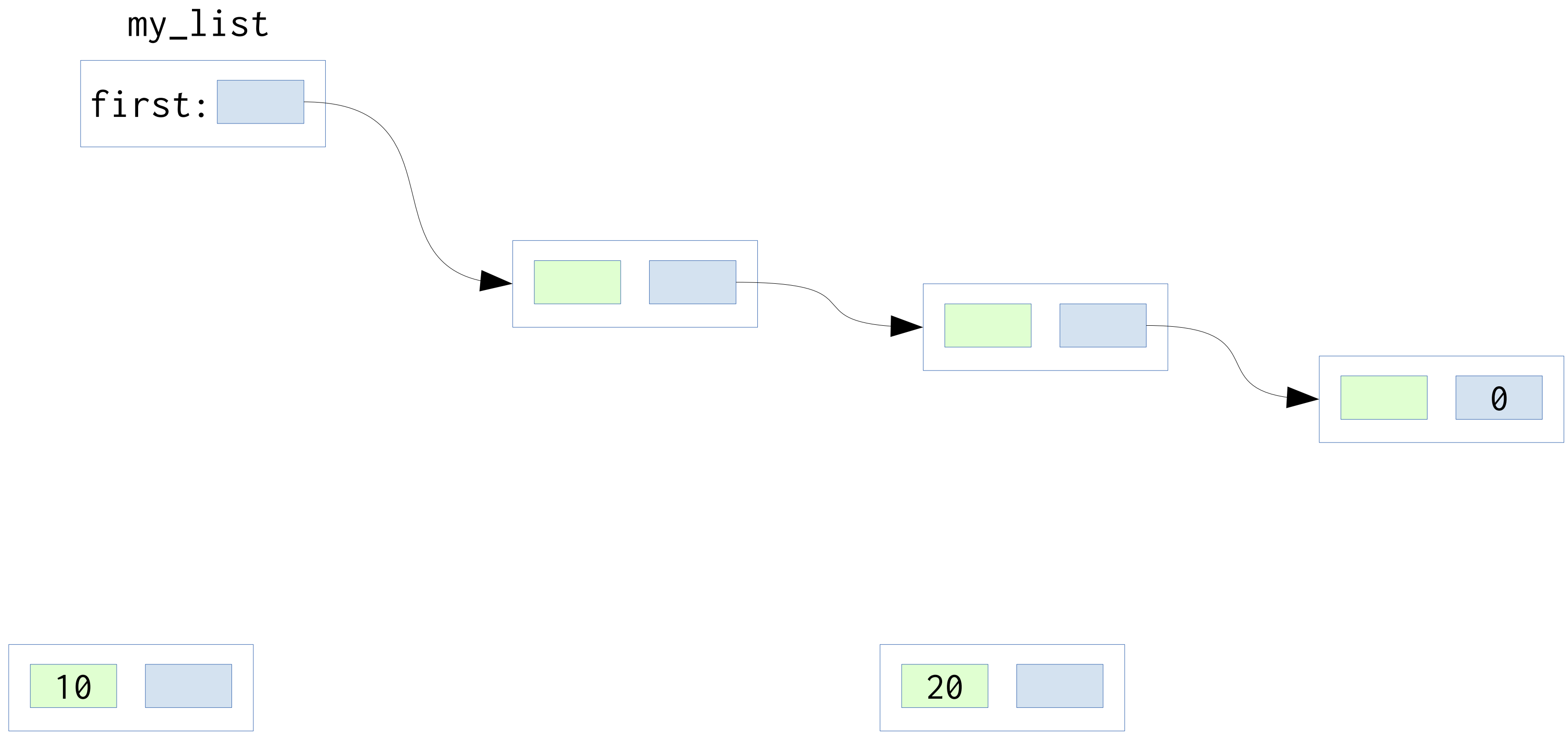


cpu1: insert(10);

cpu2: insert(20);

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

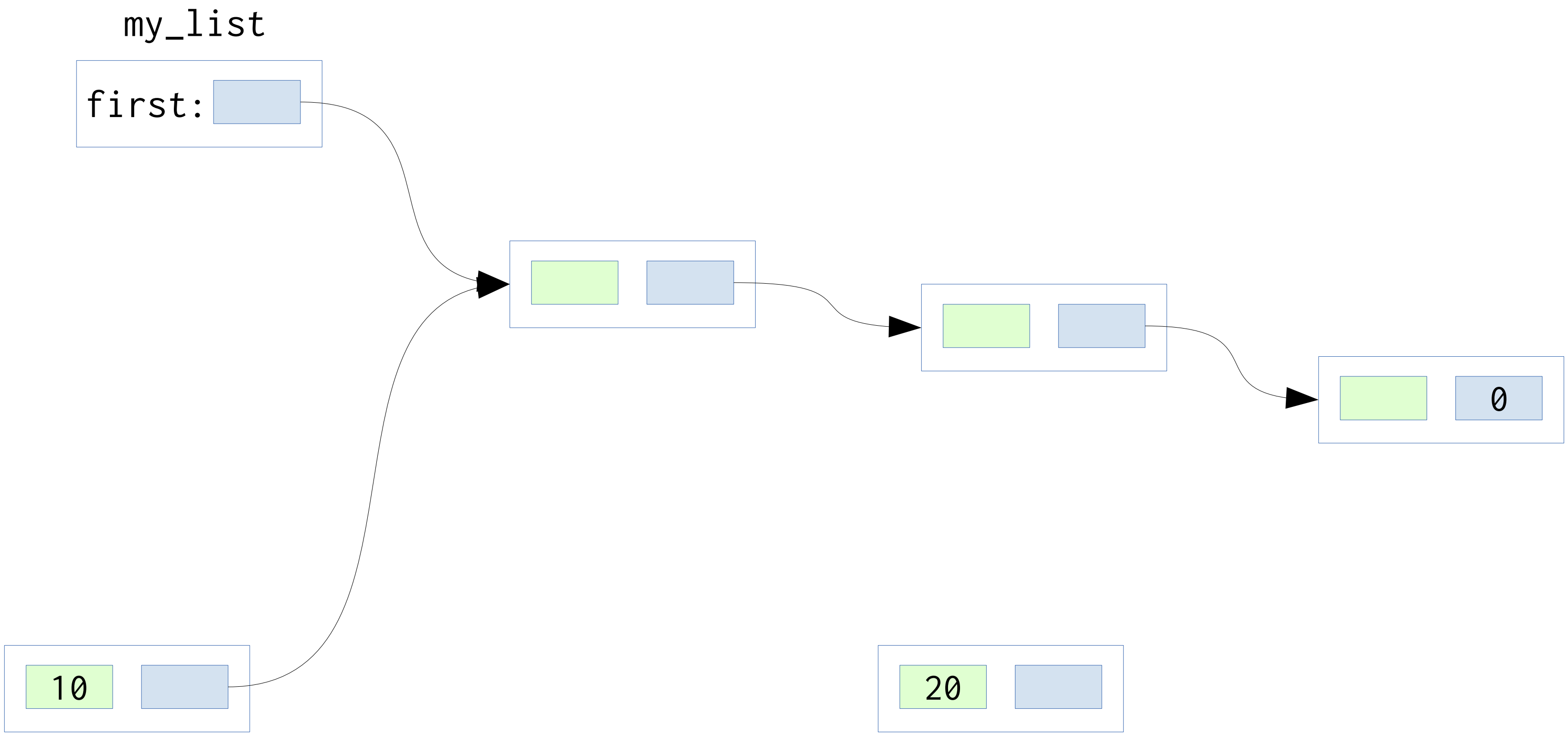


cpu1: insert(10);

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

cpu2: insert(20);

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

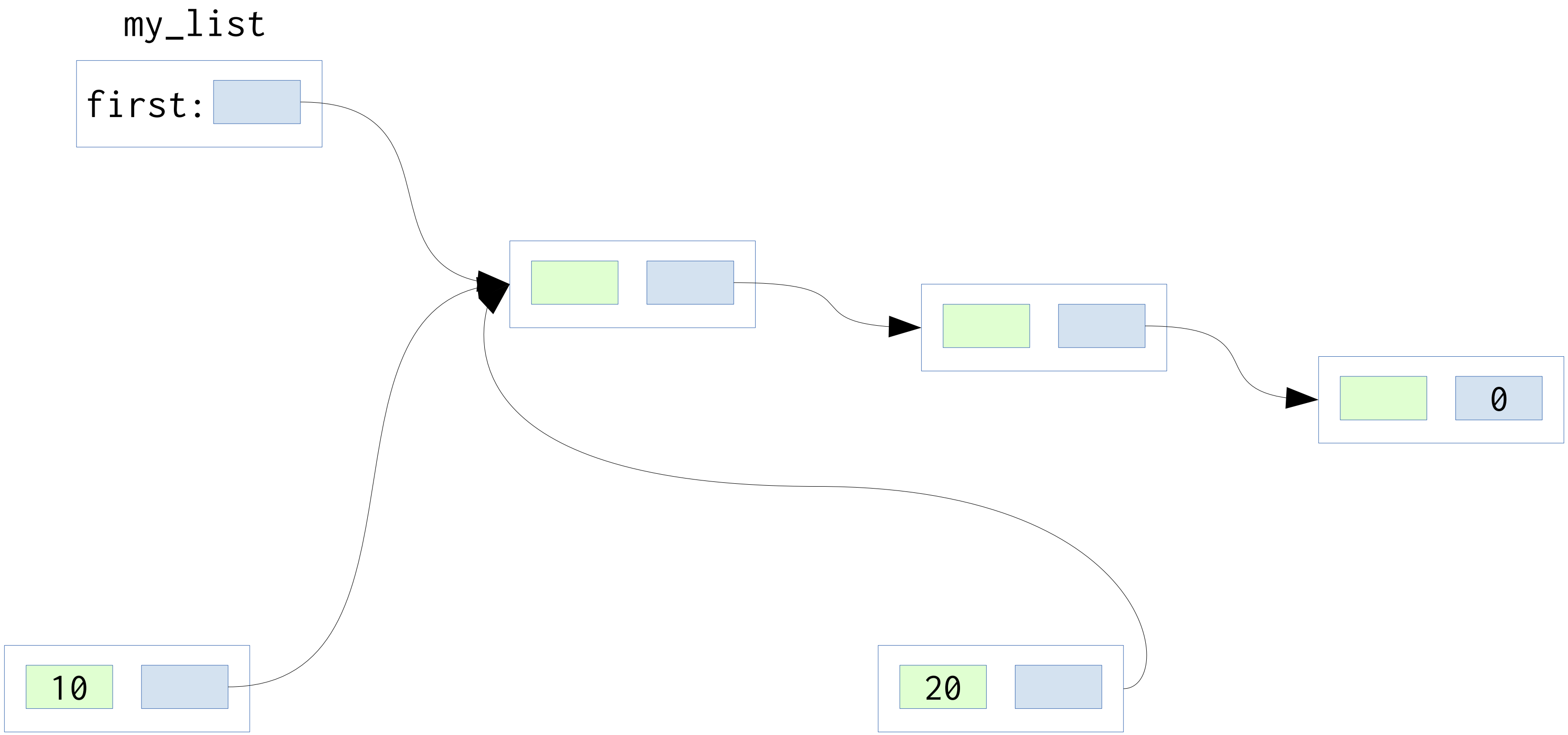


cpu1: insert(10);

cpu2: insert(20);

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

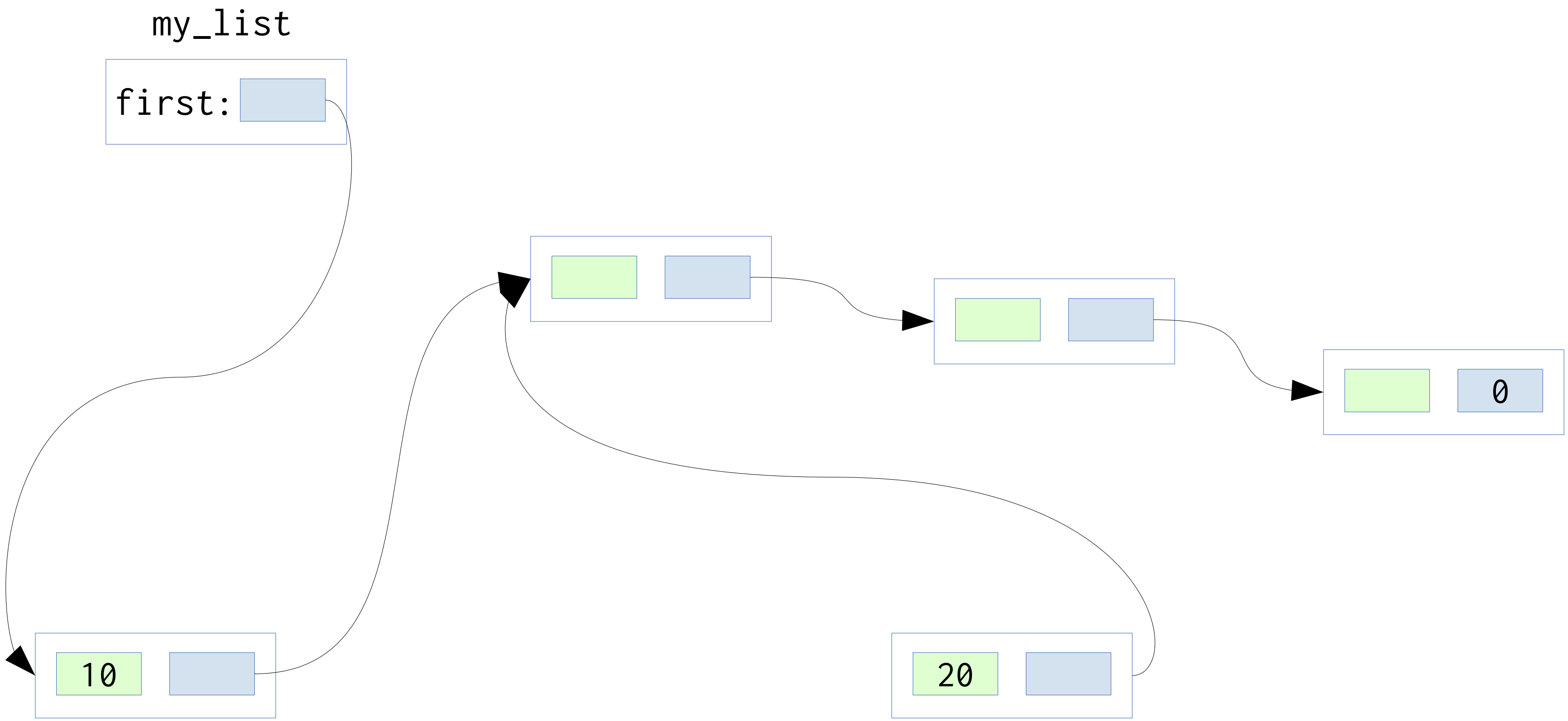


cpu1: insert(10);

cpu2: insert(20);

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

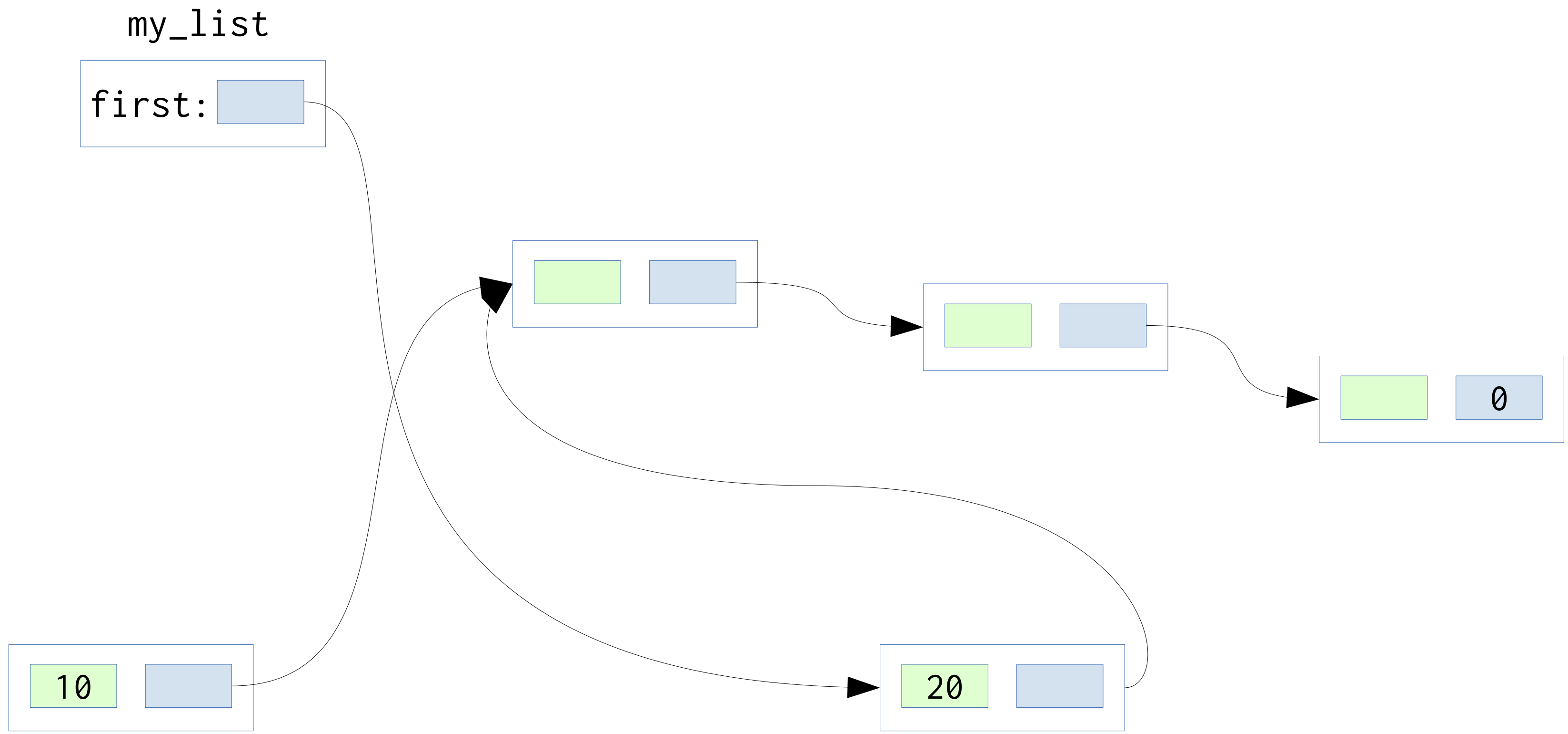


cpu1: insert(10);

cpu2: insert(20);

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```



cpu1: insert(10);

cpu2: insert(20);

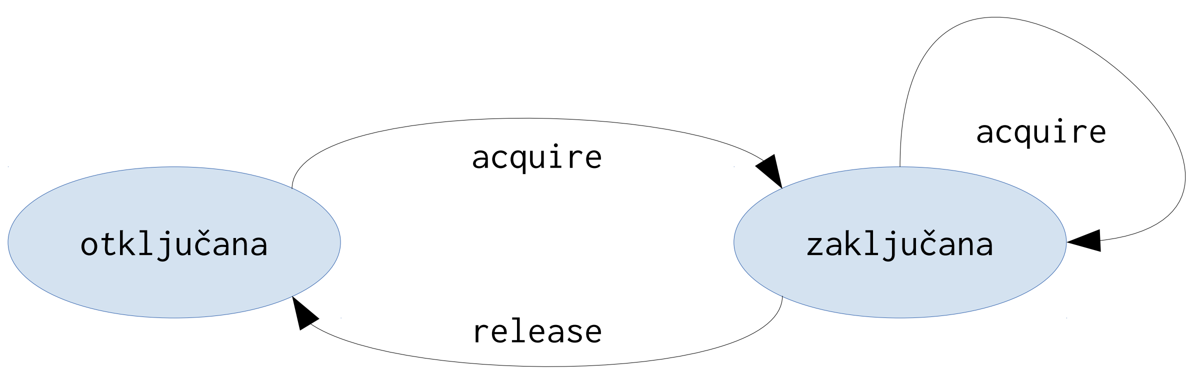
```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

```
void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  n->next = my_list.first;
  my_list.first = n;
}
```

- jezgra 1 i 2 su obavljali operacije kojima je manipuliran dijeljeni resurs zbog čega nastaje **stanje utrke** (race condition)
  - u ovom slučaju djeljeni resurs je kontejner čvorova `my_list`.
- zbog toga što dijelovanje jezgri nije koordinirano konačan rezultat njihovih operacija je nedeterminističan.
  - u ovom slučaju čvor sa vrijednosti `10` ostaje van kontejnera
- Ako više procesora mogu da manipuliraju nekim resursom konkuretno, da bi konačno stanje resursa bilo determinističano, sve operacije nad resursom moraju biti **atomske**
  - tj da se obavljaju u potpunosti, bez prekida i/ili smetnji.
- Stanja utrke rješavaju se uvođenjem sinhronizacijskih primitiva kao što je **brava** (lock).

Operacije nad bravama

- Sinhronizacijiski primitiv brava:
  - podržava operacije `acquire` i `release`;
  - ima stanja zaključana i otključana;
  - CPU koji pozove operaciju `acquire` na zaključanoj bravi mora čekati unutar funkcije `acquire` dok brava ne postane otključana.
  - za svaki dijeljeni resurs alocira se po jedna brava



Sinhronizirani my_list

```
struct node {
  int data;
  struct node *next;
};

struct list {
  struct node *first;
  struct lock list_lock;
} my_list;

void insert(int data) {
  struct node *n = malloc(sizeof(struct node));
  n->data = data;
  acquire(&my_list.list_lock);
  n->next = my_list.first;
  my_list.first = n;
  release(&my_list.list_lock);
}
```

spinlock - implementacija brave

```
struct lock {
  uint locked;
};

void acquire(struct lock *lk) {
  while(xchg(&lk->locked, 1) != 0)
    ;
}

void release(struct lock *lk) {
  xchg(&lk->locked, 0);
}
```

`xchg` atomski:

- postavlja vrijednost `newal` na adresu `addr`
- vraća staru vrijednost sa adrese `addr`

```
static inline uint xchg(volatile uint *addr, uint newval) {
  uint result;
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

zastoji

- Bez obzira na putanju u kodu, brave uvijek treba da se zaključavaju istim redosljedom
  - u suprotnom mogući su **zastoji**

```
struct lock A;
struct lock B;
```

```
void f1() {
  acquire(&A);
  acquire(&B);

  // manipulacija resursima

  release(&B);
  release(&A);
}
```

izvršava CPU1

```
void f2() {
  acquire(&B);
  acquire(&A);

  // manipulacija resursima

  release(&A);
  release(&B);
}
```

izvršava CPU2

Brave i prekidi

- Neka CPU1 prilikom tretmana nekog sistemskog poziva počne izvršavati kod od funkcije `f1`
  - tokom tretmana sistemskih poziva `xv6` ne maskira hadrverske prekide
- Predpostavimo da se u trenutku kada se iz funkcije `f1` pozove funkcija `f2` dogodi prekid čiji *ISR* pozove funkciju `f_isr`
  - u ovo situaciji nastaje zastoj

```
void f1() {
  acquire(&A);
  f2();
  release(&A);
}
```

```
void f_isr() {
  acquire(&A);

  // servisiranje prekida

  release(&A);
}
```

- Kod koji preuzme bravu ne smije biti prekidan.
- Ukoliko putanja koda zahtijeva zaključavanje više brava, hardverski prekidi trebaju biti maskirani sve dok se ne otključa zadnja brava.

```
void acquire(struct lock *lk) {
  pushcli();
  while(xchg(&lk->locked, 1) != 0)
    ;
}
```

```
void release(struct lock *lk) {
  xchg(&lk->locked, 0);
  popcli();
}
```

```
void pushcli(void) {
  int eflags;
  eflags = readeflags();
  cli();
  if(cpu->ncli++ == 0)
    cpu->intena = eflags & FL_IF;
}
```

```
void popcli(void) {
  if(--cpu->ncli < 0)
   panic("popcli");
  if(cpu->ncli == 0 && cpu->intena)
    sti();
}
```