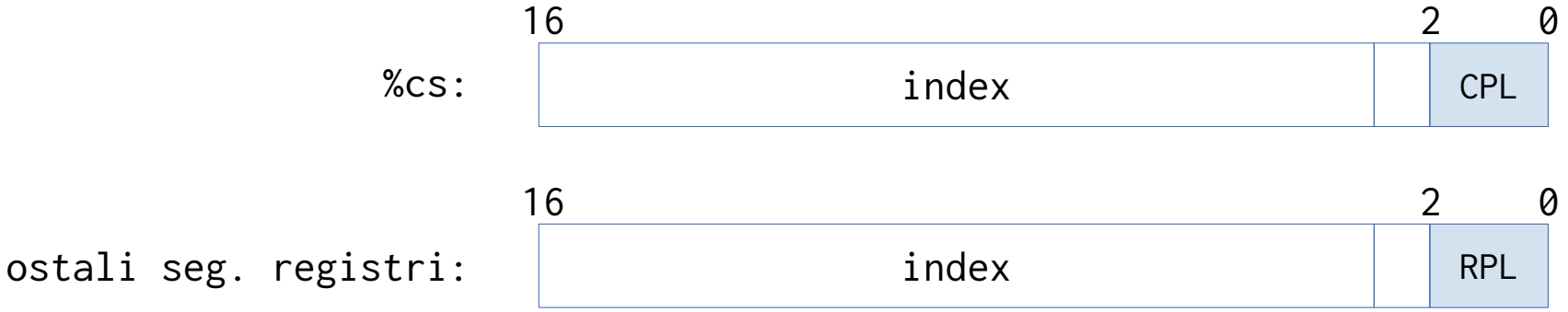
Operativni sistemi

dr.sc. Amer Hasanović

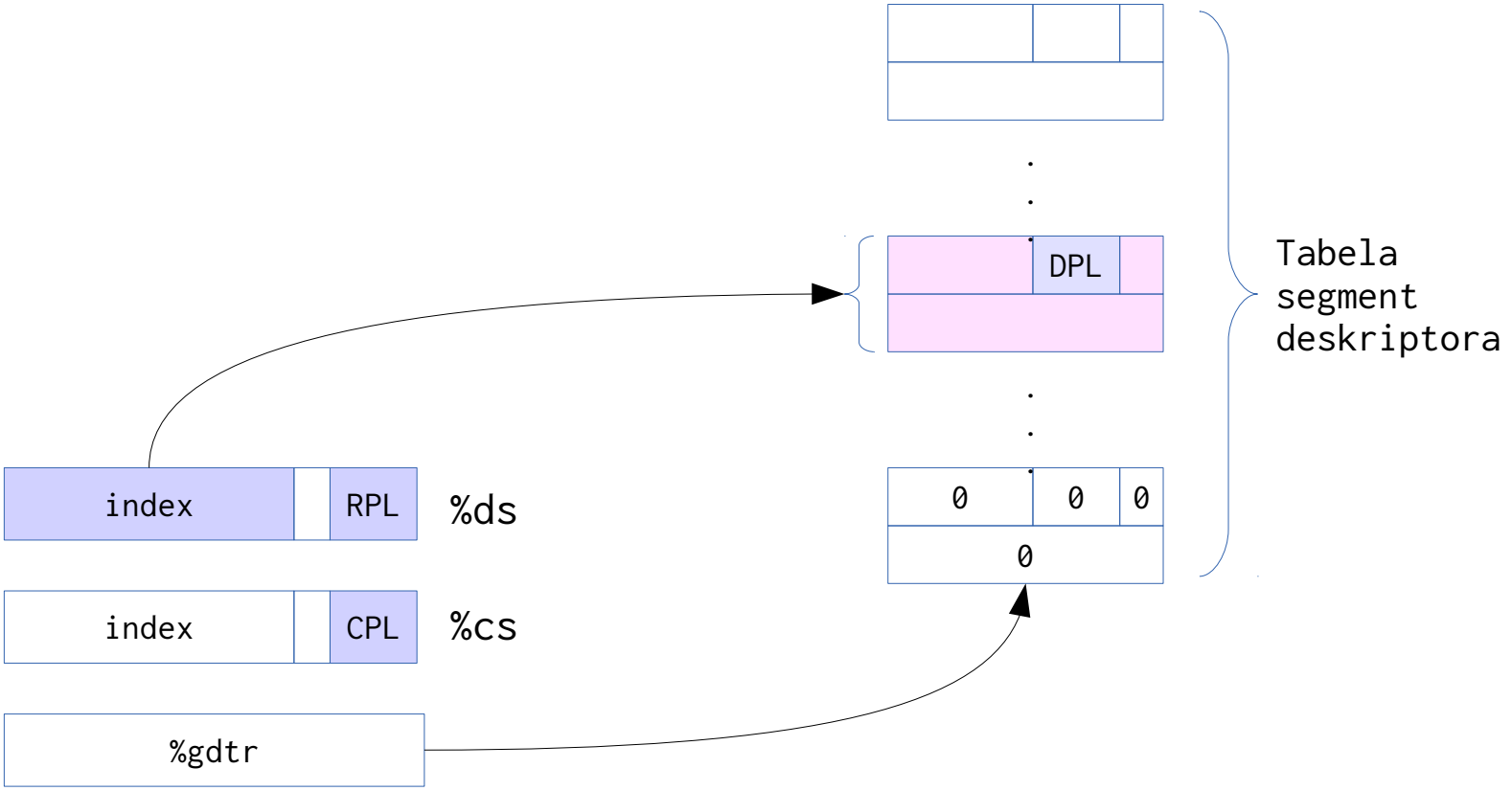
OS protekcija

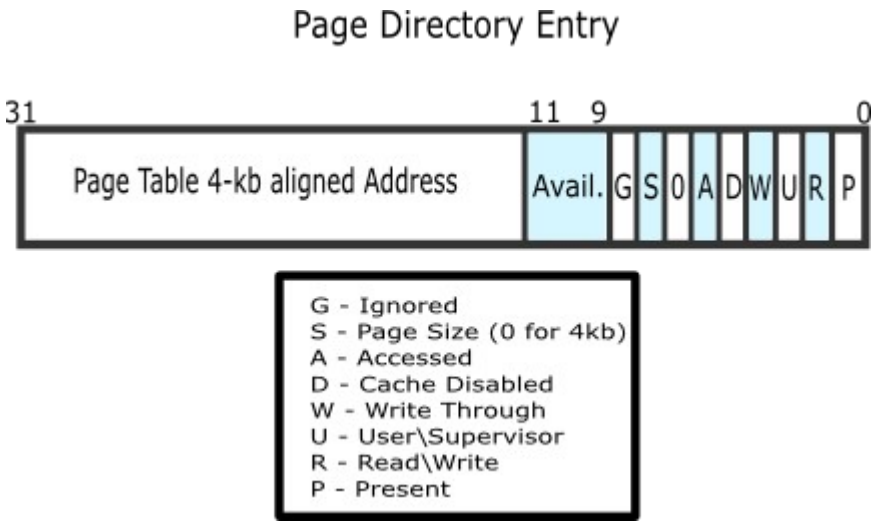
- Dok operira u protected modu x86 arhitektura definira 4 prstena protekcije koji uključuju različite nivoe dozvola spram:
  - instrukcija koje se na raspolaganju;
  - operanada koji su na raspolaganju za određene instrukcije.
- Prstenovi su označeni brojevima od 0 do 3:
  - najviše privilegija nosi prsten 0, a najmanje 3
- Operativni sistemi, obično koriste prsten 0 dok izvršavaju kod od OS-a, a prsten 3 dok izvršavaju kod od korisničkih aplikacija.

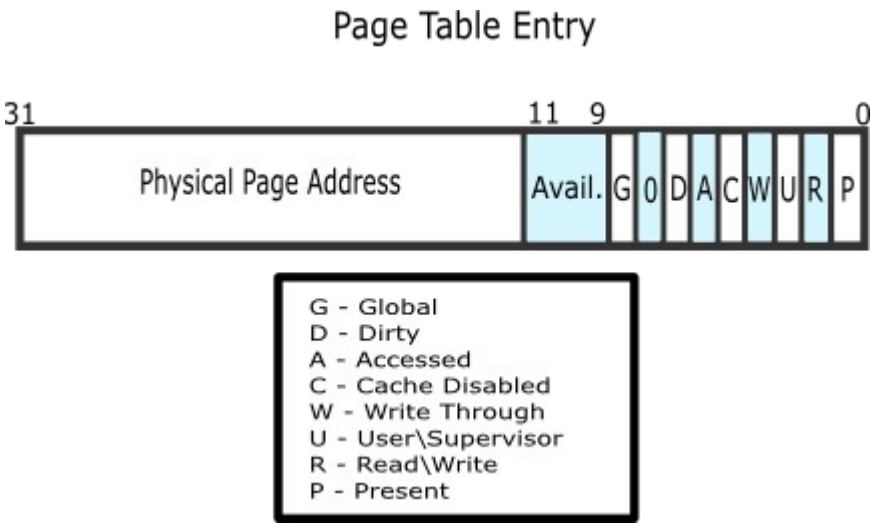
- **CPL** (Current Privilege Level) polje unutar registra `%cs` definira prsten protekcije u kojem trenutno operira procesor;
  - ovo polje u ostalim segmentnim registrima zove se **RPL** (Requested Privilege Level)
- Ukoliko se privilegovana instrukcija (npr `lgdt`) pokuša izvršiti dok je procesor u krugu `3`, generira se odgovarajuća iznimka.



- Prilikom pokušaja učitavanja novih vrijednosti u segmentne registre procesor vrši provjere spram `CPL`, `RPL` i `DPL` (Descriptor Privilege Level) pri čemu:
  - ako je `max(CPL, RPL) <= DPL` → učitavanje je dozvoljeno;
  - u suprotnom generira se iznimka (exception)







- Iznimka će biti generirana i u situaciji
  - kada je cpu u prstenu 3;
  - a pristupa se adresi unutar virtuelne stranice čije je PTE `U` polje postavljeno na `0`.

Promjena nivoa privilegija CPU-a

- Vrijednosti segmentnih registara mogu se direktno mijenjati (pomoću `mov`, `pop` itd) uz navedenu provjeru privilegija;
  - osim `%cs` registra.
- Promjena vrijednosti u `%cs` registru, tj promjena trenutnog prstena privilegije CPU-a događa se samo tokom:
  - tretmana prekida (interrupt);
  - tretmana iznimki;
  - sistemskih poziva (instrukcije `int` i `iret`)

Normalan tok programa

- Normalni tok programa uključuje slijedeće korake koje obavlja CPU
  - dohvat nove instrukcije (fetch);
  - dekodiranje preuzete instrukcije (decode);
  - izvršenje dekodirane instrukcije (execute);
  - prelaz na slijedeću instrukciju u programu;
- Tokom normalnog toka programa vrijednost registra `%eip` mijenja se unutar segmenta odabranog sa registrom `%cs`

Prekidi

- Normalan tok programa može se prekinuti na tri načina:

1. Hardver prekid
    - CPU nakon izvršenja trenutne instrukcije konstatuje promjenjen napon na posebnoj liniji (interrupt linija) kojom se signalizira potreba da kernel servisira neki uređaj;
        - npr, pomjeraj miša, pritisnut taster na tasaturi itd.
2. Iznimka
    - CPU konstatuje iznimno stanje u trenutnoj instrukciji bilo u fetch, decode ili execute fazi
        - npr, loše formatirana instrukcija, dijeljenje sa 0 itd.

3. Sistemski poziv

- Trenutna instrukcija koja se izvodi predstavlja softverski prekid (instrukcija `int`) kojim aplikacija signalizira potrebu da kernel servisira njene potrebe
  - npr, ispis teksta na ekranu, učitavanje bloka sa diska u virtuelni prostor aplikacije itd.

- Sistemski pozivi i iznimke dešavaju se sinhrono, a hardverski prekidi asinhrono

Tretman prekida

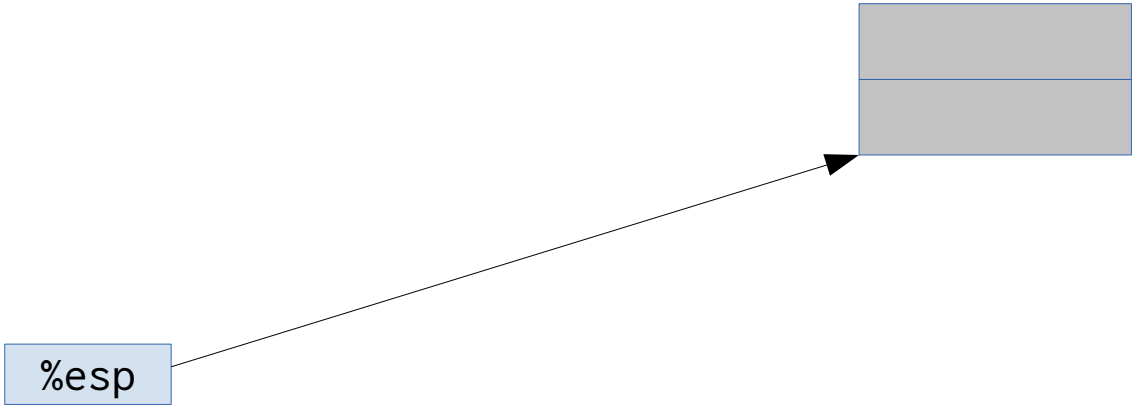
- Treba biti izvršen sa procesorom u prstenu 0
- Treba biti izvršen na tačno određenoj lokaciji unutar kod segmenta kernel-a
  - kod koji se izvršava prilikom tretmana specifičnog prekida naziva se ISR (Interrupt Service Routine)
- Mora uključivati snimanje stanja prekinutog procesa kako bi se isti mogao nastaviti naknadno, tj nakon tretmana prekida.
- Ne smije kompromitirati izolaciju adresnih prostora i sigurnost sistema

X86 arhitektura i prekidi

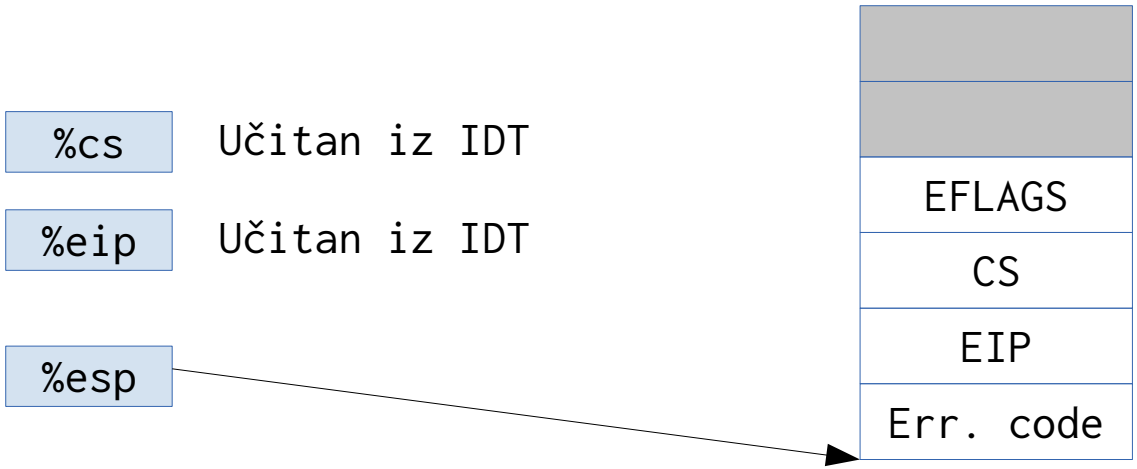
- Omogućava odabir segmenta i funkcije unutar odabranog segmenta za svaki tip prekida koji može nastati:
  - specifikacija se vrši putem IDT-a (Interrupt Descriptor Table)
  - maksimalno 255 različitih tipova prekida
    - Za svaki tip prekida, unutar tabele daje se vrijednost `%cs` i `%eip` registra koji će učitati iz tabele kada nastane dati prekid
- Omogućava specifikaciju vrijednosti `%ss` i `%esp` registra koji će biti učitani kada nastane bilo koji prekid
  - Specifikacija se vrši putem TSS-a (Task State Segment)
  - TSS je poseban segment unutar GDT-a

- Kada se dogodi prekid CPU snima vrijednosti posebnih registara na stek:
  - broj i tip snimljenih registra ovisi od prstena protekcije u kojem je CPU operirao prije nastanka prekida.
  - kada preuzme kontrolu a neposredno pred početak tretmana prekida, ISR snima ostale registre na stek pomoću instrukcije `pushall`
  - Nakon tremana iznimke prvo se sa steka na CPU vrate ostali registri sa `popall`
  - Tretman se završava kopiranjem prestalih registara sa steka na CPU instrukcijom `iret`

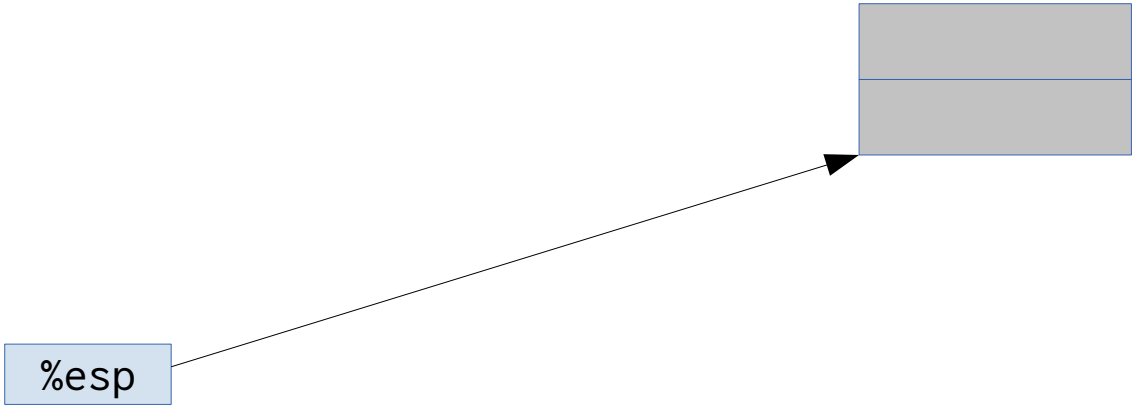
Stanje prije prekida → CPU u kernel modu



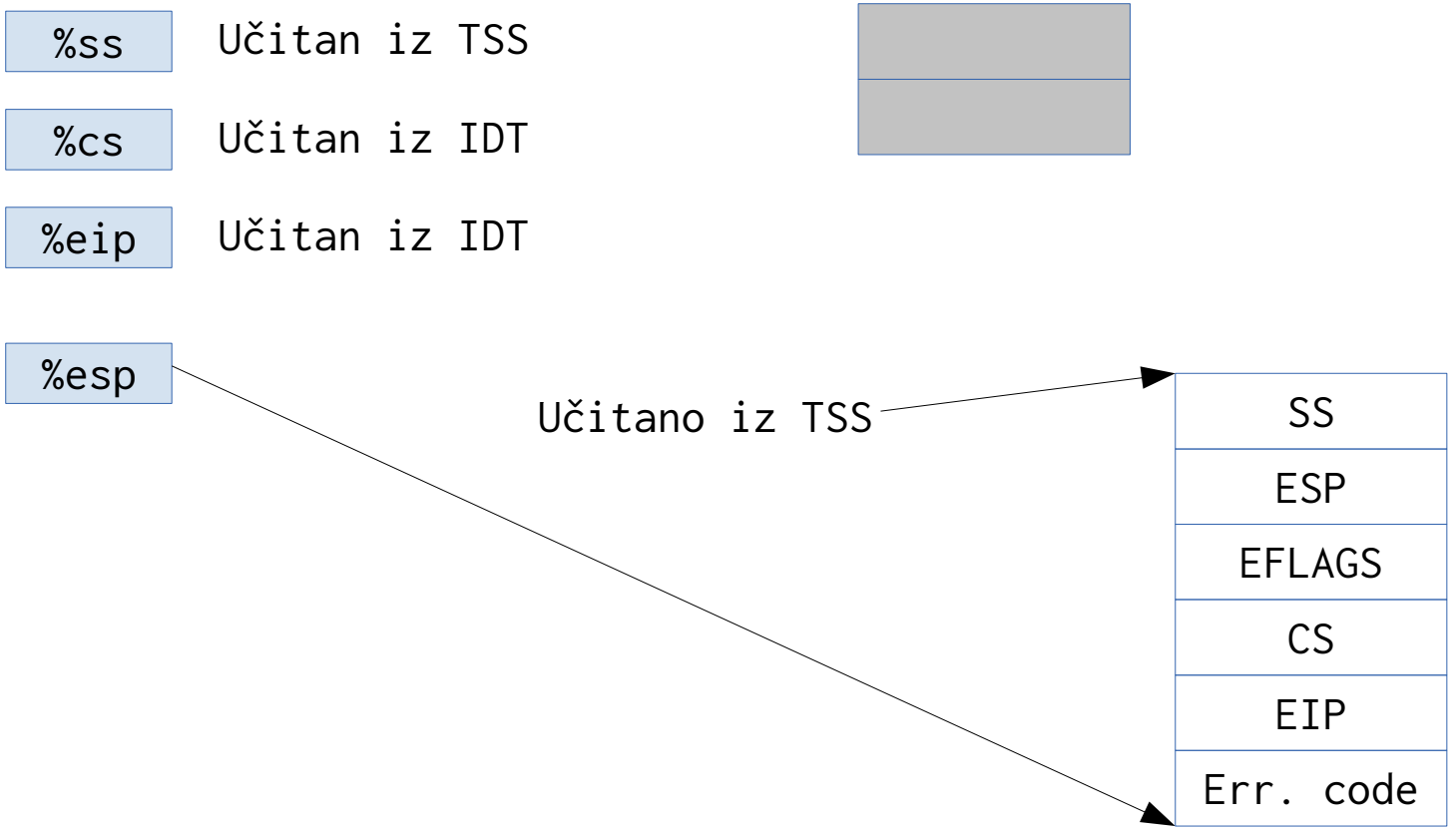
Stanje nakon prekida



Stanje prije prekida → CPU u user modu



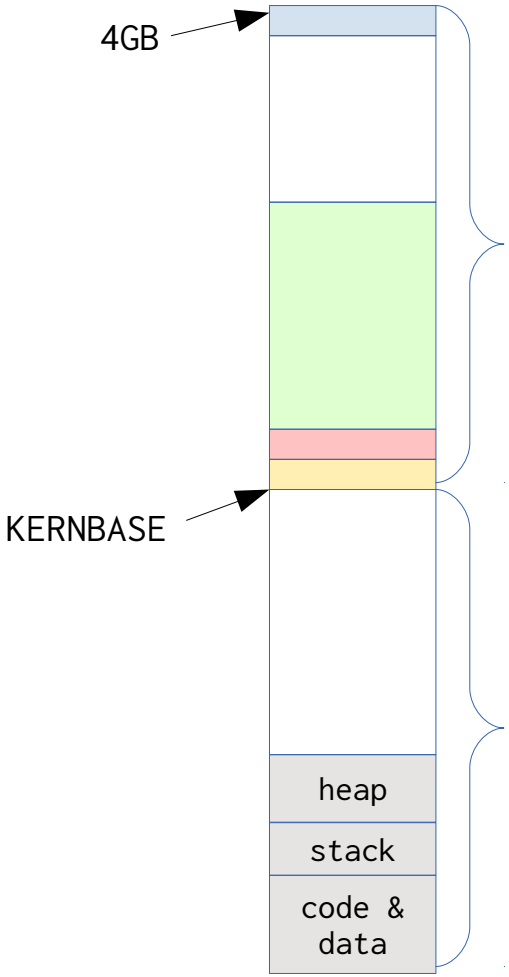
Stanje nakon prekida



xv6 i procesi

- `xv6` straničenjem definira po jedan virtuelni AP za svaki proces
  - tj svaki proces dobija svoj PDT
- U svakom virtuelnom AP-u kernel je mapiran iznad `KERNBASE` adrese
  - postavljanjem polja `U` na vrijednost `0` unutar odgovarajućih PTE-ova, adrese iznad `KERNBASE` dostupne su samo kada je CPU u prstenu protekcije `0`
- OS vodi računa o stanju izvršenja procesa alocirajući po jednu strukturu PCB (Process Control Block) za svaki proces
  - `xv6` kao PCB koristi strukturu pod imenom `proc`
  - `xv6` dozvoljava maksimalno `NPROC` procesa

- Za svaki proces `xv6` alocira po jednu fizičku stranicu koja se koristiti kao stek kada se dogodi prekid dok CPU izvršava kod datog procesa
  - tj svaki proces ima dva steka: kernel i user.
- PCB uključuje pointere na lokacije u memoriji na kernel steku gdje su snimljeni registri prilikom tretmana prekida nastalih dok se izvršavao dati proces
- Procesi tokom života mijenjaju stanja izvršenja koja se aktueliziraju unutar PCB-a
- Svi procesi kreiraju se putem posebnog sistemskog poziva `fork`
  - sem prvog procesa tzv `init` koji se kreira na poseban način unutar funkcije `userinit`, koja stvara iluziju za kreirani proces da je nastao pozivom fork
- Procesi se multipleksiraju u vremenu na svim raspoloživim jezgrama CPU-a unutar funkcije `scheduler`



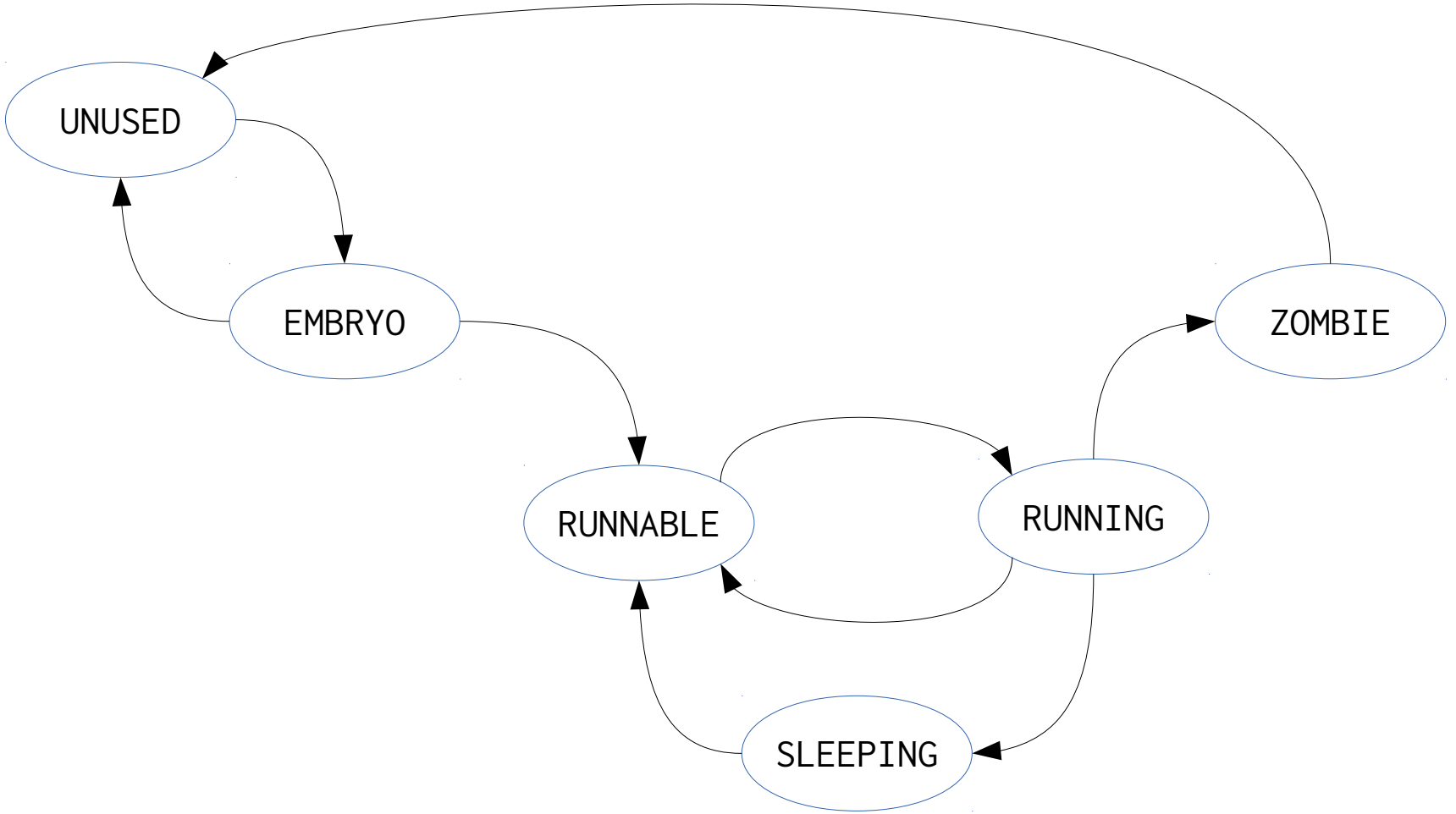

```
#define NPROC        64  // maximum number of processes

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

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

Dijagram promjene stanja procesa

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```



Inicijalizacija segment deskriptor tabele

```
void seginit(void)
{
  struct cpu *c;
  c = &cpus[cpunum()];

  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);


  c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);

  lgdt(c->gdt, sizeof(c->gdt));
  loadgs(SEG_KCPU << 3);

  cpu = c;
  proc = 0;
}
```

```
extern struct cpu *cpu asm("%gs:0");
extern struct proc *proc asm("%gs:4");
```

```
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_KCPU  3  // kernel per-cpu data
#define SEG_UCODE 4  // user code
#define SEG_UDATA 5  // user data+stack
#define SEG_TSS   6  // this process's task state
```

Inicijalizacija prvog procesa

```
void userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

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

  p->state = RUNNABLE;
}
```

```
static struct proc* allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

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

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

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

  return p;
}
```

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

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```