

# **Principi operativnih sistema kroz analizu XV6 koda**

---

*Amer Hasanović*

Tuzla, 2015

Amer Hasanović  
PRINCIPI OPERATIVNIH SISTEMA KROZ ANALIZU XV6 KODA

**Izdavač**

Izdavačka kuća Hamidović, Tuzla

**Za izdavača**

Rasim Hamidović

**Glavni redaktor**

Amer Hasanović

**Recenzenti**

Dr.sc. Damir Demirović, docent  
Fakultet elektrotehnike Univerziteta u Tuzli

Dr.sc. Emir Mešković, docent  
Fakultet elektrotehnike Univerziteta u Tuzli

**Tiraž**

200 primjeraka

CIP - Katalogizacija u publikaciji  
Nacionalna i univerzitetska biblioteka  
Bosne i Hercegovine, Sarajevo

004.451.9XV6(075.8)

HASANOVIĆ, Amer

Principi operativnih sistema kroz analizu XV6 koda / Amer Hasanović. -  
Tuzla : "Hamidović", 2015. - VII, 191 str. : graf. prikazi ; 24 cm

Bibliografija: str. 191.

ISBN 978-9958-833-08-3

COBISS.BH-ID 22494470

Odlukom Senata Univerziteta u Tuzli br. 03-5285-9.8/15 od 07.10.2015. godine, odobrena je upotreba udžbenika "Principi operativnih sistema kroz analizu XV6 koda", autora Amera Hasanovića, za potrebe izučavanja nastavnog predmeta "Operativni sistemi" na Fakultetu elektrotehnike Univerziteta u Tuzli.

Udžbenik "Principi operativnih sistema kroz analizu XV6 koda" je rezultat aktivnosti programa "Study of Excellence" u okviru projekta NORBOTECH, finansiranog od strane Ministarstva vanjskih poslova Kraljevine Norveške kroz HERD program, program za visoko obrazovanje, istraživanje i razvoj.

# Sadržaj

<i>Predgovor</i>	vii
<b>UVOD .....</b>	<b>1</b>
<b>Operativni sistem XV6</b>	
<b>Kreiranje XV6 programa</b>	
<b>OSNOVE INTEL ARHITEKTURE.....</b>	<b>9</b>
<b>IA32 naspram MIPS</b>	
<b>Registri i osnovne asembler instrukcije</b>	
<b>Funkcije i stek</b>	
<b>MODOVI OPERACIJE PROCESORA.....</b>	<b>27</b>
<b>Pristup memoriji</b>	
<b>Adresiranje u real modu</b>	
<b>Adresiranje u protected modu</b>	
<b>Protekcija</b>	
<b>KOMUNIKACIJA SA VANJSKIM UREĐAJIMA .....</b>	<b>49</b>
<b>Mapiranje vanjskih uređaja</b>	
<b>Prekidi</b>	
<b>Procesiranje prekida u protected modu</b>	
<b>UČITAVANJE KERNELA.....</b>	<b>73</b>
<b>Prva faza pokretanja sistema</b>	
<b>Inicijalna konfiguracija straničenja u kernelu</b>	

INICIJALIZACIJA KERNELA.....	93
Funkcija main	
Alokator stranica	
Kernel konfiguracija za straničenje	
Kernel konfiguracija za segmentiranje	
Pokretanje AP jezgri	
TRETMAN PREKIDA.....	113
Konfiguracija IDT tabele	
ISR rutine	
Struktura trapframe i procesiranje prekida	
SINHRONIZACIJA.....	125
Stanje utrke	
Brave	
Zastoji	
PROCESI.....	135
Struktura proc	
Implementacija sistemskih poziva	
Kreiranje procesa	
Promjena konteksta	
Uspavljivanje i buđenje procesa	
FAJL SISTEMI .....	169
Organizacija podataka na disku	
Fajl sistem organizacija u memoriji	
Fajl deskriptori	

## Predgovor

Knjiga je koncipirana tako da otkloni misteriju oko dizajna i implementacije ključnih komponenti koje čine operativni sistem. Za razliku od knjiga u kojim se daje pregled oblasti operativnih sistema navođenjem alternativnih pristupa u implementaciji, obično bez ulaska u detalje, ova knjiga se u potpunosti fokusira na praktičnu implementaciju operativnog sistema XV6, koji ima osnovne funkcije slične bilo kojem modernom Unix kompatibilnom operativnom sistemu.

Knjiga je namijenjena za studente studijskih programa računarstva i elektrotehnike ili za profesionalce koji imaju potrebu za razumijevanje implementacije ključnih komponenti sistemskog softvera. Od čitaoca se prepostavlja poznavanje programskog jezika C i asemblera za bilo koju procesorsku arhitekturu.

Tekst je organiziran u deset poglavlja. Uvodno poglavlje daje pregled osnovnih funkcija XV6 operativnog sistema i uvodi bitne termine iz oblasti operativnih sistema koji su detaljno objašnjeni u ostalim poglavlјima. Poglavlja 2–4 se fokusiraju na ključne elemente Intel hardverske arhitekture za koju je dizajniran i implementiran XV6 sistem. Proces pokretanja i inicijalizacije XV6 operativnog sistema predstavljen je u poglavlјima 5 i 6. Prekidi, sinhronizacija, procesi i fajl sistemi su tretirani u poglavlјima 7–10.

Nakon savladavanja materijala predstavljenog u knjizi, čitalac dobija prepostavke za razumijevanje kompleksnih operativnih sistema kao što su GNU/Linux ili BSD.

## Uvod

Moderan život uključuje svakodnevnu upotrebu velikog broja uređaja pri čemu posebno mjesto zauzimaju računari koje danas nalazimo u različitim formama, počev od velikih serverskih sistema, preko telefone, tableta, televizora, pa sve do pametnih satova. Čak i manje upućeni korisnici pri kupovini računara često postavljaju slijedeće pitanje: Koji operativni sistem se izvršava na tom računaru? Odgovor na postavljeno pitanje upućuje na broj aplikacija koje će korisnik moći instalirati na računar, kao i na okruženje u kojem će se koristiti instalirane aplikacije.

Pored stvaranja okruženja za izvršavanje aplikacija, operativni sistemi imaju i slijedeće zadatke:

- Upravljanje i komunikacija sa uređajima koji su povezani na računar.
- Apstrakcija različitih hardverskih uređaja kroz implementaciju unificiranih standardnih biblioteka koje se stavljaju na raspolaganje programima.
- Istovremeno izvršavanje proizvoljnog broja programa bez obzira na broj raspoloživih procesorskih jezgri.
- Rasporеđivanje računarskih resursa, a naročito procesora i memorije, između svih programa u izvršenju.
- Sprečavanje situacija u kojim greške prilikom izvršavanja jedne aplikacija mogu utjecati na izvršenje ostalih aplikacija.
- Formiranje fajl sistema u kojim programi na uređen način mogu trajno pohranjivati podatke.

Osnovna komponenta operativnog sistema u kojoj se implementiraju gore navedene funkcije naziva se *kernel* koji u najužem smislu predstavlja program koji se prvi učita u memoriju računara, u memoriji ostaje rezidentan sve do gašenja računara, a aktivira se po potrebi kada se zahtijeva neka od funkcija koju pruža. Za razliku od razvoja klasičnih programa, programiranje kornela zahtijeva intimno poznavanje: procesorske arhitekture, kompletног kompjuterskog lanca, kao i principa rada svih ključnih komponenti računara. Dodatno, kerneli modernih operativnih sistema, kao što su Linux<sup>1</sup> ili BSD<sup>2</sup>, spadaju u red najkompleksnijih sistema koji je dizajniran čovjek, obzirom da se sastoje od nekoliko desetina miliona linija koda napisanih u asembleru i programskom jeziku C.

## Operativni sistem XV6

Sa stanovišta izučavanja osnovnih funkcija operativnih sistema industrijski kerneli su suviše kompleksni, zbog čega se u ovom tekstu koristi poseban kernel pod imenom XV6, koji je razvijen u MIT<sup>3</sup> laboratoriji za paralelne i distribuirane sisteme. Kao Linux i BSD, XV6 je baziran na operativnom sistemu Unix koji je 80-tih godina razvijen u kompaniji AT&T, ali uz sljedeće razlike:

- XV6 je dizajniran za IA-32 procesorsku platformu.
- XV6 podržava više procesorskih jezgri.
- XV6 implementira sopstveni fajl sistem sa žurnalom.
- XV6 podržava jednog korisnika i nema mrežni stek.

Slično kao za originalni Unix, svaki program koji se izvršava pod kontrolom XV6 kernela dobija okruženje koje se naziva *proces*. Korisnik u bilo kojem trenutku može kreirati proizvoljan broj procesa, a XV6 kernel uz podršku hardvera za svaki proces osigurava vir-

---

1. <https://github.com/torvalds/linux>

2. <https://github.com/freebsd/freebsd>

3. <http://mit.edu>

tualni procesor i memoriju, čime program koji se izvršava unutar procesa dobija iluziju da u tom trenutku samostalno kontroliše dva ključna resursa računara. Ukoliko u nekom od procesa nastane greška, kernel intervenira na način da zatvara dati proces bez ometanja izvršavanja ostalih aktivnih procesa u sistemu.

XV6 operativni sistem i njegove aplikacije kompajliraju se na bilo kojoj Linux distribuciji uz prepostavku da je na datom sistemu instaliran kompajlerski lanac `gcc`, sistem za praćenje i kontrolu verzija `git` i program za emulaciju računara `qemu`. Na primjer, slijedeća komanda na Linux distribuciji Ubuntu instalira sve neophodne komponente za kompajliranje XV6 kernela i programa:

```
$ sudo apt-get install build-essential qemu-system-x86 git
```

Kompletan kod operativnog sistema XV6 može se preuzeti upotrebom programa `git` na slijedeći način:

```
$ git clone git://github.com/mit-pdos/xv6-public.git
```

Gornja komanda kreira direktorij `xv6-public` u kojem se, nakon preuzimanja sadržaja sa Interneta, nalazi kod od XV6 operativnog sistemskupa sa osnovnim aplikacijama. Kompajliranje bilo koje komponente XV6 sistema upravljan je putem `Makefile` skripte, koja se također nalazi u `xv6-public` direktoriju. Na primjer, slijedeća sekvenca komandi prvo kompajlira XV6 kernel i sve programe, a zatim priprema sistem za izvršenje u simuliranom okruženju koje pruža program `qemu`.

```
$ cd xv6-public/  
$ make
```

Mada podržava izvršavanje direktno na hardveru, zbog mogućnosti debagiranja i kontroliranja broja simuliranih jezgri, te zbog praktičnosti pokretanja, XV6 se primarno izvršava na simuliranom hardveru unutar okruženja kojeg obezbijeđuje program `qemu`.

Slijedeći primjer prikazuju proces pokretanja XV6 operativnog sistema, te interakciju sa korisnikom u XV6 komandnoj liniji.

```
moj@komp$ make qemu
xv6... ①
cpu1: starting ②
cpu0: starting ②
init: starting sh ③
$ ls ④
.
..
README      2 2 1972
cat          2 3 13320
echo         2 4 12428
forktest     2 5 8144
grep         2 6 15020
init         2 7 13084
kill          2 8 12580
ln            2 9 12424
ls            2 10 14812
mkdir        2 11 12572
rm            2 12 12560
sh            2 13 23564
stressfs     2 14 13280
usertests    2 15 58584
wc            2 16 13848
zombie       2 17 12196
console      3 18 0
$ ls | wc ⑤
19 76 474
$ ls > fajl.txt ⑥
$ wc < fajl.txt ⑦
20 80 498
```

Dok traje pokretanje sistema XV6 kernel ispisuje statusne poruke u linijama ①, ② i ③. Na osnovu ispisa u linijama ② zaključujemo da osnovna konfiguracija programa qemu simulira dva procesorska jezgra za XV6 kernel. Po okončanju kompletног procesa pokretanja operativnog sistema, XV6 kernel formira inicijalni proces u kojem se izvršava korisnički program pod imenom sh koji stvara komandno okuženje za kreiranje procesa.

Novi procesi se formiraju u komandnoj liniji imenovanjem programa koji će se početi izvršavati unutar kreiranog procesa. Program ls, pokrenut u liniji ④, prikazuje sadržaj trenutnog direktorija od procesa sh. Svi programi koji su kompajlirani uz XV6 kernel

operativnog  
liniji.

snimljeni su u root direktoriju XV6 fajl sistema, čiji je sadržaj prikazan nakon linije ④.

XV6 kernel implementira Unix koncept pod imenom pipe (cijev) putem kojeg izlaz jednog procesa može postati ulaz za drugi proces. Linija ⑤ prikazuje situaciju u kojoj je putem operatora “|”, ipis iz programa ls, umjesto na standardni izlaz kao u liniji ④, preusmjeren na ulaz drugog proces unutar kojeg se izvršava program wc koji ispisuje broj linija, riječi i karaktera preuzetih sa ulaza.

Putem operatora “<” i “>”, ulaz i izlaz procesa mogu se preusmjeriti u fajl. Na primjer, ispis programa ls u liniji ⑥ je putem operatora “>” preusmjeren u fajl pod imenom fajl.txt, koji je ujedno kreiran u trenutnom direktoriju, dok je ulaz za program wc u liniji ⑦ putem operatora “<” pročitan iz fajla fajl.txt. Broj linija koji je izračunao program wc uvećan je za jedan, obzirom da je u trenutni direktorij prethodnim komandama dodat novi fajl.

## Kreiranje XV6 programa

Izrada XV6 aplikacija uključuje sljedeća dva koraka:

1. Pisanje koda aplikacije u programskom jeziku C — pri čemu se mogu koristiti samo one funkcije koje su implementirane unutar standardne biblioteke XV6 kernela čije se deklaracije nalaze u fajlovima: types.h, stat.h, user.h i fcntl.h.
2. Dodavanje napisanog programa u Makefile skriptu — kako bi bio uključen u XV6 fajl sistem.

Prepostavimo da je unutar XV6 direktorija u fajlu mojprog.c napisan sljedeći C program.

```
// mojprog.c
#include "types.h" ①
#include "stat.h" ①
#include "user.h" ①
```

```

int main(int argc, char *argv[])
{
    int i=0, max=3;
    while ( i++ < max ) {
        sleep(400); ②
        printf(1,"i=%d\n",i); ③
    }
    exit(); ④
}

```

Putem linija ① program uključuje neophodna zaglavla u kojima su deklarirane funkcije `sleep`, `printf` i `exit`, koje se pozivaju u linijama ②, ③ i ④. Kada se pokrene unutar nekog procesa, program svake četiri sekunde ispisuje stanje brojača `i`, koji je inkrementiran u svakoj iteraciji petlje koja se ponavlja tri puta.

Da bi gornji program bio uključen u proces kompajliranja sa ostalim XV6 aplikacijama, te dobio svoje mjesto u XV6 fajl sistemu, `Makefile` skripta koja se nalazi u XV6 direktoriju mora biti promjenjena tako da varijabla `UPROGS` uključuje kreirani program na slijedeći način:

```

# Makefile
# ...
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _mojprog\ ①

#
# ...

```

Jedina izmjena unutar `Makefile` skripte napravljena je u liniji ①. Nakon ovog, putem komande `make qemu` izvršava se kompajliranje

nove aplikacije i neophodna izmjena XV6 fajl sistema, nakon čega se odmah nanovo pokreće XV6 operativni sistem u kojem je napravljena slijedeća komandna sesija:

```
$ ls
.
..
README          1 1 512
cat             2 2 1972
echo            2 4 12428
forktest        2 5 8144
grep            2 6 15020
init            2 7 13084
kill             2 8 12580
ln               2 9 12424
ls              2 10 14812
mkdir           2 11 12572
rm              2 12 12560
sh              2 13 23564
stressfs        2 14 13280
usertests       2 15 58584
wc              2 16 13848
zombie          2 17 12196
mojprog         2 18 12484 ①
console         3 19 0

$ mojprog ②
i=1
i=2
i=3
$ mojprog & ③
$ i=1
$ wc < README ④
49 278 1972 ⑤
i=2
i=3

$ 1 sleep init 80104d0b ... ⑥
2 sleep sh 80104d0b ...

$ mojprog > abc.txt & ⑦
$ 1 sleep init 80104d0b ... ⑧
2 sleep sh 80104d0b ...
10 sleep mojprog 80104d0b ... ⑨
$ cat abc.txt ⑩
i=1
i=2
i=3
$
```

Na osnovu ispisa sadržaja root direktorija u liniji ① konstatujemo da se nova aplikacija nalazi u fajl sistemu.

javlja u kojima pozivaju u li-cesa, program inkrementiran

ajliranja sa os- fajl sistemu, a biti promje-gram na slike-

a je u liniji ①. kompajliranje

Oglavlje 1, Uvod

Kreiranje XV6 programa

7

Po pokretanju aplikacije u liniji ❷, kao što je i očekivano, dobijamo ispis vrijednosti brojača i u svakoj iteraciji petlje. Obzirom da proces u kome se izvršava program sh čeka na završetak izvršenja procesa u kojem je program mojprog, tokom dvanaest sekundi, koliko traje izvršavanje programa, korisnik nema mogućnost unosa novih komandi u komandnoj liniji. Ovo ponašanje moguće je promjeniti upotrebom operatora “&” na kraju unesene sh komande, kao što je prikazano u liniji ❸. Od ovog trenutka dva procesa nastavljaju se izvršavati paralelno. Putem komandne linije, koja je sada u aktivnom stanju, u liniji ❹ pokreće se novi proces koji izvršava program wc. Ispis iz novog procesa pojavljuje se u liniji ❺, nakon čega slijedi ostatak ispisa iz programa mojprog.

Unosom u komandnu liniju kombinacije tipki `Ctrl` + `p`, kao što je urađeno u liniji ❻, korisnik u bilo kojem trenutku može dobiti listu XV6 aktivnih procesa. Na osnovu prikazane liste može se zaključiti da, pored procesa koji izvršava program sh, u trenutku formiranja liste postoji još jedan proces koji izvršava program init. Ovaj proces na Unix sistemima ima specijalnu namjenu koja će biti predmet analize narednih poglavljja.

Nakon što je u liniji ❼ ponovo pokrenut proces sa programom mojprog, ponovljen je prikaz liste aktivnih procesa u liniji ❽. Ispis u liniji ❾ potvrđuje da je proces koji izvršava program mojprog aktiviran, međutim, njegov ispis nije prikazan na standardnom izlazu obzirom da je preusmjeren u fajl abc.txt i to prilikom pokretanja programa u liniji ❼. Ovo je potvrđeno u liniji ❿ putem programa cat, koji na standardnom izlazu ispisuje sadržaj fajla čije je ime dato kao parametar pri pokretanju programa cat.

Prikazana funkcionalnost programa sh, a koju omogućava XV6 kernel, čini temelj bilo kojeg Unix kompatibilnog operativnog sistema. Međutim, implementacija ovakvog kernela je kompleksan proces koji će biti razmatran u narednim poglavljima ovog teksta.

## Osnove Intel arhitekture

Intel arhitektura (IA) je danas najčešće korištena arhitektura na personalnim računarima i serverskim sistemima. Vuče korijene iz 1976 godine kada je uspostavljena za 16-bitni Intel procesor pod oznakom 8086. Od tada, Intel je na tržište izbacio veliki broj procesora koji su na različite načine proširivali originalnu arhitekturu. Dvije varijante skupa instrukcija ove arhitekture (ISA - Instruction Set Architecture) danas se najčešće koriste prilikom kompajliranja programa radi distribucije u binarnom obliku, i to: *IA32* i *AMD64*.

IA32 je ISA za 32-bitne procesore u upotrebi od 1985 godine kada je Intel proizveo svoj prvi 32-bitni procesor pod oznakom 386. Alternativni naziv za ovu arhitekturu je *X86*. AMD64 je ISA za 64-bitne procesore nastala u kompaniji Advanced Micro Devices (AMD) koja je tek naknadno preuzeta od strane Intela. Naziv koji je još u upotrebi za ovu arhitekturu je *X86-64*. Svi moderni procesori proizvedeni u kompanijama Intel i AMD podržavaju obje ISA specifikacije, a mogu da rade i u 16-bitnom modu koji je kompatibilan sa izvornim 8086 procesorom.

Obzirom da je XV6 operativni sistem namijenjen za 32-bitne procesore u ovom poglavlju kratko ćemo se osvrnuti na IA32 arhitekturu prvenstveno poređenjem sa MIPS platformom. Detaljan tretman ove materije zahtijevao bi značajan prostor. Zbog toga ćemo se fokusirati na najčešće korištene instrukcije, kao i interpretaciju programa na nivou IA32 asemblera. Ovo predznanje neophodno je radi razumijevanja rada operativnih sistema, a još više radi mogućnosti njihove implementacije.

Opš  
O  
i

Segm  
Š  
ri

Kont  
P  
đ

FLAG  
Je  
P

IP  
Je  
lo

FPU  
C  
al

R  
P  
ovih  
prev  
ebp i  
vira

S  
ime  
truk  
truk

## IA32 naspram MIPS

X86 je arhitektura koju odlikuje veliki broje instrukcija i mali broj registara opšte namjene. Za razliku od MIPS arhitekture koja je kreirana spram RISC (Reduced Instruction Set Computer) filozofije, X86 spada u CISC (Complex Instruction Set Computer) arhitekturu. Zbog toga postoje bitne razlike između MIPS i Intel procesora koje se mogu sumirati u slijedećim konstatacijama koje važe za X86 procesore:

- Instrukcije najčešće koriste dva operanda.
- Većina instrukcija podržava jedan operand iz memorije.
- Mašinske instrukcije su varijabilne dužine, i to od 1 do 17 bajta.
- Pristup memoriji ne mora biti poravnat na određene adrese.
- Procesor radi samo u malom endijan modu.

Nadalje, za Intel arhitekturu postoje dvije sintakse asembler jezika pod nazivima *AT&T* i *Intel*. Suštinska razlika je u načinu formiranja instrukcija i načinu interpretiranja redoslijeda operanda, što je dovoljno da ove dvije notacije rezultiraju nekompatibilnim asembler kodom.

Kompajlerski lanac otvorenog koda GCC koristi AT&T notaciju, dok Microsoft kompjajler koristi Intel notaciju. Obzirom da se XV6 operativni sistem kompjajlira sa GCC kompjajlerskim lancem, svi primjeri u ovom poglavlju odnose se na AT&T notaciju.

## Registri i osnovne asembler instrukcije

Registre u Intel procesorima možemo klasificirati u sljedećih šest kategorija:

### **Opšti**

Osam 32-bitnih registara oznaka: eax, ebx, ecx, edx, esi, edi, esp i ebp, koji se koriste za pohranu i procesiranje podataka.

### **Segmentni**

Šest 16-bitnih registara oznaka: cs, ds, ss, es, fs i gs, koji se koriste prilikom pristupanja memoriji.

### **Kontrolni**

Pet 32-bitnih registara oznaka: cr0, cr1, cr2, cr3 i cr4, koji određuju trenutni mod operacije procesora.

### **FLAGS**

Jedan 32-bitni registar oznake eflags, koji sadrži status izvršenja posljednje instrukcije.

### **IP**

Jedan 32-bitni registar oznake eip, koji uvijek sadrži memoriju lokaciju instrukcije koja se izvršava.

### **FPU**

Osam 64-bitnih registara za procesiranje podataka u domenu realnih brojeva.

Registri opšte namjene prikazani su na slici 2-1.

Programi u svakom trenutku mogu slobodno mijenjati sadržaj ovih registara. Iako spadaju u red registara opšte namjene, prilikom prevođenja programa napisanih u programskom jeziku C, registri ebp i esp koriste se kao pointeri na početak, tj. kraj aktivacijskog okvira funkcije koja se trenutno izvršava.

Svi registri su 32-bitni i u asembler instrukcijama označavaju se imenom kojem prethodi karakter %. Ukoliko se u određenoj instrukciji u imenu bilo kojeg registra izostavi karakter e, tada se ta instrukcija obavlja na donjoj riječi tj. prvih 16 bita datog registra.

trukcije  
trukciji  
dataku

Kao  
tantu, il  
memori  
vor i od

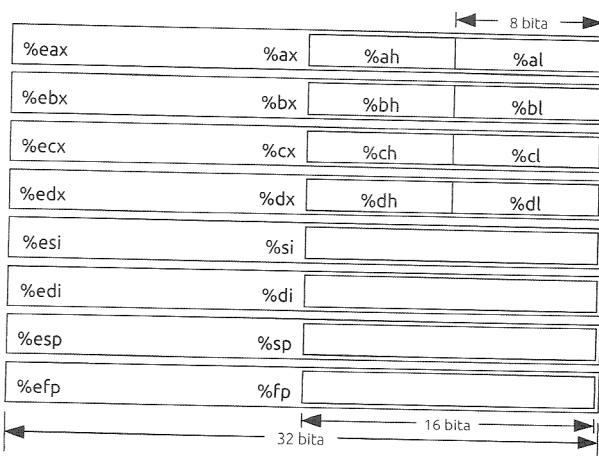
Činje  
predstav  
mo spec  
ad ili sto

U pr  
sfera po  
je, odre

- ka
- ka
- ka

Većin  
odrediš  
transfera

Sve tr  
izvor po  
tu koji je  
nog form  
za specifi  
cimalni b  
šteni for  
karakter  
te kao op



Slika 2-1. IA32 registri opšte namjene

Nadalje, za registre a, b, c i d, u instrukcijama je moguće manipulirati pojedinačnim bajtima donje riječi registra, dodavanjem karaktera l za donji bajt, ili h za gornji bajt u riječi.

U svakoj od slijedećih linija asembler programa, instrukcijom mov djelimično ili u potpunosti mijenja se sadržaj registra eax:

```
movl $0xffffffff, %eax ①
movw $0xaaaa, %ax   ②
movb $0xbb, %ah ③
```

- ① Postavlja kompletan sadržaj registra na vrijednost 0xffffffff.
- ② Mijenja samo donja dva bajta registra.
- ③ Modificira samo drugi bajt tako da je konačni sadržaj registra 0xffffbbaa.

Analizom prethodnog primjera možemo utvrditi neke bitne karakteristike asemblera koda za X86 arhitekturu. Za razliku od MIPS platforme gdje instrukcije najčešće imaju tri operanda, IA32 ins-

trukcije obično su u formatu od dva operanda. Prvi operand u instrukciji predstavlja izvor za podatke, a drugi odredište u koji se podatak upisuje po izvršenju instrukcije.

Kao izvor za većinu instrukcija možemo koristiti: registar, konstantu, ili memoriju. Za odredište na raspolaganju imamo registar ili memoriju. Nije moguće u istoj instrukciji koristiti memoriju kao izvor i odredište.

Činjenica da većina instrukcija prihvata memoriju kao operand predstavlja značajnu razliku u odnosu na MIPS platformu, gdje samo specijalizirane instrukcije za učitavanje i snimanje podataka (load ili store) mogu komunicirati sa memorijom.

U prethodnom primjeru koristili smo instrukciju `mov` radi transfera podataka, pri čemu sufiks, tj. karakter dodat na kraj instrukcije, određuje tačnu količinu podataka koju prenosimo, i to:

- karakter `b` (byte) za transfer od 8 bita,
- karakter `w` (word) za transfer od 16 bita,
- karakter `l` (long) za transfer od 32 bita.

Većina asembler instrukcija podržava sva tri sufiksa, pri čemu odredište kapacitetom mora tačno odgovarati veličini zahtijevanog transfera, u suprotnom asembler prijavljuje grešku ili upozorenje.

Sve tri instrukcije iz prethodnog primjera koriste konstantu kao izvor podataka. Konstante su ovdje date u heksadecimalnom formatu koji je nagovješten upotrebom prefiksa `0x`. Pored heksadecimalnog formata možemo koristiti bilo koji format koji bi bio validan za specifikaciju konstanti u programskom jeziku C (npr. `$-10` za decimalni broj deset sa negativnim predznakom). Bez obzira na korišteni format, prije vrijednosti konstante obavezno moramo dodati karakter `$`. Izostavljanjem ovog prefiksa umjesto brojčane konstante kao operand bi dali adresu memorijske lokacije.

Pored pristupa memoriji specifikacijom absolutne adrese u obliku konstante bez prefiksa \$, asembler podržava dodatnih osam formata izraza za pristup memoriji. Svi podržani formati prikazani su u slijedećoj tabeli:

**Tabela 2-1.** Formati izraza za pristup memoriji

Izraz	Rezultat
$\text{Imm}$	$M[\text{Imm}]$
$(\text{Ea})$	$M[\text{Ea}]$
$\text{Imm}(\text{Eb})$	$M[\text{Imm}+\text{Eb}]$
$(\text{Eb}, \text{Ei})$	$M[\text{Eb}+\text{Ei}]$
$\text{Imm}(\text{Eb}, \text{Ei})$	$M[\text{Imm}+\text{Eb}+\text{Ei}]$
$(, \text{Ei}, s)$	$M[\text{Ei}*s]$
$\text{Imm}(, \text{Ei}, s)$	$M[\text{Imm}+\text{Ei}*s]$
$(\text{Eb}, \text{Ei}, s)$	$M[\text{Eb}+\text{Ei}*s]$
$\text{Imm}(\text{Eb}, \text{Ei}, s)$	$M[\text{Imm}+\text{Eb}+\text{Ei}*s]$

Gdje su:

- $\text{Ea}$  i  $\text{Eb}$  — bilo koji opšti registri,
- $\text{Imm}$  — neka konstanta,
- $s$  — faktor skaliranja: 2, 4 ili 8,
- $M[X]$  — predstavlja operaciju preuzimanja podataka iz memorije, i to sa adresu X.

Izrazi iz tabele mogu se koristiti kao izvor ili odredište u asembler instrukcijama. Količina podataka preuzetih iz memorije, ili upi-

adrese u obliku osam bitova prikazani su

sanih u memoriju, ovisi od sufiksa (b, w ili l) koji se koristi u instrukciji. Npr.:

```
movb 18(%eax,%ebx), %cl
```

Nakon izvršenja gornje instrukcije, u najnižih osam bita registra ecx će biti upisan jedan bajt pročitan iz memorije sa adrese koja je izračunata kao suma broja 18 i dvije vrijednosti pročitane iz registara eax i ebx.

Adrese koje se dobiju računanjem bilo kojeg od izraza za pristup memoriji ne moraju biti poravnate kao što je slučaj za MIPS platformu. Ipak, kompjuler u generiranom asembleru kodu direktivom .align vrši poravnavanje podataka na određene adrese, prvenstveno radi optimizacije pristupa memoriji.

Ukoliko se izraz za pristup memoriji koristi u instrukciji lea, tada se računa samo adresa. Kompajler ovu instrukciju često koristi u aritmetičkim izrazima koji nemaju nikakvu vezu sa pristupom memoriji. Instrukcija prihvata sufikse w i l, tj. vrši račun u domenu 16-bitnih i 32-bitnih brojeva. Npr.:

```
leal (%eax,%ebx,8), %ecx
```

Gornja instrukcija 32-bitnu vrijednost dobivenu kao rezultat operacije izraza  $%eax+8*%ebx$  upisuje u registar ecx.

Pored instrukcije mov, za situacije gdje je potrebno izvršiti transfer podataka sa proširivanjem broja bita mogu se koristiti instrukcije:

- movz → za proširivanje bez zadržavanja predznaka,
- movs → za proširivanje sa zadržavanjem predznaka.

Obje instrukcije imaju dva karaktera kao sufiks. Prvi karakter sufksa predstavlja broj bita u izvoru i može biti b ili w, dok drugi karakter predstavlja broj bita u odredištu i može biti w ili l. Npr.:

```
movzbl 4(%eax), %ebx ①  
movswl %ax, %ecx ②
```

Vrši transfer jednog bajta iz memorije sa adresi  $4 + \%eax$  u naj-

- ① niži bajt registra ebx. Gornja tri bajta registra ebx popunjavaju se nulama.

Vrši transfer riječi preuzete iz registra ax u donju riječ registra ecx. Kompletna gornja riječ registra ecx popunjava se na

- ② osnovu predznaka vrijednosti registra ax, tj. ekstenzijom naj-višeg bita.

Slijedeća tabela daje sumarni pregled često korištenih asembler instrukcija, pri čemu su dozvoljeni sufiksi za svaku instrukciju dati unutar zagrada {}:

Tabela 2-2. Aritmetičko-logičke instrukcije

Instrukcija	Interpretacija
sal{b,w,l} s, d	$d = d \ll s$
sar{b,w,l} s, d	$d = d \gg s$ (predznak sačuvan)
shl{b,w,l} s, d	$d = d \ll s$ (isto kao sal)
shr{b,w,l} s, d	$d = d \gg s$ (prazna mjesta 0)
and{b,w,l} s, d	$d = d \& s$
orl{b,w,l} s, d	$d = d   s$
xor{b,w,l} s, d	$d = d ^ s$

Instrukcija	Interpretacija
not{b,w,l} d	d = ~d
inc(b,w,l) d	d = d + 1
dec(b,w,l) d	d = d - 1
neg(b,w,l) d	d = ^d
add{b,w,l} s, d	d = s + d
sub{b,w,l} s, d	d = d - s
mulb s	%ax = %al * s;
mulw s	%dx:%ax = %ax * s;
mult s	%edx:%eax = %eax * s;
imul{b,l,w} s	isto kao mult, samo sa predznakom
imul{b,l,w} s, d	d = d * s;
cmp{b,w,l} s1, s2	s2 - s1 (mijenja stanje eflags)
test{b,w,l} s1, s2	s2 & s1 (mijenja stanje eflags)

Većina instrukcija u prethodnoj tabeli tokom izvršenja, pored promjene sadržaja odredišnog registra, vrši promjenu i posebnih bita, tzv. statusnih zastavica, u registru EFLAGS. Slijedeći biti ovog registra nose informaciju o statusu izvršenja posljednje instrukcije:

- bit 0 — Carry flag (CF)
- bit 2 — Parity flag (PF)
- bit 7 — Sign flag (SF)
- bit 6 — Zero flag (ZF)
- bit 11 — Overflow flag (OF)

Aritmetičko-logičke instrukcije na različite načine mijenjaju navedene statusne zastavice, npr. instrukcija add računa vrijednost iz-

raza  $rez = a + b$ , ali istovremeno postavlja i statusne zastavice u skladu sa vrijednostima slijedećih izraza:

1. CF  $\rightarrow rez < a$
2. ZF  $\rightarrow rez == 0$
3. SF  $\rightarrow rez < 0$
4. OF  $\rightarrow (a < 0 == b < 0) \&& (rez < 0 != a < 0)$

Promjenu toka IA32 programa moguće je izvršiti bezuslovno, ili na osnovu trenutnih vrijednosti statusnih zastavica registra EFLAGS. Za uslovnu promjenu toka postoji veliki broj instrukcija koje vrše grananja na osnovu vrijednosti jedne statusne zastavice, ili izraza koji uključuje vrijednost više statusnih zastavica. Slijedeća tabela daje pregled često korištenih instrukcija za promjenu toka programa:

Tabela 2-3. Instrukcije za grananje

Instrukcija	Opis	Statusne Zastavice
JA	Jump if above	CF=0 i ZF=0
JAE	Jump if above or equal	CF=0
JB	Jump if below	CF=1
JBE	Jump if below or equal	CF=1 ili ZF=1
JC	Jump if carry	CF=1
JG	Jump if greater	ZF=0 i SF=OF
JGE	Jump if greater or equal	SF=OF
JL	Jump if less	SF<>OF
JLE	Jump if less or equal	ZF=1 ili SF<>OF
JNA	Jump if not above	CF=1 ili ZF=1
JNAE	Jump if not above or equal	CF=1

bezuslovno, ili registra EFLAGS. Instrukcija koje vrše zastavice, ili izraza navedeća tabela daje tok programa:

Instrukcija	Opis	Statusne Zastavice
JNB	Jump if not below	CF=0
JNBE	Jump if not below or equal	CF=0 i ZF=0
JNC	Jump if not carry	CF=0
JNE	Jump if not equal	ZF=0
JNG	Jump if not greater	ZF=1 ili SF<>OF
JNGE	Jump if not greater or equal	SF<>OF
JNL	Jump if not less	SF=OF
JNLE	Jump if not less or equal	ZF=0 i SF=OF
JNO	Jump if not overflow	OF=0
JNP	Jump if not parity	PF=0
JNS	Jump if not sign	SF=0
JNZ	Jump if not zero	ZF=0
JO	Jump if overflow	OF=1
JP	Jump if parity	PF=1
JPE	Jump if parity even	PF=1
JPO	Jump if parity odd	PF=0
JS	Jump if sign	SF=1
JZ	Jump if zero	ZF=1
JMP	Jump	

Primjenom instrukcija iz prethodne tabele neposredno nakon odgovarajuće aritmetičko-logičke instrukcije, najčešće `cmp` ili `test`, moguće je u IA32 asembleru implementirati bilo koju sekvencu grananja i petlji iz programskog jezika C.

Pretpostavimo da je dat slijedeći segment C koda:

```
int b;
if ( a > 10 )
    b = 5;
else
    b = 1;
```

Pod pretpostavkom da se varijabla **b** nalazi u registru **eax**, a varijabla **a** u registru **ebx**, gornji segment C programa može se napisati u obliku slijedećeg IA32 asembler koda:

```
cmpl    $10, %ebx  ①
jle     .L2  ②
movl    $5, %eax
jmp     .L3  ③
.L2:
    movl    $1, %eax
.L3:
```

① Vrši se poređenje broja **10** sa vrijednosti varijable **a** u registru **ebx**.

② Za slučaj da je **a** manje ili jednako od **10**: preskaču se slijedeće dvije instrukcije do oznake **.L2**, postavlja se varijabla **b** na **1**, a zatim nastavlja izvršenje programa.

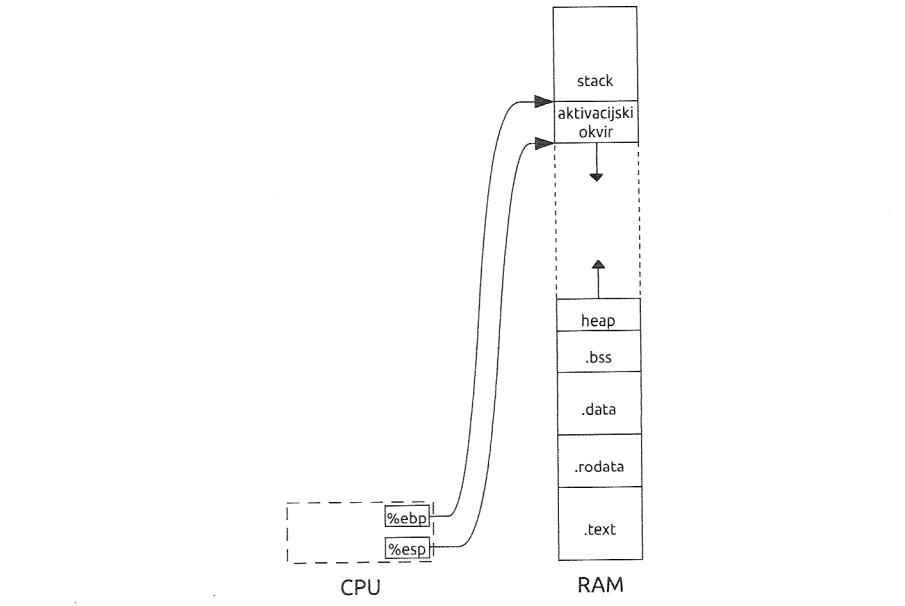
③ Nakon što se utvrdi da je **a** veće od **10**, te se varijabla **b** postavi na **5**, vrši se bezuslovan skok kojim se preskače jedna instrukcija do oznake **.L3**, čime se nastavlja izvršenje programa.

## Funkcije i stek

IA32 asembler funkcije imenuju se putem oznaka koje se postavljaju na lokaciji u asembler kodu od koje počinje prva instrukcija u funkciji. Za potrebe lokalnih varijabli funkcije mogu koristiti prostor u stek segmentu memorije, i to na način da se po pozivu funkcije potrebbni prostor alocira na steku, a neposredno prije povratka iz funkcije izvrši dealokacija prostora na steku kojeg je koristila funkcija. Prostor koji funkcija alocira na steku naziva se aktivacijski okvir.

da:

egistru eax, a vari-  
može se napisati u



Slika 2-2. Memorijska slika programa

Kao što je prikazano na slici 2-2, stek segment programa počinje od vrha adresnog prostora i raste prema nižim adresama. Trenutni vrh steka prati se u registru esp. Vrh steka je u isto vrijeme i adresa kraja aktivacijskog okvira funkcije čiji se kod trenutno izvršava. Adresa početka ovog aktivacijskog okvira bilježi se u registru ebp.

Za proširivanje steka obično se koristi instrukcija `push{w, l}`. Obrnuta operacija, tj. smanjenje steka, obavlja se putem operacije `pop{w,l}`. Instrukcija `push` kao parametar može uzeti konstantu ili registar, dok `pop`, uzima kao parametar samo registar.

Instrukcija `push{w,l}` u ovisnosti od korištenog sufiksa prvo smanjuje vrijednost registra esp za 2 ili 4, čime se alocira potrebni prostor na steku. Nakon toga, procesor na adresi u memoriji u skladu sa novom vrijednosti registra esp učitava 2 ili 4 bajta pročitana iz parametar proslijedenog instrukciji `push`. Na primjer, instrukcija `pushl %eax` ima isto značenje kao slijedeće dvije instrukcije:

koje se postavljaju  
instrukcija u funk-  
koristiti prostor u  
ozivu funkcije po-  
e povratka iz funk-  
e koristila funkcija.  
aktivacijski okvir.

```
subl $4, %esp  
movl %eax, (%esp)
```

Instrukcija `pop{w,l}` ima obrnut efekat u odnosu na instrukciju `push`. Ova instrukcija iz memorije sa adresi asocirane sa registrom `esp` prvo čita 2 ili 4 bajta, nakon čega pročitanu vrijednost upisuje u odredišni registar, a zatim inkrementira vrijednost u registru `esp` za 2 ili 4. Na primjer, instrukcija `popw %ax` ima ekvivalent u slijedećoj sekvenci instrukcija:

```
movw (%esp), %ax  
addl $2, %esp
```

Instrukcijom `pushad` stek je moguće proširiti i sa više vrijednosti koje su istovremeno preuzete iz registara: `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi` i `edi`. Obrnuta operacija obavlja se instrukcijom `popad`, tj. čitaju se vrijednosti za sve navedene registre sa steka. Za slučaj kada se koriste instrukcije `pusha` i `popa`, događaju se iste promjene na steku kao sa instrukcijama `pushad` i `popad`, samo sa 16-bitnim vrijednostima navedenih registara.

Registar `EFLAGS`, snima se na stek instrukcijom `pushfl`, a popunjava sa vrijednosti pročitanom sa steka putem instrukcije `popfl`.

Pored prostora za lokalne varijable, IA32 platforma stek koristi i kao isključivi način za proslijedivanje argumenata za funkcije. Ne-posredno prije poziva funkcije, argumente za funkciju potrebno je postaviti na stek, pri čemu se prvi argument postavlja se na stek posljednji.

Sam poziv funkcije obavlja se putem instrukcije `call` koja kao parametar očekuje ime funkcije, tj. asembler označku kojom je markirana prva instrukcija od koje počinje kod date funkcije. Povratna adresa iz funkcije za programski brojač `eip` snima se na steku ne-posredno ispod argumenata funkcije. Procesor u toku izvršenja instrukcije `call` proširuje stek sa povratnom vrijednošću za program-

ski brojač. Nakon čega u registar eip postavlja adresu prve instrukcije pozvane funkcije.

Povrat na lokaciju u kodu odakle je funkcija pozvana, i to jednu instrukciju nakon instrukcije `call`, obavlja se putem instrukcije `ret` koja ne uzima parametre. Instrukcija `ret` čita sa steka povratnu adresu koju je pozivom funkcije ostavila instrukcija `call`, a zatim pročitanu vrijednost upisuje u registar `eip`. Očigledno da prije poziva `ret`, stek mora biti vraćen u isto stanje u kojem je bio kad je počelo izvršenje pozvane funkcije.

Putem registra `eax`, kodu koji je izvršio poziv, funkcija može da vrati maksimalno jednu vrijednost.

Neka je dat slijedeći C program:

```
#include <stdio.h>
int main() {
    printf("par2 = %d\n", 5);
    return 0;
}
```

Prethodni program moguće je napisati u obliku IA32 asemblera:

```
.section .rodata
format:
    .asciz "par2 = %d\n" ①
.text
.globl main
main:
    pushl $5 ②
    pushl $format ③
    call printf ④
    addl $8, %esp ⑤
    movl $0, %eax ⑥
    ret ⑦
```

- ① Definira se prostor u `rodata` sekciji programa za prvi parametar, tzv. format string funkcije `printf`.

na instrukciju  
irane sa registrom  
rijednost upisuje u  
ost u registru esp za  
ivalent u slijedećoj

sa više vrijednosti  
ecx, edx, ebx, esp,  
strukcijom popad, tj.  
steka. Za slučaj ka-  
se iste promjene na  
a 16-bitnim vrijed-

om `pushfl`, a popu-  
nstrukcije `popfl`.

forma stek koristi i  
ata za funkcije. Ne-  
unkciju potrebno je  
tavljaju se na stek po-

je `call` koja kao pa-  
ku kojom je marki-  
funkcije. Povratna  
ma se na steku ne-  
toku izvršenja ins-  
inošću za program-

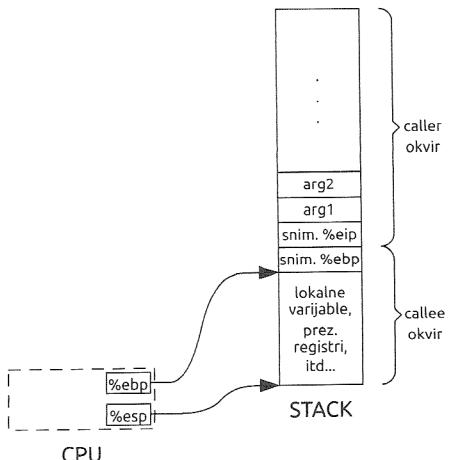
- ② Na stek se postavlja drugi argument za poziv funkcije, tj. broj 5.
- ③ Na stek se postavlja prvi argument za poziv funkcije, tj. adresu u memoriji odakle počinje prvi bajt asociran sa oznakom **format**.
- ④ Poziva se funkcija **printf**.  
Po povratku iz funkcije **printf**, stek se vraća na stanje prije poziva funkcije **main**, obzirom da je u funkciji **main** dva puta proširen stek sa **pushl**.
- ⑤ Postavlja se povratna vrijednost iz funkcije **main**.
- ⑥ Vrši se povrat iz funkcije **main** na onu lokaciju u kodu odakle je funkcija pozvana.

Obzirom na uloge u programu, **main** iz prethodnog primjera označava se kao funkcija koja poziva, tzv. *caller*, dok se **printf** označava kao funkcija koja je pozvana, tzv. *callee*. U interakciji između funkcije koja poziva i pozvane funkcije postoje određene konvencije. Pozvana funkcija slobodno može da mijenja sadržaj registara: **eax**, **ecx** i **edx**. Ukoliko mijenja neki od *prezerviranih* registara: **ebx**, **esi**, **edi**, **ebp** i **esp**, pozvana funkcija mora vratiti registar u originalno stanje prije instrukcije **ret**. Slika 2-3 prikazuje izgled stek segmenta memorije u kojem se nalaze aktivacijski okviri pozvane funkcije i funkcije koja poziva, i to u trenutku dok se izvršava pozvana funkcija.

Uzimajući u obzir sliku 2-3, vidljivo je da bi svaka funkcija trebala početi sa slijedećim segmentom koda, tzv. *prolog*:

```
pushl %ebp
movl %esp, %ebp
```

Nakon prologa, funkcija obično alocira sav potrebnii prostor za svoj aktivacijski okvir odgovarajućom promjenom registra **esp**. Tek



Slika 2-3. Aktivacijski okviri funkcija

nakon ovih operacija može početi kod koji implementira operacije iz funkcije.

Funkcija se završava slijedećom sekvencom koda, tzv. *epilog*.

```
movl %ebp, %esp
popl %ebp
ret
```

Epilogom se efektivno dealocira okvir funkcije koja je pozvana, a procesor vraća na aktivacijski okvir funkcije koje je vršila pozivanje.

Intel pr

### Real

U ov  
ima 2  
prote  
trimat

### Protected

U ov  
ima 3.  
i stran  
svih re

Kontr  
ži inform  
no. Ukol  
protynom  
tra cr0 m  
u protect  
transfer m

### Pristup m

Svaki pris  
cesora, m  
ra: cs, ds,  
jednost k

Pristup me

## Modovi operacije procesora

Intel procesori mogu operirati u jednom od dva operativna moda:

### Real

U ovom modu procesor je kompatibilan sa 8086 procesorom, ima 20-bitnu adresnu magistralu i nema nikakve mogućnosti za protekciju. Dozvoljeno je korištenje samo donjih 16-bitova u registrima.

### Protected

U ovom modu procesor je kompatibilan sa 386 procesorom, ima 32-bitnu adresnu magistralu i ima mogućnost za protekciju i stranicenje. Dozvoljeno je korištenje potpune 32-bitne širine svih registara.

Kontrolni register `CR0` u najnižem bitu, označenom kao PE, sadrži informaciju o modu operacije u kojem se procesor nalazi trenutno. Ukoliko bit PE ima vrijednost 0 procesor je u real modu, u suprotnom, procesor je u protected modu operacije. Vrijednost registra `CR0` može se mijenjati sa instrukcijom `Mov`. Transfer iz real moda u protected mod moguće je izvršiti u bilo kojem trenutku. Obrnuti transfer nije moguć bez resetovanja procesora.

### Pristup memoriji

Svaki pristup memoriji, bez obzira na trenutni mod operacije procesora, mora uključivati jedan od slijedećih šest segmentnih registara: `CS`, `DS`, `SS`, `ES`, `FS` ili `GS`. Ovi registri su 16-bitni, i mogu dobiti vrijednost kopiranjem iz opštih registara putem instrukcije `Mov`, ili sa

steka putem instrukcije `pop`. Ovo jedino ne važi za registar `cs` čija se vrijednost može eksplisitno postaviti putem posebno formatiranih instrukcija `jmp` i `call`. Instrukcije označene kao `ljmp` i `lcall`, pored određene vrijednosti za `eip`, uključuju i novu vrijednost za registar `cs`.

Već ranije su naglašeni dozvoljeni izrazi za pristup memoriji u asembler kodu. Informacije o segmentnom registru mogu biti izostavljene u ovim izrazima, tada asembler vrši odabir segmentnog registra koji će biti korišten za formiranje adrese.

Svaki izraz može uključivati i eksplisitno odabrani segmentni registar i to u formatu `segment:izraz`. Na primjer:

```
movw %ax, %ss:12(%esp) ①  
movb %ds:0xa0, %bh ②
```

Prilikom formiranja adrese za transfer dva bajta iz 16-bitnog   
① registra `ax` u memoriju koristit će se vrijednost izraza `12+%esp` i segmentni registar `ss`.

Prilikom formiranja adrese za transfer jednog bajta iz memorije u 8-bitni registar `bh` koristit će se vrijednost `0xa0` i segmentni registar `ds`.

Ukoliko se u izrazu za pristup memoriji izostavi segmentni registar, asembler će sam odabratи registar, i to:

- `cs` → Ukoliko se pristupa `text` segmentu programa, tj koriste instrukcije za kontrolu toka programa npr.: `jmp`, `call` itd.
- `ss` → Ukoliko se pristupa `stack` segmentu programa, tj koriste instrukcije koje uključuju `esp` registar npr.: `push`, `pop`, `mov` sa `%esp` itd.
- `ds` → Ukoliko se pristupa `data`, `rodata` ili `bss` segmentima programa, tj koriste instrukcije za snimanje i učitavanje podataka.

Fundamentalna razlika između real i protected moda operacije je u načinu na koji procesor koristi informaciju iz odabranog segmentnog registra prilikom formiranja adrese za pristup memoriji.

## Adresiranje u real modu

Svaki moderni Intel procesor kada dobije napon aktivira se u 8086 kompatibilnom modu, tj. u real modu.

Ukoliko se u nekoj instrukciji pristupa memoriji u formatu `reg:offset`, dok je procesor u real modu operacije, memorijska lokacija kojoj će se pristupiti računa se na slijedeći način:

$$(a \ll 4) + b$$

Gdje:

- `a` → Predstavlja 16-bitnu vrijednost dobivenu čitanjem vrijednosti segmentnog registra `reg` iz instrukcije.
- `b` → Predstavlja 16-bitnu vrijednost dobivenu računanjem asemblera izraza `offset` iz instrukcije.

Kao primjer, pretpostavimo da je dat slijedeći kod:

```
movw $0xaf2c, %ax  
movw %ax, %ds ①  
movb %bh, %ds:0x0b ②
```

- ① Registrovati `ds` punim konstantom `0xaf2c`.

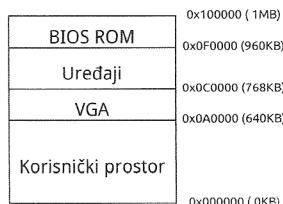
Pristupamo memoriji na adresi  $(0xaf2c \ll 4) + 0xb \rightarrow 0xaf2cb$ , gdje snimamo bajt pročitan iz registra `bh`. Identičan

- ② rezultat bi se dobio za slučaj da je instrukcija glasila `movb %bh, 0x0b`, obzirom da bi asembler implicitno opet koristio registrovati `ds`.

Obzirom na način kako se vrši adresiranje, real mod ima slijedeća ograničenja:

- Nije moguće adresirati više od 1MB memorije.
- Memoriji se pristupa u blokovima od 64KB, tzv segmenti, gdje početak bloka mora biti učitan u odgovarajući segmentni registar.
- Ne postoji nikakav oblik protekcije za pristupanje određenim blokovima u memoriji.

Dodatni problem predstavlja činjenica da adresni prostor od 1MB nije kompletan na raspolaganju za aplikacije. Slika 3-1 prikazuje da fizički prostor, pored prostora od 640KB koje mogu koristiti aplikacije, uključuje i dijelove koji su rezervirani za VGA prikaz, prostor za uređaje i BIOS program.



Slika 3-1. Fizički adresni prostor u real modu

Operativni sistemi u procesu pokretanja nastoje što prije prebaciti procesor iz real moda u protected mod operacije, zbog činjenice da procesor u ovom modu nema niti jedno od navedenih ograničenja.

### Adresiranje u protected modu

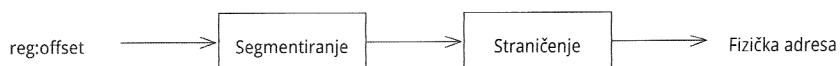
Adresiranje u protected modu procesora pruža niz prednosti, ali je istovremeno i značajno kompleksniji proces u odnosu na real mod operacije. Kao i u real modu, adrese u asembler kodu daju se u obliku izraza koji predstavlja *logičku adresu*:

reg:offset

Gdje su:

- reg → 16-bitni segmentni registar,
- offset → bilo koji asembler izraz za pristup memoriji.

Da bi se od logičke adrese dobila fizička adresa u memoriji procesor vrši dvije translacije koje su prikazane na slici 3-2.



Slika 3-2. Translacija adresa u protected modu

Prva translacija zove se *segmentiranje* (segmentation), njome se od logičke adrese dobija *linearna* adresa. Tokom slijedeće translacije, označene kao *straničenje* (paging), linearna adresa prevodi se u fizičku adresu. Hardver jedinica za segmentiranje aktivna je od trenutka kada procesor pređe u protected mod operacije i nije je moguće isključiti. Sa druge strane, jedinica koja implementira straničenje mora se eksplicitno uključiti. Kada je straničenje isključeno, linearna adresa dobivena nakon prve translacije adresa ujedno je i fizička. Hardver koji obavlja straničenje i segmentiranje jednim imenom se označava MMU (Memory Management Unit).

## SEGMENTIRANJE

Procesom segmentiranja linearni adresni prostor organizira se u *segmente*. Segment predstavlja kontinuirani memorijski prostor unutar kojeg procesor aplicira iste dozvole za pristup. Definira se putem 64-bitne strukture označene kao *eskriptor*, unutar koje se nalaze informacije o tipu segmenta, njegovom početku i kraju u linearном adresnom prostoru, kao i dozvole koje procesor provjera-va kada se pristupa segmentu.

63	56 55	54 53	52 51	48 47 46	45 44 43	40 39	16 15	0		
Base 24-31	G	D/B	0	AVL	Limit 16-19	P	DPL	S Type	Base 0-23	Limit 0-15

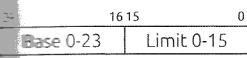
Slika 3-3. Segment deskriptor

Sadržaj segment deskriptora prikazan je na slici 3-3. Segment asociran sa deskriptorom je validan, tj prisutan u memoriji, ako deskriptor polje P ima vrijednost 1. Adresa početka segmenta u linearnom adresnom prostoru je 32-bitna vrijednost koja se dobija ujedinjavanjem dva polja u deskriptoru: Base 0-23 i Base 24-31, dok se krajnja adresa segmenta dobija putem polja Limit 0-15 i Limit 16-19. Kada deskriptor polje G ima vrijednost 1, procesor šifta polje Limit u lijevo za 12 mesta, a nova mjesta u polju popunjava sa binarnim ciframa 1. Ovisno od polja G, segment efektivno može imati slijedeće veličine:

- $G=0 \rightarrow$  Veličina segmenta je od 0B do 1MB, i može se regulisati u koracima od 1B.
- $G=1 \rightarrow$  Veličina segmenta je od 4KB do 4GB, i može se regulisati u koracima od 4KB.

Informacije o segmentima čuvaju se u tabelama koje su označavaju kao: GDT (Global Descriptor Table) i LDT (Local Descriptor Table). Svaki red ovih tabela zahtijeva 8 bajta prostora i predstavlja jedan deskriptor. Operativni sistemi, uključujući XV6, obično koriste samo GDT, dok LDT zanemaruju. Broj redova u GDTu određuje ukupan broj definiranih globalnih segmenata u sistemi. Prvi red GDTa, označen kao NULL segment, se ne koristi.

Procesor u posebnom 48-bitnom registru, označenom kao gdtr, čuva informacije veličini GDTa, kao i o lokaciji u memoriji gdje počinje ova tabela. Prije prelaska u protected mod sadržaj gdtr registra mora se učitati sa neke validne memorijske lokacije uz pomoć instrukcije lgdt. Postojeći sadržaj registra se može snimiti u memoriju putem instrukcije sgdt. Obzirom da se prvih 15 bita registra gdtr



slici 3-3. Segment asociran je s memorijom, ako desektor u segmentu u linearnoj memoriji post koja se dobija uje-  
23 i Base 24-31, dok  
a Limit 0-15 i Limit 1, procesor šifta polje  
polju popunjava sa bi-  
efektivno može imati

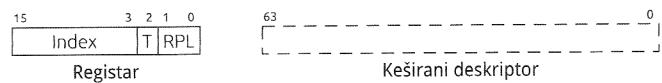
B, i može se regulisati  
GB, i može se reguli-

zama koje su označeni GDT (Local Descriptor Table) prostora i predstavlja  
jući XV6, obično ko-  
dovava u GDTu odre-  
enata u sistemi. Prvi  
priosti.

označenom kao gdtr,  
ji u memoriji gdje po-  
d sadržaj gdtr registra  
okacije uz pomoć ins-  
te snimiti u memoriju  
15 bita registra gdtr

dovi operacije procesora

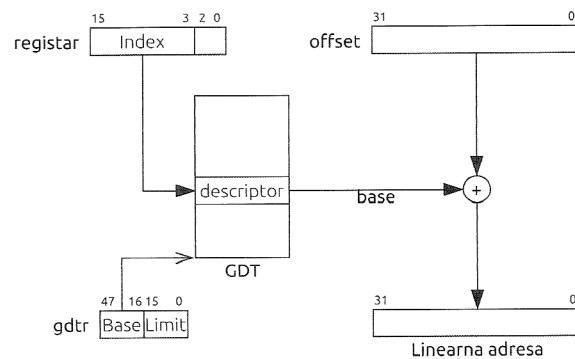
koristi za snimanje informacije o ukupnoj veličini GDTa u bajtima, maksimalan broj deskriptora u tabeli je 8192.



Slika 3-4. Segment selektor

Deskriptor tabele se indeksiraju putem segmentnih registara koji se u protected modu označavaju kao *selektori*. Sadržaj segmentnog registra prikazan je na slici 3-4. Pri svakoj promjeni vrijednosti bilo kojeg segmentnog registra mijenja se segment sa kojim je taj register asociran. Ova operacija obavlja se na način da se iz GDTa preuzima kompletan deskriptor, i to iz reda koji ima redni broj u skladu sa novom vrijednošću polja Index datog registra. Ukoliko je polje T u selektoru postavljeno na 0, tada se indeksira GDT, u suprotnom indeksira se LDT. Deskriptor preuzet iz tabele snima za buduće korištenje u nevidljivom dijelu selektora, na slici označenom kao keširani deskriptor.

Nakon što se asocira da nekim segmentom, register se može koristiti prilikom segmentiranja kao što je prikazano na slici 3-5.



Slika 3-5. Segmentno adresiranje

Segmentiranje započinje od logičke adrese kojom je dat segment selektor *register*, i memorijska lokacija *offset* unutar segmenta. Iz keširanog deskriptora u selektoru čita se adresa početka segmenta, i to iz polja *base*, a zatim se sumira sa poljem *offset* iz logičke adrese. Dobivena vrijednost predstavlja linarnu adresu koja služi kao ulaz u drugi nivo translacije, ili, ukoliko je isključeno straničenje, predstavlja konačnu fizičku adresu u memoriji.

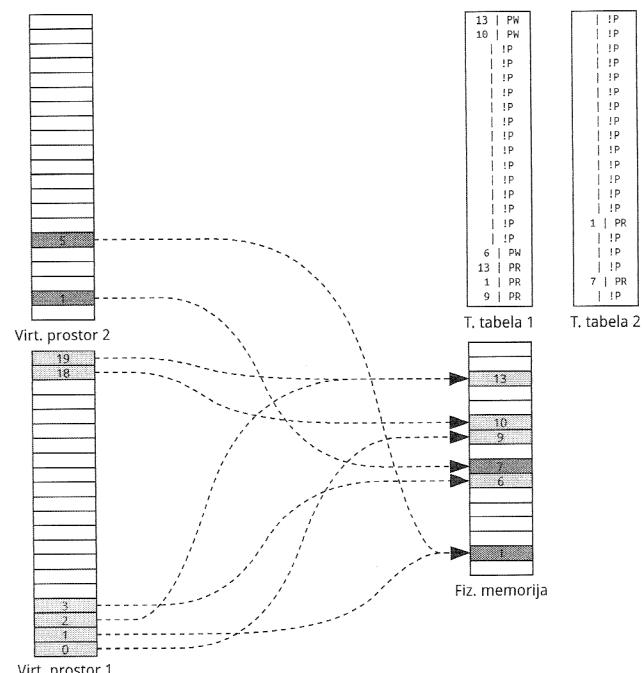
## STRANIČENJE

Straničenje je proces kojim se formira proizvoljan broj virtualnih linearnih adresnih prostora, i vrši njihovo mapiranje (translacija) u jedan fizički adresni prostor. Nakon što je aktivirano, svaka linearna adresa, koju procesor generira u bilo kojem trenutku, pripada određenom virtualnom prostoru. Osnovna jedinica za mapiranje je *stranica*, koja predstavlja kontinuirani adresni blok fiksne veličine. Za Intel platformu stranice mogu biti velike 4KB ili 4MB.

Svaki definirani virtualni adresni prostor dijeli se na stranice. Raspoloživa fizička memorija također se dijeli na stranice, koje se označavaju kao *okviri*. Stranica iz bilo kojeg virtualnog adresnog prostora, ukoliko se koristi, mora sa sobom imati asociran neki okvir u fizičkoj memoriji. Proces uspostavljanja relacije između stranice u viruelnoj memoriji i okvira u kojeg je stranica učitana u fizičkoj memoriji naziva se mapiranje. Nakon što se uspostavi mapiranje, svaki pristup nekom bajtu unutar bilo koje stranice virtualnog prostora predstavlja pristup tom bajtu u nekom asociranom okviru fizičkog adresnog prostora. Informacije o mapiranjima stranica iz pojedinačnih virtualnih prostora čuvaju se u translacionim tabelama. Ukoliko se vrši straničenje u jednom nivou neophodna je jedna translaciona tabela kako bi se definirao jedan virtualni adresni prostor. Kada je uključeno straničenje, procesoru mora biti data neka translaciona tabela koju će koristi prilikom straničenja u nadrednom vremenskom intervalu. Ovim se zapravo određuje aktuelni virtualni adresni prostor iz kojeg procesor generira linearne adrese. Procesor podržava instrukcije za promjenu zadane translacione ta-

bele, čime je omogućena promjena aktuelnog virtualnog adresnog prostora.

Slika 3-6 predstavlja konceptualni primjer mapiranja dva virtualna adresna prostora sastavljenih od maksimalno 20 stranica, u jedan fizički prostor koji na raspaganju ima 16 okvira. Virtualni adresni prostori definirani su translacionim tabelama koje su također prikazane na slici. Svaki red pojedinačne tabele opisuje mapiranje jedne stranice iz odgovarajućeg virtualnog adresnog prostora. Broj reda u tabeli korespondira broju stranice u virtualnom adresnom prostoru. Red u tabeli sadrži broj okvira fizičke memorije u koji se dala stanica mapira, kao i dodatne bite koji opisuju mapiranje. Ako je bit P uključen, stranica ima mapiranje, u suprotnom, dio virtualnog adresnog prostora asociranog sa datom stranicom se ne koristi. Biti R i W određuju da li dozvoljeno samo čitanje, ili i pisanje unutar date virtualne stranice.



Slika 3-6. Primjer straničenja u jednom nivou

Prepostavimo da su stranice u primjeru velike 4KB. Neka je u nekom trenutku aktuelan virtualni prostor 1, tj procesor za stranicenje koristi translacionu tabelu 1. Neka program koji procesor izvršava u datom trenutku generira linearnu adresu 4098. Ovo je adresa drugog bajta u stranici 1 iz virtualnog prostora 1. Procesor u skladu sa aktuelnom translacionom tabelom generira fizičku adresu 4098, jer je to adresa drugog bajta fizičkog okvira 1 sa kojim je asocirana stranica 1. Igram slučaja, adresa iz virtualnog adresnog prostora poklopila se sa fizičkom adresom. Ukoliko bi, uz iste pretpostavke, program generirao istu linearnu adresu 4089, dok je aktuelan virtualni prostor 2, fizička adresa koju bi procesor potraživao u ovom slučaju bila bi 28674, obzirom da je to adresa drugog bajta u fizičkom okviru 7 sa kojim je asocirana stranica 1 iz virtualnog prostora 2. Dakle, iste adrese unutar dva različita virtualna prostora generiraju različite fizičke adrese.

Na osnovu prethodnog primjera možemo izvesti određene opšte zaključke:

- Stranicenje omogućava potpunu separaciju virtualnih adresnih prostora.
- Određeni virtualni prostor koristi količinu fizičke memorije u skladu sa brojem stranica iz tog virtualnog prostora koje imaju mapiranje.
- Svaki virtualni prostor je kontinuiran, međutim, fizički okviri asocirani sa stranicama iz jednog virtualnog prostora su u diskontinuitetu.
- Više stranica iz istog virtualnog prostora mogu se mapirati u jedan okvir, npr. stranice 2 i 19 iz virtualnog prostora 1 mapiraju se u fizički okvir 13.
- Stranice iz različitih adresnih prostora mogu se mapirati u isti okvir, čime se dobija kontrolirano dijeljenje fizičke memorije između različitih virtualnih prostora, npr. stranica 1 iz virtualnog prostora 1 i stranica 5 iz virtualnog prostora 2 mapiraju se u okvir 1.

velike 4KB. Neka je u 1, tj procesor za str- program koji procesor u adresu 4098. Ovo je prostora 1. Procesor u generira fizičku adresu kvira 1 sa kojim je aso- tualnog adresnog pros- iko bi, uz iste prepos- su 4089, dok je aktu- eli procesor potraživao to adresa drugog bajta manica 1 iz virtualnog kota virtualna prostora

izvesti određene opšte

aciju virtualnih adres-

nu fizičke memorije u og prostora koje imaju

neđutim, fizički okviri lnog prostora su u di-

ra mogu se mapirati u lnog prostora 1 mapi-

nogu se mapirati u isti jenje fizičke memorije pr. stranica 1 iz virtu- prostora 2 mapiraju

zovi operacije procesora

Prethodni primjer zbog jednostavnosti je uključivao nerealistič- ~~ne~~ virtualne adresne prostore sa svega 20 stranica. Ukoliko uzmemo ~~realan~~ virtualni adresni prostor od 4GB, i ako prepostavimo da svaki red u translacionoj tabeli koristi 4B za opis mapiranja pojedinač- ~~ne~~ stranice, dobijamo da je za pohranjivanje jedne translacione ta- bele potrebno 4MB prostora. Ova količina memorije neophodna je bez obzira na količinu memorije koju virtualni adresni prostor potražuje za svoje funkcionisanje. Ukoliko prepostavimo da će svaki program u izvršenju dobiti svoj virtualni adresni prostor, možemo zaključiti da će za deset istovremeno aktivnih programa u nekom trenutku biti potrebno 40MB fizičke memorije samo za pohranjiva- ~~ne~~ translacionih tabela, što je očigledno nepraktično. Zbog toga se uvodi straničenje u više nivoa.

Kod straničenja u dva nivoa, kao i u prethodnom slučaju, fizička memorija je podijeljena na okvire, dok se virtualni adresni prostor prvo dijeli na direktorije, koji se dalje dijele na stranice.

Kod IA32 platforme virtualni adresni prostor sastoji se od 1024 direktorija. Svaki direktorij sadrži tačno 1024 stranice od po 4KB, ~~ili~~ predstavlja samo jednu stranicu od 4MB. Kao i kod straničenja u jednom nivou, stranice iz pojedinačnih direktorija mapiraju se u okvire fizičke memorije. Razlika je u tome što se za mapiranje, umjesto jedne, sada koristi više tabela. Konkretno, po jedna tabela za opis mapiranja svih stranica koje pripadaju jednom direktoriju. Ove tabele predstavljaju tabele drugog nivoa straničenja i nose oznaku PT (Page Table). Jeden red u PTu pored bita za dozvole, sadrži redni broj fizičkog okvira u koji se mapira stranica iz datog direktorija. Pojedinačni red u PTu označava se kao PTE (Page Table Entry). Ukoliko se u nekom virtualnom adresnom prostoru ne koristi niti jedna stranica u određenom direktoriju, PT za taj direktorij se izostavlja.

Dok se prilikom opisa virtualnog adresnog prostora određeni PTi izostavljaju, tabela, koja nosi oznaku PD (Page Directory) i služi za prvi nivo straničenja, je neizostavna. Unutar ove tabele u pojedi-

načnim redovima, pored određenih bita za dozvole pristupa, nalaze se memorijske lokacije na kojima su tabele za drugi nivo straničenja. Indeks reda u tabeli odgovara rednom broju direktorija čija je adresa PTa snimljena u tom redu. Red u PDu označava se kao PDE (Page Directory Entry).

Ukoliko prepostavimo da se jedan PDE može smjestiti u 4B, obzirom da ukupno imamo 1024 direktorija zaključujemo da kompletan PD zahtijeva 4KB memorije. Isto tako, pod uslovom da se jedan PTE može kodirati u 4B, i obzirom da svaki direktorij ima 1024 stranice, zaključujemo da svaki PT također zahtijeva 4KB memorije. Dakle, jedan okvir fizičke memorije od 4KB, dovoljan je za smještanje jedne kompletne tabele za straničenje prvog ili drugog nivoa.

Uštede u količini korištene fizičke memorije ostvarene upotrebom straničenja u dva nivoa možemo ilustrirati putem slijedećeg primjera. Prepostavimo da se u nekom virtualnom adresnom prostoru izvršava program koji zahtijeva:

- 5KB — Za sve sekcije pročitane iz izvršnog fajla, i to: `text`, `data`, `rodata` i `bss`,
- 4KB — Za stek.

Sa stanovišta konzumacije fizičkih okvira dati program potražuje:

- Jedan okvir, za mapiranje jedne stranice steka koja se nalazi na vrhu virtualnog adresnog prostora,
- dva okvira, za mapiranje dvije suksesivne stranice koje se koriste za sekcije: `text`, `rodata`, `data` i `bss`, a koje se nalaze na početku virtualnog adresnog prostora.
- jedan okvir, za PD kojim se opisuju mapiranja pojedinačnih direktorija virtualnog adresnog prostora.
- dva okvira, za dva PTa u kojima su opisana pojedinačna mapiranja stranica iz direktorija 0 i 1023, obzirom da se ta dva di-

rel  
to  
Prog  
adresni  
alnog a  
je. Ovo  
se koris  
red nece  
ce, zaht  
prostora

Nast  
su prik  
koji se i

1. B
2. B
3. B

Slika 3-1

Slika 3-2  
Za H  
20 bita  
kompli  
ima vri  
predsta

zvole pristupa, nalaze se na nivo straničenja. Direktorija čija je adresa daje se kao PDE (Page

če smjestiti u 4B, obično učujemo da kompletan uslovom da se jedan direktorij ima 1024 stranice, a svaka je 4KB memorije. Dovoljan je za smještaj ili drugog nivoa.

ne ostvarene upotrebiti putem slijedećeg binarnog adresnom prostora.

og fajla, i to: text, da-

dati program potražu-

steka koja se nalazi na

ne stranice koje se koristi, a koje se nalaze na po-

mapiranja pojedinačnih stranica.

ana pojedinačna mapiranja, a zato da se ta dva di-

zovi operacije procesora

rektorijskih stranica nalaze na početku i kraju virtualnog adresnog prostora.

Program zahtijeva tri okvira za efektivno korišteni virtualni adresni prostor, te dodatna tri okvira za uspostavljanje svog virtualnog adresnog prostora, što ukupno iznosi 24KB fizičke memorije. Ovo predstavlja značajnu redukciju u odnosu na slučaju kada bi se koristilo straničenje u jednom nivou, gdje bi isti program, posredstvom neophodnih 12KB memorije za mapiranje tri korištene stranice, zahtijevao i dodatnih 4MB za uspostavljanje virtualnog adresnog prostora.

Nastavimo sa analizom sadržaja pojedinačnih PDEa i PTEa koji su prikazani na slikama: 3-7 i 3-8. Obje strukture imaju sličan sadržaj koji se interpretira na slijedeći način:

1. Biti 0-9 — nose informacije o mapiranju, npr.: dozvole pristupa, način keširanja, itd, gdje se svaki bit interpretira ponaosob.
2. Biti 9-11 — operativni sistemi mogu koristiti na bilo koji način.
3. Biti 12-31 — čine broj okvira.



Slika 3-7. PDE



Slika 3-8. PTE

Za PDE, pod uslovom da njegov bit S ima vrijednost 0, gornjih 20 bita predstavlja broj okvira u fizičkoj memoriji gdje se nalazi kompletan PT od asociranog direktorija. U suprotnom, kada bit S ima vrijednost 1, cijeli direktorij u virtualnom adresnom prostoru predstavlja samo jednu stranicu veličine 4MB. Gornjih 10 bita PDE-a

u tom slučaju predstavlja broj okvira veličine 4MB u koji se mapira data stranica. Direktorij se koristi, tj ima svoj PT, samo pod uslovom da je u PDEu datog direktorija bit P postavljen na 1. U suprotnom, niti jedna stranica koja u virtualnom adresnom prostoru pripada tom direktoriju se ne koristi, a PT od tog direktorija se izostavlja.

Za PTE, pod uslovom da njegov bit P ima vrijednost 1, gornjih 20 bita predstavlja broj okvira u fizičkoj memoriji u koji se mapira stranica iz virtualne memorije asocirana sa datim PTEom. U suprotnom, ako je bit P postavljen na 0, stranica asocirana sa PTEom se ne koristi, tj. nema mapiranje.

Za aktivaciju i konfiguraciju straničenja Intel platforma koristi posebne kontrolne registre, i to na način:

- Registar cr0 — Ukoliko bit 31, tzv. *paging flag*, ima vrijednost 1, straničenje je uključeno.
- Registar cr4 — Ukoliko bit 4, tzv. *page size extensions*, ima vrijednost 1, aktivirana je podrška za stranice veličine 4MB.
- Registar cr3 — Treba biti postavljen na lokaciju u memoriji na kojoj počinje PD virtuelnog prostora kojeg želimo napraviti aktuelnim.

Svi navedeni kontrolni registri mogu se modificirati sa standardnim `mov` instrukcijama. Očigledno da prije aktiviranja straničenja trebamo odabrati određeni virtualni adresni prostor učitavanjem adrese adekvatnog PDA u registar cr3.

Po uključenju straničenja, procesor počinje vršiti drugu translaciju memorijskih adresa koja je prikazana na slici 3-9. Straničenje počinje od linearne adrese koja predstavlja lokaciju u aktuelnom virtualnom adresnom prostoru odabranom putem registra cr3. Ova lokacija pripada određenoj stranici unutar određenog direktoriju.

Slika 3-9. 1

Da bi  
adresu un  
dijelimo na

indeks2 —  
Predstava

indeks1 —  
Predstava

offset — bi  
Predstava

Nakon po  
ku adresu

1. Proces  
nutne  
rekto

2. Iz direk  
memorijski

Adresiranje u

4MB u koji se mapira  
u PT, samo pod uslo-  
avljen na 1. U suprot-  
adresnom prostoru pri-  
og direktorija se izos-

vrijednost 1, gornjih 20  
u koji se mapira stra-  
m PTEom. U suprot-  
cirana sa PTEom se ne

Intel platforma koristi

flag, ima vrijednost

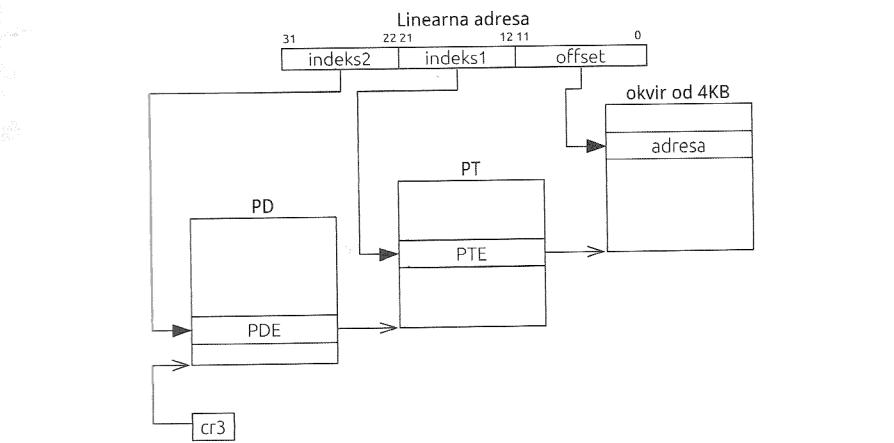
size extensions, ima vri-  
te veličine 4MB.

lokaciju u memoriji na  
kojeg želimo napraviti

odificirati sa standard-  
aktiviranja straničenja  
i prostor učitavanjem

vršiti drugu transla-  
slici 3-9. Straničenje  
lokaciju u aktuelnom  
tem registra cr3. Ova  
edenog direktoriju.

čovi operacije procesora



Slika 3-9. Translacija adresa straničenjem u dva nivoa

Da bi odredili brojeve direktorija i stranice, kao i konkretnu adresu unutar stranice kojoj pristupamo, 32-bitnu linearnu adresu dijelimo na slijedeća tri polja:

#### indeks2 — biti 21-31

Predstavlja broj direktorija.

#### indeks1 — biti 12-21

Predstavlja broj stranice u odabranom direktoriju.

#### offset — biti 0-11

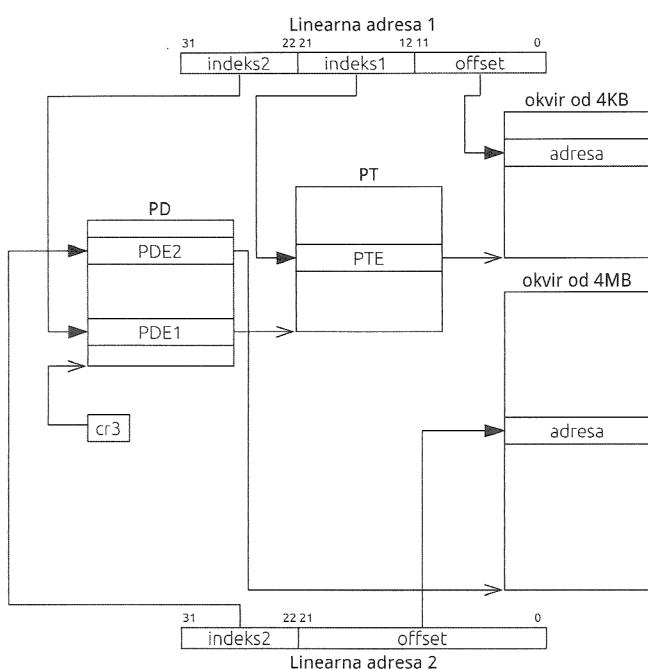
Predstavlja adresu bajta unutar odabrane stranice.

Nakon podjele na polja, proces konverzije linearne adrese u fizičku adresu odvija se u sljedećim koracima:

1. Procesor koristi polje indeks2 za indeksiranje PDA na koji trenutno pokazuje registar cr3, čime procesor dolazi do PDEa direktorija kojem se pristupa.
2. Iz dobivenog PDEa, procesor preuzima broj okvira u fizičkoj memoriji u kojem se nalazi PT od datog direktorija.

3. Procesor indeksira dobiveni PT sa poljem indeks1 iz linearne adrese, čime dolazi do PTEa od odabrane stranice u direktorijsku.
4. Iz PTEa procesor preuzima gornjih 20 bita, tj. broj okvira u koji se mapira odabrana stranica.
5. Kombiniranjem dobivenog broja okvira sa offset poljem iz linearne adrese, dobija se konkretna fizička adresa kojoj se pristupa.

Navedeni koraci u straničenju obavljaju se pod pretpostavkom da nije uključena podrška za stranice od 4MB. Ukoliko se uključi ova opcija, slika 3-10 prikazuje proces straničenja.



Slika 3-10. Straničenje sa mješanom veličinom stranica

Za ovaj slučaj, kada na osnovu vrijednosti polja indeks2 dobije PDE od direktorija, procesor prvo vrši provjeru stanja njegovog bita S. Za vrijednost 0 translacija se nastavlja kao u prethodnom slučaju, što je na slici prikazano sa PDE1. Međutim, ukoliko polje S ima

vrijednost 1, dolazi do PTEa od odabrane stranice u direktoriju. Procesor preostale bitove u direktoriju koju je odabranu stranicu, p

Dok pristup stranici aktivira bit A direkto u direktoriju koju je odabranu stranicu, p

Straničenje je potrebno za fizičke memorijske stranice, jer pristup stranici zahtijeva tri adresiranja: jedno PDEa, drugo PTEa i treće adresiranje u okviru stranice, p

Za reduciranje vremenskih razmaka između sebno brzih prelaza, Lookaside Buffer je implementiran u procesoru. Članovi LBU su organizirani u skupove po broju stranica. Svaki skup sadrži PTE od dva do četiri elementa, p

Pri svakom mijenjanju stranice obavljaju se sledeći koraci:

1. Iz logičke adrese dobivaju se dvadeset jedan bitova, pomoću kojih se pristupa LBU.
2. Ukoliko je u LBU postojao referentni bit, ukoliko je vrijednost 1, mijenjanje stranice neće biti obavljeno.

vrijednost 1, kompletan direktorij mapira se u jedan okvir veličine 4MB. Procesor u ovom slučaju uzima samo 10 gornjih bita iz PDEa, a preostale bite za fizičku adresu preuzima iz polja offset linearne adrese koje sada ima 20 bita. Ovo je na slici prikazano sa PDE2.

Dok pristupa nekoj stranici u procesu translacije adresa, procesor aktivira bit A od PTEa pristupane stranice, kao i bit A od PDEa direktorija kojem stranica pripada. Ako pristup uključuje pisanje u datu stranicu, procesor aktivira i D bit PTEa pristupane stranice.

Straničenje u dva nivoa donosi fleksibilnost i uštede u upotrebi fizičke memorije. Međutim, drastično smanjuje performanse sistema, jer pristup virtualnoj memoriji radi čitanja ili pisanja podataka zahtijeva tri pristupa fizičkoj memoriji, i to: prvi — radi preuzimanja PDEa, drugi — radi preuzimanja PTEa, i treći radi pristupa konkretnoj lokaciji fizičke memorije u koju se čita ili piše. Trostruki proces indirekcije događa se čak i prilikom preuzimanja svake instrukcije koju procesor izvršava u okviru nekog programa.

Za reduciranje problema indirekcije u procesor se ugrađuje posebno brza asocijativna memorija označena kao TLB (Translation Lookaside Buffer), koja se organizira kao podatkovna struktura mape. Članovi mape su parovi, gdje prvi element u paru predstavlja broj stranice u virtualnom adresnom prostoru, a drugi element je PTE od date stranice. Ova mapa se može brzo pretraživati po prvim elementima snimljenih parova.

Pri svakom pristupu virtualnoj memoriji uzimajući u obzir TLB obavljaju se slijedeći koraci:

1. Iz logičke adrese radi pretraživanja TLBa uzima se gornjih dvadeset bita koji predstavljaju broj stranice kojoj se trenutno pristupa.
2. Ukoliko TLB sadrži element spram date logičke adrese, informacije iz pronađenog para u TLBu se koriste za straničenje, tj.

reducira se dvostruki pristup memoriji potreban za pronalaženje PTEa.

3. Ukoliko TLB ne sadrži potrebne informacije za translaciju, obavljaju se standardne operacije preuzimanja PTEa iz memorije, koji se zatim pohranjuje u TLBu.
  - Ukoliko je u datom trenutku TLB pun, da bi se napravio prostor neophodan za novi par, procesor nasumično briše neki postojeći par unutar TLBa.

Obzirom da se programi povezuju principu lokalnosti, tj. često koriste prethodno korištene funkcije i podatke, TLB memorija od nekoliko desetina parova dovoljna je da skoro u potpunosti eliminiра problem indirekcije prilikom straničenja.

Funkcionisanje TLBa je transparentno za programere. Prilikom promjene aktuelnog virtualnog adresnog prostora, promjenom registra  $cr3$ , svi parovi unutar TLB postaju nevažeći, tj. TLB se u potpunosti prazni (TLB flush). Određeni PTEovi koji se često koriste mogu biti označeni kao globalni, aktiviranjem bita  $G$  u PTEu. Kada je označen kao globalan, par asociran sa PTEom neće biti uklonjen iz TLBa prilikom promjene registra  $cr3$ . Bit 7 registra  $cr4$ , aktivira podršku za globalne PTEove.

Ukoliko se u aktuelnom virtualnom prostoru promjeni mapiranje već mapiranog PTEa, PTE je potrebno proglašiti nevažećim u TLBu putem instrukcije `invlpg`. Instrukcija uzima adresu koja pripada stranici čiji PTE treba proglašiti nevažećim unutar TLBa, a može proglašiti nevažećim i PTE od globalne stranice.

## Protekcija

Protected mod pored načina pristupa memoriji dodatno se razlikuje od real moda uvođenjem koncept privilegija. Nakon transfera u protected mod, procesor dok izvršava instrukcije ima asociran trenutni nivo privilegija. Ovaj nivo se mjeri brojevima od 0 do 3. Naj-

viši nivo privilegija je 0, a najniži 3. Trenutni nivo privilegija koji ima procesor označava se sa CPL (Current Privilege Level), a definiran je poljem DPL (Descriptor Privilege Level) u segment deskriptoru koji je trenutno asociran sa registrom CS, tj. segment deskriptoru koji je keširan u CS selektoru.

Privilegijama se ostvaruje hardverska protekcija resursa od neovlaštenog pristupa. Resursi koji se štite na Intel platformi su:

- segmenti,
- stranice i direktoriji u virtualnoj memoriji,
- registri,
- instrukcije,
- uređaji.

Svaki od navedenih resursa sa sobom vezuje određeni nivo privilegija koji procesor mora da posjeduje da bi u nekom trenutku mogao koristiti resurs. Ukoliko procesor dok ima niži nivo privilegija od potrebnog pokuša koristiti štićeni resurs, hardver proizvodi određenu iznimku, a načeće GPF (General Protection Fault). Iznimkom se obustavlja izvršenje programa čija instrukcija je proizvela iznimku, a procesor počinje izvršavati kod sa posebne lokacije u memoriji, koji ima zadatak da adekvatno riješi nastalo iznimno stanje. Iznimke predstavljaju varijantu prekida koje ćemo tretirati u posebnom poglavljju.

Sistemski softver, tj. operativni sistem, za većinu resursa može konfigurirati potrebni nivo privilegija za njihovo korištenje. U praksi, većina operativnih sistema pri konfiguraciji pristupa pojedinačnim resursima koristi jedan od dva nivoa privilegija: najviši i najniži.

Nivoi privilegija potrebni za korištenje instrukcija i registara definirani su Intel platformom, i to pojedinačno za svaku instrukciju i registar. Ove nivoe privilegija nije moguće konfigurirati. Veći-

na instrukcija IA32 platforme su neprivilegovane i mogu se koristiti dok je procesor u stanju najnižeg nivoa privilegija. Slijedeće instrukcije su privilegovane, te se mogu koristiti samo dok je procesor u najvišem, tj. nultom, nivou privilegija:

- `lgdt`, `lidt`, `invlpg`, `hlt`.

Za određene neprivilegovane instrukcije vrše se provjere privilegija spram korištenih operanda.

Kontrolnim registrima `cr0` – `cr4` moguće je pristupati samo sa nultim nivoom privilegija. Bilo kakav pokušaj `mov` operacije sa nižim nivoom privilegija kad je destinacija jedan od kontrolnih registara izaziva GPF.

Prilikom promjene vrijednosti segmentnih registara vrše se elaboratne provjere nivoa privilegija. U ovim provjerama učestvuju:

#### **CPL (Current Privilege Level)**

Trenutni nivo privilegija procesora.

#### **RPL (Requested Privilege Level)**

Polje u segmentnom registru.

#### **DPL (Descriptor Privilege level)**

Polje u segment deskriptoru koji je odabran putem polja Index u segmentom registru.

#### **EPL (Effective Privilege Level)**

Maksimum od RPL i CPL.

Da bi u opštem slučaju operacija promjene vrijednosti segmentnog registra uspjela, potrebno je da EPL ima manju ili jednaku vrijednost u odnosu DPL. U suprotnom, nastaje GPF.

Iznimno

• za prvo

◦ C

◦ P

◦ b

◦ tu

• za prvo

◦ C

◦ P

◦ b

◦ d

• za prvo

◦ I

◦ T

◦ J

Dodatno  
GPF nastaje

• indeks

• odabir

• odabir

◦ P post

Nakon štiranje i ka pristup logi spram dozv branih tabe

Iznimno od ostalih segmentnih registara:

- za promjenu vrijednosti registra cs:
  - CPL i DPL moraju imati istu vrijednost.
  - polje TYPE u odabranom segmentnom deskriptoru mora biti postavljeno tako da je tip segmenta code, tj. da su date dozvole za izvršavanje koda iz segmenta.
- za promjenu vrijednosti registra ss:
  - CPL, RPL i DPL, moraju imati istu vrijednost.
  - polje TYPE u odabranom segmentnom deskriptoru mora biti postavljeno tako da je tip segmenta data i da su date dozvole za pisanje u segment.
- za promjenu vrijednosti registra ds, es, gs i fs:
  - nije moguće odabrati segmentne deskriptore čije je polje Type tipa system ili ukoliko je polje Type tipa code u kojem nije dozvoljeno čitanje.

Dodatno, prilikom promjene vrijednosti segmentnog registra GPF nastaje i u situacijama:

- indeksiranja van granica GDTa,
- odabira NULL segmentnog deskriptora,
- odabira segmentnog selektora koji nije prisutan, tj. čije je polje R postavljeno na 0.

Nakon što se učitavanjem ispravnih selektora uspostavi segmentiranje i kada se registrom cr3 odabere PD za straničenje, svaki pristup logičkom i linearном memorijskom prostoru provjerava se spram dozvola datih unutar aktivnih segmentnih deskriptora i odbrih tabelama za straničenje.

Prilikom adresiranja iznimke nastaju kada se pokuša:

- pisanje u segment koji nema postavljene dozvole za pisanje,
- čitanje iz segmenta koji nema postavljene dozvole za čitanje,
- pristupanje PDEu ili PTEu čije polje  $P$  ima vrijednost 0.
- pisanje u stranicu ili direktorij čiji PTE ili PDE imaju polje  $R$  postavljeno na 0.
- pristupanje, dok procesor ima niži nivo privilegija od nultog, PDEu ili PTEu čije polje  $U$  ima vrijednost 0.

## Komunikacija sa vanjskim uređajima

Da bi bili korisni, pored sposobnosti da izvode računanje i procesiranje podataka, programi trebaju imati mogućnost komunikacije sa vanjskim uređajima. Uređaji kao što su: miš, tastatura, skener itd., za programe obezbijedju ulazne podatke, dok monitor, printer, zvučnik itd, služe za prezentaciju rezultata proizvedenih u programima. Određeni uređaji, kao što su: disk, mrežna kartica, zvučna kartica itd., mogu imati dvostruku namjenu, tj. mogu se koristiti kao ulaz ili izlaz za programe.

Uzimajući u obzir različite funkcije i veliki broj uređaja različitih proizvođača, hardverska platforma treba da obezbjedi standardizirani interfejs za komunikaciju programa sa vanjskim uređajima. U ovom poglavlju osvrnut ćemo se na infrastrukturu koju IA32 platforma pruža za podršku komunikaciji sa vanjskim uređajima.

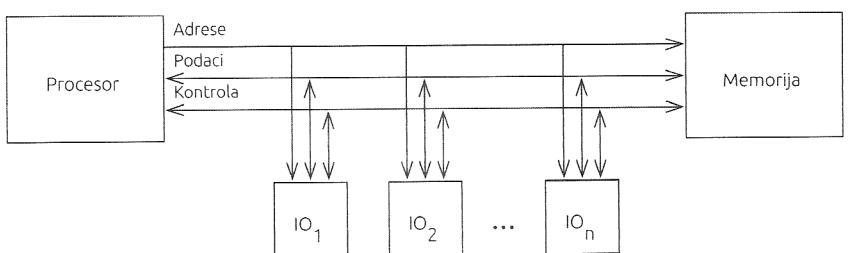
### Mapiranje vanjskih uređaja

Za komunikaciju procesora sa vanjskim uređajima obično se koriste dvije metodologije, i to:

1. memorijsko mapiranje,
2. port mapiranje.

Memorijsko mapiranje se češće implementira u praksi, mada određene kompanije, kao npr. Intel, na svojim arhitekturama istovremeno implementiraju obje metodologije.

Memorijsko mapiranje bazira se na šemi prikazanoj na slici 4-1.



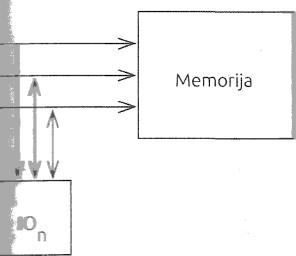
Slika 4-1. Memorijsko mapiranje uređaja

Slika prikazuje sistemsku sabirnicu, koja se sastoji od kontrolne, adresne i podatkovne sabirnice, na koju su povezani: procesor, memorija i proizvoljan broj uređaja označenih sa:  $IO_1, IO_2, \dots, IO_n$ . Še-ma je konceptualna, obzirom da se vanjski uređaji ne povezuju direktno na sistemsku sabirnicu, već na zasebne vanjske sabirnice, kao što su PCI, PCIe, ili ISA. Vanjske sabirnice se putem posebnih uređaja, tzv. mostova (bridge), povezuju na sistemsku sabirnicu.

Preko sabirnice procesor može pristupati vanjskim uređajima na isti način kao što pristupa memoriji. Ovaj koncept naziva se memorijsko mapiranje uređaja. Naime, jedan ili više kontinuiranih blokova fizičkog adresnog prostora rezervira se za uređaje. Kada procesor nakon obavljenog segmentiranja i straničenja emituje fizičku adresu koja pripada nekom od ovih blokova, zahtjev za čitanje ili pisanje podataka umjesto u memoriju proslijedi se u određeni registar koji pripada *kontroleru*, posebnom digitalnom sklopu koji kontroliše neki vanjski uređaj. Kontroler može da ima proizvoljan broj registara sa različitim brojem bita, a svaki registar dobija sukcesivnu adresu u fizičkom adresnom prostoru procesora. Spram uloge, registri se mogu podijeliti u sljedeće klase:

#### **statusni**

Čitanjem ovih registara procesor može utvrditi trenutno stanje uređaja.



### Kontrolni

**Pisanjem u ove registre procesor može zahtijevati od uređaja da obavi određenu operaciju.**

### Dodatakovni

**Ovi registri služe za razmjenu podataka između procesora i uređaja, potencijalno u oba smjera,**

**Određeni uređaji, a najčešće grafičke kartice, pored registara u fizički adresni prostor procesora mapiraju i bafere, sopstvenu internu memoriju.**

**Memorijskim mapiranjem bafera i registara kontrolera, procesor ponosno bilo koje instrukcije za pristup memoriji može ostvariti komunikaciju sa uređajem. Uz svaki uređaj proizvođači dostavljaju i specifikacije sekvenci pisanja ili čitanja registara njegovog kontrolera, putem kojih se uređaju može narediti da obavi određene operacije. Na osnovu zadatih specifikacija pišu se driveri, posebni moduli u operativnim sistemima, koji za obične aplikacije omogućavaju jednostavljenu, a često i unificiranu, vezu sa različitim vanjskim uređajima.**

**Slika 4.2 prikazuje raspored fizičkog adresnog prostora IA32 platforme. Za RAM memorijske module na raspolaganju su adresni responzi označeni kao RAM1 i RAM2.**

**Registri i baferi uređaja mapiraju se u tri regiona, i to:**

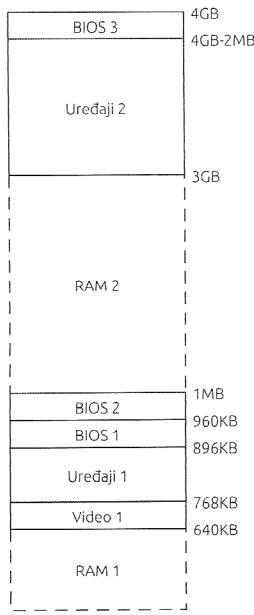
#### Vredniji 1

**Za zastarjeli pristup grafičkim karticama.**

#### Vredniji 2

**Za stare uređaje koji obično koriste ISA sabirnicu.**

**Za novije uređaje koji koriste PCI i PCIe sabirnice.**



Slika 4-2. Fizički memorijski adresni prostor za IA32 sisteme

Konkretan redoslijed uređaja u posljednja dva rezervirana segmenta mora biti u skladu sa Intel platformom, a definira se prilikom pokretanja sistema. Za ovu svrhu poseban program, označen kao BIOS, postavlja se u permanentnoj memoriji matične ploče, i to obično u obliku fleš memorije koja je mapirana u fizičku memoriju na tri lokacije BIOS 1, BIOS 2 i BIOS 3. Nakon što dobije napajanje, procesor počinje izvršavati kod iz bloka BIOS 2. Ovaj kod, između ostalog, komunicira sa svim kontrolerima uređaja koji su integrirani na matičnoj ploči, kao i onim koji se nalaze u utorima predviđenim za ekspanziju. Tokom ove komunikacije, koja je u skladu sa Intel specifikacijom, formira se memorijsko mapiranje registara i bafera od kontrolera otkrivenih uređaja.

Druga metodologija, port mapiranje, predstavlja koncept sličan memorijskom mapiranju. Ključna razlika je u tome da se za port mapiranje odvaja potpuno odvojeni adresni prostor za pristup vanjskim uređajima. Adrese u ovom prostoru predstavljaju registre kontrolera vanjskih uređaja i označavaju se kao *portovi*. Za pristup por-

tovima, obzirom da nije moguće koristiti standardne izraze za adresiranje memorije, platforma mora definirati posebne instrukcije.

Na Intel platformi instrukcije `in` i `out` koriste za pristup portovima. Putem instrukcije `in` moguće je čitanje određenog porta, dok `out` služi za pisanje u port.

Opšti format ovih instrukcija je:

```
in{b,w,l} port, reg  
out{b,w,l} reg, port
```

Pri čemu:

- `port` — može biti 8-bitni broj, ili 16-bitna vrijednost pročitana iz registra `dx`,
- `reg` — može biti vrijednost jednog od registara: `al`, `ax` ili `eax`, i to u ovisnosti od broja bita u podacima koji se iz nekog porta čitaju putem instrukcije `in` ili pišu u port putem instrukcije `out`.

Adresiranje iznad porta 255 moguće je postići samo putem registra `dx`. Redoslijed i lokacija portova ovise od uređaja koji su povezani na matičnoj ploči konkretnog sistema, a definirani su Intel platformom. Slijedeća lista u heksadecimalnom formatu prikazuje raspored nižih portova na tipičnom Intel računaru.

```
0000-001f : dma1  
0020-0021 : pic1  
0040-0043 : timer0  
0050-0053 : timer1  
0060-0060 : keyboard  
0064-0064 : keyboard  
0070-0071 : rtc0  
0080-008f : dma page reg  
00a0-00a1 : pic2  
00c0-00df : dma2  
00f0-00ff : fpu  
0170-0177 : pata_amd  
01f0-01f7 : pata_amd  
0376-0376 : pata_amd  
03c0-03df : vesafb
```

03f6-03f6 : pata\_amd  
03f8-03ff : serial

Kao primjer korištenja portova putem `in` i `out` instrukcija, analizirat ćemo asembler kod koji komunicira sa kontrolerom tastature. Kontroler tastature, kao što je prikazano na prethodnoj listi, ima komandne registre na portovima `0x64` i `0x60`. Vrijednost koju procesor upiše u ove portove predstavlja komandu za kontroler tastature. Port `0x64` ujedno služi i kao statusni registar. Stanjem drugog bita statusnog registra kontroler tastature u datom trenutku indicira spremnost da prima podatke na nekom od ulaznih portova. U slijedećem primjeru koriste se oba porta:

```
L:  
inb    $0x64,%al  
testb  $0x2,%al  ①  
jnz    L  
  
movb   $0xdf,%al  
outb   %al,$0x60 ②
```

Na samom kraju primjera, u liniji ②, procesor u ulazni port kontrolera upisuje vrijednost `0xdf`. Prije nego obavi upisivanje, procesor u liniji ① provjerava stanje uređaja. Kontroler nema mogućnost prijema komandi od procesora dok je uređaj zauzet obavljanjem neke operacije. Zbog toga, sve dok se ne promjeni status kontrolera, procesor ostaje u beskonačnoj petlji.

Obzirom da su vanjski uređaji značajno sporiji u odnosu na procesor, aktivna provjera statusa nekog uređaja u petlji, koja se označava kao *polling*, predstavlja značajan problem. Na modernim računarskim sistemima istovremeno se izvršava veliki broj programa koji imaju potrebu da koriste više uređaja. Procesor bi što više trebao biti zauzet izvršavanjem koda aktivnih programa, a ne beskorisnim čekanjem na promjenu statusa sporih uređaja. Zbog toga se uvodi sistem *preki-a*, putem kojeg uređaji mogu da obavijeste procesor o promjeni svog statusa, ili nakon što okončaju obavljanje neke operacije. Ovim se omogućava da procesor i uređaji mogu raditi

~~zaključeno~~, čime se nepotrebitno čekanje procesora na uređaje značajno udaljira.

~~Na~~ stanovišta protekcije, instrukcije `in` i `out` spadaju u red senzorskih instrukcija. Mogućnost upotrebe ovih instrukcija može se konfigurirati spram trenutnog nivoa privilegija procesora, i to se odnosi na IOPL polja EFLAGS registra kojeg čine biti 11 — 13. Kada procesor operira sa nultim nivoom privilegija, instrukcijom `popf` sa stečenim moguće postaviti sva polja EFLAGS registra, uključujući i IOPL polje. Na ovaj način se može postaviti minimalni nivo privilegija koji je neophodan za korištenje `in` i `out` instrukcija. Ukoliko procesor u trenutku kada je CPL veće od IOPL pokuša izvršiti `in` ili `out` instrukciju, generira se iznimka.

## Prekidi

Prekidi omogućavaju asinhroni rad procesora i kontrolera vanjskih uređaja u sistemu. Na Intel platformi postoje dva pristupa za implementaciju prekida koji se označavaju imenima:

### ~~8259~~ — Programmable Interrupt Controller

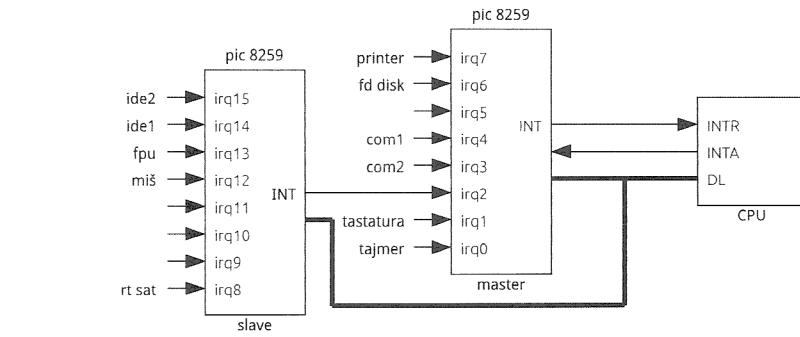
Šema za sisteme sa jednim procesorom koja uključuje dva uređaja tipa PIC-8295.

### ~~APIC~~ — Advanced Programmable Interrupt Controller

Šema za višeprocesorske sisteme koja uključuje LAPIC (Local — APIC), uređaj integriran u svaki pojedinačni procesor, te proizvoljan broj uređaja sa oznakom IOAPIC (Input Output — APIC).

## SLIČICA 4-3

~~Na~~ slici 4-3 prikazana je šema kojom se implementiraju prekidi na Intel sistemima sa jednom procesorskom jezgrom.

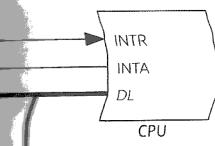


Slika 4-3. Jednoprocesorski sistem i kontroleri prekida

Kontroler svakog vanjskog uređaja koji proizvodi prekid mora se povezati na odgovarajuću ulaznu liniju jednog od dva PICa koji se označavaju kao **master** i **slave**. Ulazne linije PICa nazivaju se IRQ (Interrupt Request Line) i označavaju se brojevima od 0 do 15. Na liniji **IRQ2** nije moguće povezati bilo kakav uređaj, obzirom da se ta linija koristi za međusobno povezivanje dva PICa. Intel platforma unaprijed definira uređaje koji trebaju biti povezani na odgovarajuće IRQ linije. Na linije **IRQ5**, **IRQ9**, **IRQ10** i **IRQ11**, mogu se vezati PIR (Programmable Interrupt Router) kontroleri, koji imaju sposobnost da dinamički povezuju linije prekida uređaja koji se nalaze na mačnoj ploči sa IRQ linijama koje su date na raspolažanje PIR kontroleru. Proses povezivanja odvija se putem **in** i **out** instrukcija sa odgovarajućim portovima na kojim se nalaze registri PIR kontrolera.

Tri linije **master** PICa povezuju ovaj kontroler sa procesorom, pri čemu ulazne linije procesora **INTR** i **INTA** imaju po jednu, dok linija **DL** ima osam žica. Promjenom naponskog nivoa svog IRQ-a svaki uređaj može proizvesti signal koji za procesor treba da predstavlja prekid. PICovi imaju zadatku da putem **INT** i **DL** linija opišu ovaj događaj procesoru.

Kada dobije adekvatan signal na nekoj IRQ liniji, putem izlazne linije **INT**, **master** PIC signalizira procesoru prekid. Nakon toga, ka-



da preko INTA izlazne linije procesora stigne potvrda o primitku prekida, jedan od dva PICa, u skladu sa brojem IRQ linije koja je generirala prekid, na DL liniji postavlja broj koji označava *vektor* prekida.

Vektor prekida računa se kao suma broja IRQ linije koja je proizvela prekid i pomaka vektora koji svaki PIC prima prilikom inicijalizacije. Na primjer, pretpostavimo da je *master* PIC inicijaliziran sa pomakom vektora 32. Kada tastatura proizvede prekid, obzirom da je priključena na liniju irq1, procesor prima vektor prekida 33.

Nakon što procesor sa DL linije preuzme vektor prekida, sa staništa PIC sistema počinje procesiranje prekida. Novi prekidi istog ili nižeg prioriteta neće biti generirani sve dok se ne okonča procesiranje trenutnog prekida.

Sa druge strane, nakon izvršenja svake instrukcije, procesor provjerava stanje INTR linije. U slučaju prekida, tj. promjene stanja napona ove linije, procesor prestaje izvršavati trenutni program i putem INTA linije signalizira da je primio prekid. Potom, sa memorijске lokacije koja je definirana spram vektora prekida, procesor počinje izvršavati instrukcije kojim se tretira nastali prekid.

Sekvenca instrukcija sa memorijske lokacije koja se koristi u okviru tretmana prekida, označava se kao ISR (Interrupt Service Routine). Svaki vektor prekida mora imati asociirani ISR. Prije nego počne izvršavati odgovarajući ISR, procesor u stek memorijskom segmentu snima stanje registara prekinutog programa. Snimljeno stanje koristi se za nastavljanje izvršenja prekinutog programa, i to nakon što ISR tretira prekid. Neposredno pred kraj tretmana prekida, putem mapiranih portova PIC kontrolera, ISR treba poslati EOI (End of Interrupt) signal kojim se potvrđuje kraj tretmana datog prekida. Ovo se čini da bi PIC sistem mogao nastaviti slati nove prekide istog ili nižeg prioriteta. Ukoliko je tretirani prekid došao

proizvodi prekid mora biti od dva PICa koji su PICa nazivaju se IRQ i imaju od 0 do 15. Na taj način, obzirom da se taj prekid generiše u dva PICa. Intel platforma koristi vezani na odgovarajuće portove, mogu se vezati PIR (Parallel I/O) čipovi koji imaju sposobnost generisanja prekida koji se nalaze na manevrinskim raspolažanjima. PIR kontroler je uključen u instrukciju sa kojom se registri PIR kontrole.

Kontroler sa procesorom, u ovom slučaju po jednu, dok linije na nivou svog IRQa svašto treba da predstavlja na DL linija opisuju ovaj do-

prekid. Nakon toga, kada je

sa vanjskim uređajima

Kad  
maskin

sa ulazne linije od master PICa, EOI se šalje samo master PICu. U suprotnom, EOI se šalje na oba PIC kontrolera.

Za komunikaciju sa master PICom koriste se portovi 0x20 i 0x21, a za slave PIC definirani su portovi 0xa0 i 0xa1. Putem ovih portova moguće je slati komande za konfiguraciju i promjenu načina rada PICa.

Režim rada PICa definiran je sa tri 8-bitna registra. Svaki bit u ovim registrima odgovara jednoj ulaznoj liniji za prekide na tom PICu. Registri se označavaju kao:

#### **IMREG (Interrupt Mask Register)**

Registrar omogućava operaciju maskiranja prekida kojom PIC prestaje slati prekide sa određenih ulaznih linija. Za onemogućavanje prekida sa neke linije, bit asociran sa tom ulaznom linijom u IMREGu treba biti postavljen na 1. Ovaj register se može mijenjati preko portova PICa.

#### **IRREG (Interrupt Request Register)**

Putem ovog registra, PIC vodi računa o ulaznim linijama na kojim je signaliziran prekid koji čeka na procesiranje. Registrar nije moguće mijenjati putem portova.

#### **ISREG (In-Service Register)**

Putem ovog registra, PIC vodi računa o ulaznim linijama na kojim je signaliziran prekid koji je u fazi procesiranja, tj. za njega još nije stigao EOI signal. Registrar nije moguće mijenjati putem portova.

U radu PIC kontrolera koristi se šema fiksnih prioriteta. Viši prioritet imaju prekidi koji potiču sa IRQ linija sa nižim rednim brojem. Obzirom da se apliciraju na ulaz irq2, svi prekidi sa slave PICa imaju veći prioritet u odnosu na prekide sa master PICa koji potiču od linija irq3 do irq7.

1. A  
P  
2. U  
L  
3. U  
P  
I

Kad  
u ISRI  
višeg  
tualno

Po  
putem  
ljene  
status  
ovog  
na sta  
trukci  
že se  
instru  
puten  
je. U  
vrijed  
tim p  
jedno  
se ign

Kada na nekom IRQu nastane prekid, pod uslovom da linija nije maskirana u IMREGu, obavljaju se slijedeći koraci:

1. Aktivira se bit u IRREGu asociran sa linijom koja je proizvela prekid.
2. Ukoliko stanje ISREGa upućuje da se trenutno servisira prekid istog ili višeg prioriteta, ništa se ne poduzima.
3. Ukoliko stanje ISREGa upućuje da se trenutno servisiraju prekidi nižeg prioriteta, na INT liniji se proizvodi novi prekid. Nakon ovog, PIC čeka na potvrdu prekida.
  - Kada procesor na INTA liniji potvrdi prekid, bit u IRREGa asociran sa linijom koja je proizvela prekid se gasi, dok se bit sa istim rednim brojem u ISREGu pali.

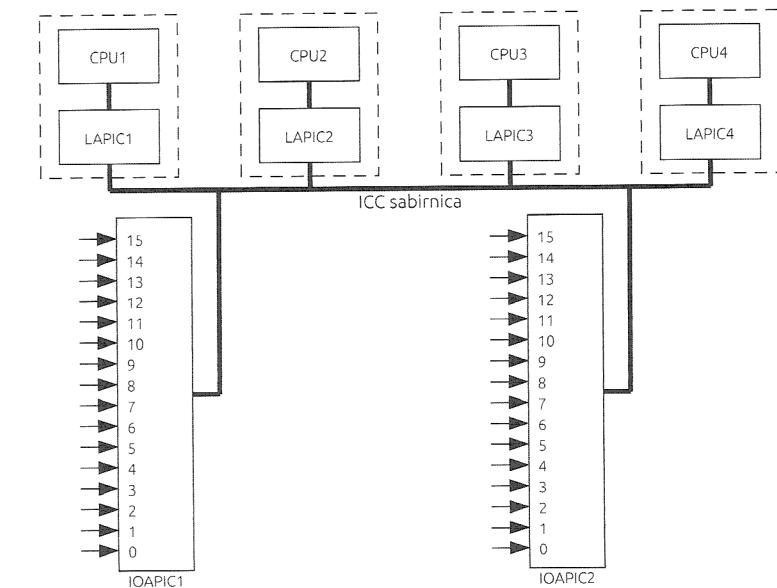
Kada putem mapiranih portova dobije EOI signal, PIC kontroler u ISREGu gasi jedan od aktivnih bita koji je asociran sa IRQom najvišeg prioriteta. Ukoliko je ugašen poslednji bit ISREGa, PIC eventualno proizvodi novi prekid spram zatečenog stanja IRREGa.

Pored mogućnosti maskiranja prekida pojedinačnih IRQ linija putem IMREGa PICa, procesor može maskirati sve prekide primljene na ulaznoj liniji INTR, promjenom polja IF (Interrupt Flag) u statusnom registru EFLAGS. Polje IF asociрано је са деветим bitom ovog регистра. Ukoliko је полje IF активно, процесор прими prekide на стандардан начин. У suprotnom, nakon izvršavanja svake instrukcije, процесор више не проверава стање INTR линије. Поле IF може се експлицитно активирати са инструкцијом sti, а деактивирати путем инструкције cli. Dodatno, могуће је постављање овог поља са стека путем инструкције popf. Инструкције cli и sti су sensitivne инструкције. Уколико се покуша извршење ових инструкција док CPL има већу vrijednost у односу на IOPL, генерира се GPF. Уколико се при истим предпоставкама путем инструкције popf покуша промјенити vrijednost kompletног регистра EFLAGS, eventualna промјена поља IF се ignorira.

## APIC

Centralno mjesto u modernim računarima zauzima procesor sa dva ili više jezgra (core). Svako jezgro predstavlja potpuno kvalifikovani procesor koji ima sopstvene: registre, keš i MMU. Jezgra se spajaju na zajedničku sabirnicu i operiraju po principima SMPa (Symmetric Multiprocessing), obzirom da ravnopravno dijele pristup jednoj memoriji i svim vanjskim uređajima. SMP sistem omogućava pravi paralelizam. Svako jezgro u datom trenutku može biti angažovano na izvršavanju koda bilo koje aplikacije koja je učitana u memoriju, ili na servisiranju prekida od različitih uređaja.

PIC sistem može da proslijedi prekide samo jednom procesoru, zbog čega nije adekvatan za SMP računare. Ovo je glavni razlog zbog čega je Intel razvio APIC šemu za distribuciju prekida koja je prikazana na slici 4-4.



Slika 4-4. Višeprocesorski sistem i kontroleri prekida

Kod APIC šeme, svako jezgro ima lokalni kontroler prekida označen kao LAPIC. Linije prekida vanjskih uređaja povezuju se sa vanjskim kontrolerom prekida koji se označava sa IOAPIC. Sistem

može da se  
kazana dve  
kontroleri  
nim LAPIC  
APIC s  
ulaznim  
ulaznu lin  
putem IC  
formacija  
gurirati  
se konfig  
proslijed  
Zadat  
situacija  
jezgro:

1. Svaki  
povez  
senzori  
Kada  
uređaji  
se vrat  
2. Kad  
izvode  
sa vrat  
3. Zbroj  
kontrol  
putem  
koji  
da n  
4. Sva  
prekida  
Intel

može da sadrži proizvoljan broj IOAPIC kontrolera. Na slici su prikazana dva IOAPICa, koji imaju po 16 linija za prekide. Vanjski kontrolери su putem separatne ICC sabirnice povezani sa pojedinačnim LAPIC kontrolerima od četiri jezgra u procesoru.

APIC specifikacija unaprijed ne definira raspored uređaja na ulaznim linijama prekida IOAPICa. Kada neki uređaj povezan na ulaznu liniju IOAPICa proizvede prekid, informacija o prekidu se putem ICC sabirnice proslijeđuje nekom LAPICu. Proslijedena informacija uključuje i vektor prekida čija se vrijednost može konfigurirati za svaku ulaznu liniju pojedinačno. Dodatno, IOAPIC može se konfigurirati tako da prekide koji potiču sa određene ulazne linije proslijeđuju jednom jezgru ili određenoj grupi procesorskih jezgri.

Zadatak LAPICA je da prekida procesor po potrebi. Postoji više situacija zbog kojih LAPIC može proizvesti prekid za procesorsko jezgro:

1. Svako jezgro integrira određene uređaje čije se linije prekida povezuju na lokalne ulaze od LAPICA. Na primjer, tajmer i senzor temperature su obavezno prisutni u svakom jezgru. Kada LAPIC na lokalnoj ulaznoj liniji primi signal od nekog uređaja, procesor se prekida sa odgovarajućim vektorom čija se vrijednost može konfigurirati.
2. Kada preko ICC sabirnice dobije informaciju o prekidu čiji je izvor linija prekida nekog IOAPICA, LAPIC prekida procesor sa vektorom dobivenim od IOAPICA.
3. Zbog kompatibilnosti, sistem može uključivati i klasični PIC kontroler prekida. U tom slučaju, jedan od LAPIC kontrolera putem LINT0 i LINT1 ulaza može da prima prekide od uređaja koji su povezani na PIC sistem. LAPIC se može konfigurirati da maskira ove prekide.
4. Svako jezgro putem svog LAPICA može proizvesti međuprocesorski prekid koji se označava kao IPI (Inter-Processor Interrupt). IPI uključuje identifikator odredišnog jezgra koje

treba da primi prekid. Izvođeni LAPIC putem ICC sabirnice šalje IPI odredišnom LAPICu koji na svom jezgru proizvodi prekid, i to sa vektorom prekida čija je vrijednost pročitana iz IPI poruke.

Registri za konfiguraciju LAPICA i IOAPICA su memorijski mapirani u fizičkom memorijskom prostoru rezerviranom za uređaje iznad 3GB. Tačne lokacije mapiranja nalaze se u MP (Multi-Processor) tabelama koje BIOS kreira i konfigurira u svojoj memoriji ispod 1MB. MP tabele sadrže informacije o LAPIC i IOAPIC kontrolerima, uređajima u sistemu, raspoloživim jezgrima itd. Tačna adresa od koje počinje MP struktura može se odrediti skeniranjem BIOS memorije ispod 1MB spram niza karaktera "\_MP\_".

BIOS u MP tabeli obično konfigurira baznu adresu od koje počinju LAPIC registri na vrijednost 0xFEC0\_000H, a za IOAPIC od lokacije 0xFEE0\_000H. Svako jezgro na istoj adresi može pristupati registrima sopstvenog LAPICA, dok mu registri LAPIC kontrolera ostalih jezgri nisu vidljivi. Sva jezgra mogu pristupati registrima IOAPIC kontrolera u sistemu i to spram bazne adrese definirane u MP tabeli.

Bilo koju ulaznu liniju prekida IOAPICA, kontroler putem regista IOREDTBL (IO Redirection Table) može asocirati sa bilo kojim vektorom prekida, kao i sa jezgrom na koji se šalje prekid sa date linije. Za razliku od PIC sistema gdje su prioriteti vezani za broj IRQ linije, IOAPIC prioriteti definirani su putem gornja četiri bita vektora prekida, pri čemu veća vrijednost predstavlja viši prioritet.

Kao i kod PIC sistema, po završetku procesiranja određenog prekida, jezgro slanjem EOI poruke mora obavijestiti sopstveni LAPIC da je završeno procesiranje prekida. Kroz komunikaciju sa LAPIC i IOAPIC registrima, procesor može maskirati pojedinačne prekide koji potiču sa određenih ulaznih linija. Svi prekidi mogu biti istovremeno maskirani direktno na procesoru putem IF polja EFLAGS regista, i to instrukcijama cli i popf.

Prilikom pokretanja sistema, a nakon što detektuje sve uređaje i jezgre, BIOS jednom jezgru dodjeljuje ulogu BSP (Bootstrap Processor) jezgre. Ostale jezgre u sistemu označavaju se kao AP (Application Processors) jezgre. AP jezgre se stavlja u zaustavljeni stanje, u kojem ne izvršavaju nikakav kod, ali imaju mogućnost primanja IPI prekida. Zadatak BSP jezgre je da završi inicijalizaciju sistema i da, u okviru sekvence pokretanja operativnog sistema, putem IPI prekida probudi ostale jezgre. Nakon pokretanja, operativni sistem za sve jezgre definira konkretnе zadatke.

## Procesiranje prekida u protected modu

Postoje tri varijante prekida na IA32 platformi:

### Iznimke

Predstavljaju prekide koji nastaju sinhrono tokom izvršavanja aplikacija. Ovaj tip prekida proizvodi se implicitno u situacijama:

- kada aplikacija pokuša izvršiti neku instrukciju za koju nema dovoljno privilegija,
- kada nije moguće proizvesti validan rezultat po izvršenju određene instrukcije,
- kada aplikacija pokuša izvršiti nelegalnu operaciju.

### Sistemski pozivi

Predstavljaju prekide koji nastaju sinhrono tokom izvršavanja aplikacija. Proizvode ih aplikacije eksplicitnim pozivom instrukcije int. Ova instrukcija omogućava da aplikacija nižeg nivoa privilegija koristi neku funkcionalnost koju obezbeđuje aplikacija sa višim nivoom privilegija.

### Prekidi vanjskih uređaja

Ovi prekidi nastaju asinhrono, a proizvode ih uređaji povezani na PIC ili APIC sistem.

Sa stanovišta koda aplikacije, sistemski pozivi su namjerni, a iznimke obično nastaju kao rezultat pogreške u kodu. Aplikacije nemaju nikakvu kontrolu nad prekidima koji dolaze od vanjskih uređaja. Bez obzira na razlog nastanka, prekid mora biti tretiran putem adekvatne ISR rutine. Procesor bira ISR rutinu koja će tretirati prekid, spram vektora koji je parametar prekida.

Intel platforma iznimkama dodjeljuje fiksne vektore prekida. Za tu namjenu odvajaju se vektori u intervalu od 0 – 31.

**Tabela 4-1.** Vektori prekida iznimki

Vektor	heks.	Ime iznimke
0	0x00	Divide error
1	0x01	Debug exception
2	0x02	Nonmaskable interrupt
3	0x03	Breakpoint
4	0x04	Overflow
5	0x05	Bounds check
6	0x06	Invalid opcode
7	0x07	Coprocessor not available
8	0x08	Double fault
9	0x09	Coprocessor segment overrun
10	0x0A	Invalid TSS
11	0x0B	Segment not present
12	0x0C	Stack exception
13	0x0D	General protection violation
14	0x0E	Page fault

Vektor	heks.
15	0
16	0
17	0
...	...
31	0

Iznimke pogreški rutine generiraju kom tretuju.

Vektori iznimki su generirani sa II.

Za sisteme obično parametri iznimki sa njenog sistema se konveruju.

Bez obzića uključuju:

1. Smjer
2. Određenje

namjerni, a iz-  
kodu. Aplikacije ne-  
od vanjskih ure-  
titi tretiran putem  
koja će tretirati pre-

vektore prekida. Za  
31.

Vektor	heks.	Ime iznimke
15	0x0F	Reserved
16	0x10	Coprocessor error
17	0x11	Reserved
...	...	...
31	0x01F	Reserved

Iznimke sa vektorima 13 i 14 najčešće se generiraju kao rezultat pogreški u kodu aplikacija. Ukoliko u kodu koji je sastavni dio ISR rutine koja tretira neku iznimku nastane nova iznimka, procesor generira DF (Double Fault) iznimku čiji je vektor 8. Ukoliko prilikom tretmana DF iznimke nastane nova iznimka, procesor se resetuje.

Vektori u prethodnoj tabeli fiksno su dodjeljeni asociranim iznimkama. Zbog toga se kontroleri prekida konfigurišu tako da prekidi sa IRQ linija dobiju odgovarajuće vektore iznad 32.

Za sistemski poziv, operativni sistemi kao privilegovane aplikacije obično izdvajaju jedan vektor. Taj se vektor koristi kao jedini parametar u `int` asembler instrukciji, i to u situacijama kada aplikacije sa nižim nivoom privilegija zahtijevaju neki servis od operativnog sistema. Proslijedivanje argumenata sistemskog poziva reguliše se konvencijom koju definira konkretni operativni sistem.

Bez obzira na razloge nastanka, procesiranje generiranog prekida uključuje:

1. Snimanje na stek stanja programa koji se izvršavao u trenutku nastanka prekida.
2. Odabir ISR za tretman nastalog prekida naspram njegovog vektora.

3. Adekvatnu modifikaciju privilegija procesora.
4. Tretman prekida izvršenjem odabranog ISR.
5. Ukoliko ima smisla, preuzimanje stanja prekinutog programa sa steka i nastavak izvršenja programa sa njemu dodjeljenim nivoom privilegija.
  - Program se uvijek nastavlja nakon tretmana prekida od vanjskih uređaja ili kada je prekid izazvao sam program putem `int` instrukcije.
  - U okviru tretmana iznimki, prekinuti program se najčešće u potpunosti terminira.

Za implementaciju navedenih koraka u hardveru, pored tabele sa segmentnim deskriptorima, potrebna je još jedna tabela koja se označava kao IDT (Interrupt Descriptor Table). IDT sadrži informacije potrebne da se tretiraju svi prekidi u sistemu. Red u IDTu predstavlja 64-bitni deskriptor prekida (Interrupt Gate) i sadrži polja neophodna za procesiranje prekida čiji je vektor redni broj datog reda.

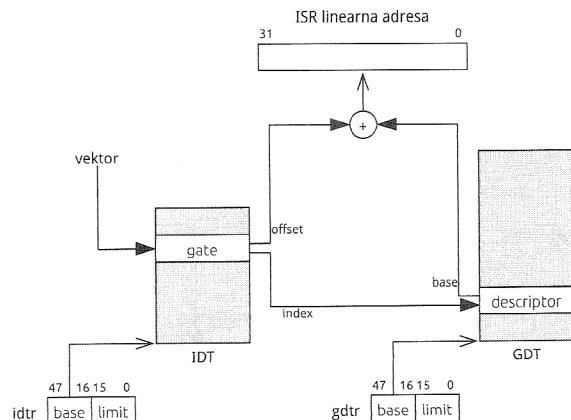
Procesor čuva informaciju o lokaciji IDTa u posebnom 48-bitnom registru `idtr` koji se sastoji od dva polja, i to:

- 32-bitno polje `base` u kojem se nalazi adresa početka IDTa,
- 16-bitno polje `limit` u kojem se nalazi veličina tabele u bajtima.

Sadržaj registra može se učitati sa neke memorijske lokacije putem `lidt` instrukcije.

Slika 4-5 prikazuje proces određivanja adrese ISR rutine koja se koristi za tretman nekog prekida.

Nastankom prekida, procesor dobija vektor kojeg koristi za indeksiranje IDTa. Ovim se pronađazi `gate` asociran sa datim vektorom.



Slika 4-5. Određivanje linearne adrese za ISR

rom u kojem se nalaze dva ključna polja, 32-bitni **index** i 16-bitni **offset**. Polje **index**, procesor koristi za indeksiranje GDT tabele čime se dobija novi segmentni deskriptor asociran sa CS selektorom. Keširanjem novog deskriptora u CS selektoru, efektivno se mijenja nivo privilegija procesora i to spram nove vrijednosti DPL polja CS keširanog deskriptora. Adresa prve instrukcije ISR rutine kojom se tretira prekid računa se kao suma polja **base** iz CS keširanog deskriptora i polja **offset** koje je dobiveno iz indeksiranog reda u IDTu. Procesor potom počinje izvršavati instrukcije sa proračunate adrese za ISR.

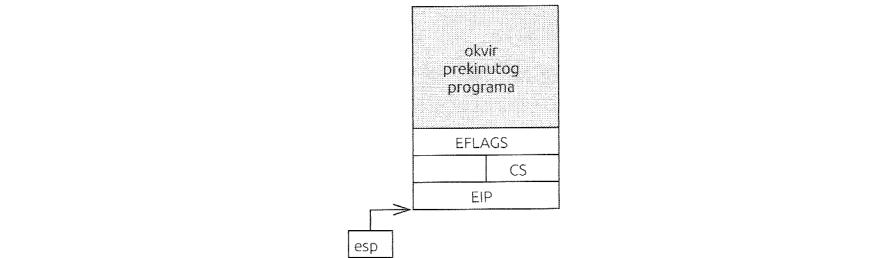
Polje **type** u odabranom redu IDTa reguliše maskiranje svih prekida kada počne izvršenje ISR rutine. Ukoliko je postavljeno na tip **interrupt gate**, ISR rutina se izvršava sa maskiranim prekidima, tj. IF zastavica EFLAGS registra će biti ataktivirana. Prekidi neće biti maskirani tokom izvršenja ISR rutine ako je polje postavljeno na vrijednost **trap gate**.

Polje **DPL** u bilo kojem redu tabele određuje nivo privilegija potrebnih da se vektor asociran sa datim redom koristi sa instrukcijom **int**. Kod konfiguracije IDTa, operativni sistemi postavljaju DPL polje u redovima asociranim sa vektorima dodjeljenim uređajima ili iz-

nimkama na 0, dok se u onom redu koji je asciran sa vektorom do-djeljenom sistemskom pozivu, DPL polje postavlja na 3.

Prije nego počne izvršavati odabranu ISR rutinu, procesor automatski snima stanja određenih registara koje je koristio prekinuti program u trenutku kada je nastao prekid. Registri se snimaju na stek, a sadržaj koji je snimljen na steku ovisi od tipa prekida.

Ukoliko se prekid dogodio dok se izvršavao program sa nultim nivoom privilegija, za snimanje registara koristi se postojeći stek. Slika 4-6 prikazuje stek kojeg u većini slučajeva ISR zatiče kada počne sa izvršavanjem radi tretmana prekida. Ukoliko prekid nastane zbog iznimki sa vektorima: 8, 10, 11, 12, 13, i 14, stek će sadržavati pored registara i kod greške (Error Code) koji za svaku od navedenih iznimki nosi dodatne informacije koje je opisuju. Izgled steka za ovaj slučaj prikazan je na slici 4-7. Ukoliko je prekid nastao zbog iznimke sa vektorom 14 (Page Fault), kontrolni registar cr2 sadrži linearnu adresu kojoj je program pokušao pristupiti kada je nastala iznimka u straničenju.



Slika 4-6. ISR stek nakon sistemskog poziva ili prekida od uređaja

Ukoliko se prekid dogodio dok se izvršavao program sa prioritetom nižim od nultog, za snimanje registara koristi se poseban stek. Slika 4-8 prikazuje elemente koji su uključeni u određivanje adrese steka koji će koristi ISR za ovaj slučaj.

Ukoliko je predviđena mogućnost izvršavanja programa sa nižim prioritetom od nultog, što je slučaj kod većine operativnih sistema,

Slika 4-7. IS

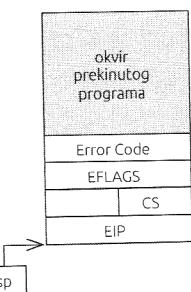


Slika 4-8.

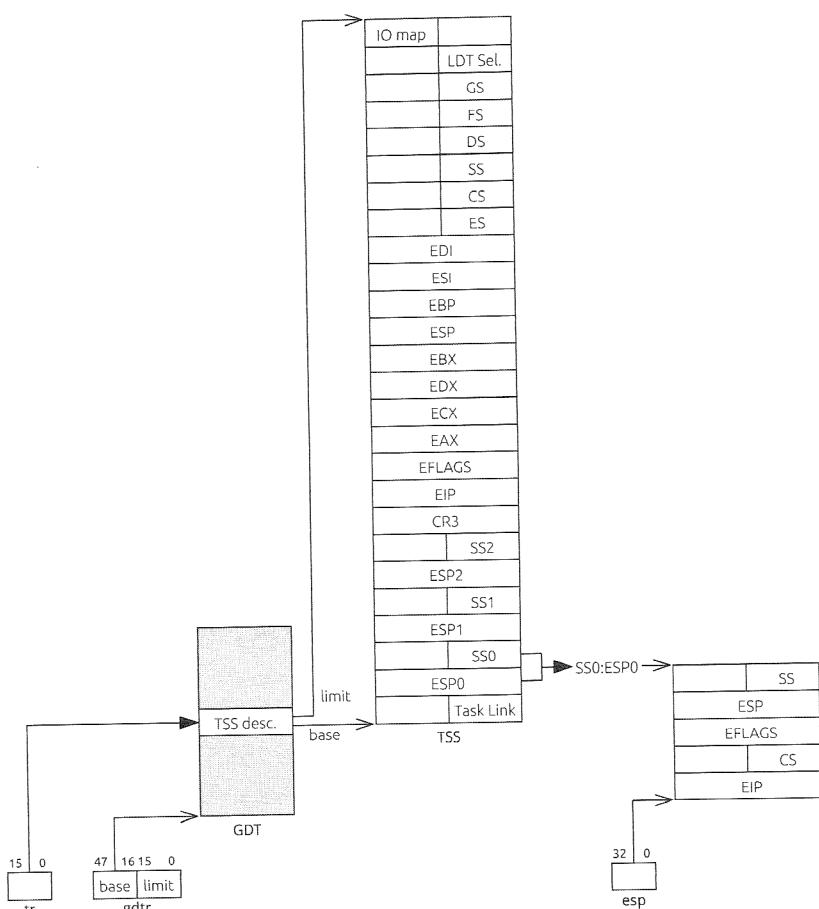
ran sa vektorom do-  
vana 3.

nu, procesor auto-  
koristio prekinuti  
egistri se snimaju na  
lupa prekida.

program sa nultim  
se postojeći stek.  
ISR zatiče kada po-  
preko prekid nastane  
14, stek će sadržava-  
svaku od nave-  
isuju. Izgled steka  
prekid nastao zbog  
registar cr2 sadrži  
spiti kada je nastala



Slika 4-7. ISR stek nakon određenih iznimki



Slika 4-8. ISR stek nakon prekida programa sa nižim nivoom privilegija

na bilo kojoj lokaciji u memoriji mora biti definirana posebna struktura pod nazivom TSS (Task Switch Segment). Kao što je vidljivo na slici, TSS ima veliki broj polja, od kojih su sa stanovišta ISR rutine jedino bitna polja  $SS_0$  i  $ESP_0$  kojim se određuje logička adresa početka steka kada nastane prekid dok se izvršava program sa nižim nivoom privilegija. Da bi procesor mogao locirati tačnu lokaciju TSS strukture u memoriji, kompletan TSS mora biti u zasebnom segmentu koji se opisuje posebnim deskriptorom u GDTu. Tip deskriptora koji se koristi za ovu namjenu mora biti TSS descriptor. TSS deskriptor ima standardna polja, od kojih base određuje linearnu adresu početka TSS strukture u memoriji, a limit određuje veličinu TSS-a u bajtima. Procesor ima poseban 16-bitni registar  $tr$ , koji predstavlja indeks reda GDTa u kojem se nalazi TSS deskriptor. Sadržaj ovog registra može se učitati putem instrukcije  $ltr$ .

Kada nastane prekid programa koji ima niži nivo privilegija, procesor u posebne nevidljive registre kopira vrijednosti registara  $ss$  i  $esp$  koji određuju trenutnu lokaciju steka prekinutog programa. Na osnovu registra  $tr$ , procesor pronađe TSS deskriptor. Zatim, preko polja  $base$  u deskriptoru dolazi do TSS strukture iz koje preuzima polja  $SS_0$  i  $ESP_0$ . Pročitana polja, procesor učitava u registre  $ss$  i  $esp$ , čime je efektivno promjenjena adresa steka koji se trenutno koristi. Procesor zatim na novi stek prvo iz nevidljivih registara kopira vrijednosti  $ss$  i  $esp$  registara prekinutog programa, a zatim na *stek stavlja i registre eflags, cs i eip*. *Ovim je pripremljen novi stek za ISR rutinu.*

Prije nego što započne tretman prekida, ISR rutina obično na stek snima sve ostale registre prekinutog programa. Ovo se čini da bi nakon tretmana prekida, program eventualno mogao nastaviti sa izvršenjem. Kada se nakon prekida nastavlja izvršenje prekinutog programa, ISR prvo na procesor vraća sve registre programa koji su snimljeni u okviru ISR rutine, a zatim putem jedne instrukcije  $iret$  na procesor istovremeno vraća vrijednosti svih registara koji su automatski snimljeni na stek prilikom nastanka prekida. Ukoliko

je tretman prekida uključivao promjenu nivoa privilegija procesora, instrukcija iRET samostalno detektuje takve situacije i sa steka na procesor, uz registre EFLAGS, CS i EIP, vraća i registre SS i ESP. Ovim je procesor vraćen u identično stanje kojeg je imao neposredno pred nastanak prekida, a program može da nastavi sa svojim izvršenjem kao da nije ni prekidan.

## Učitavanje kernela

Kada je XV6 sistem u potpunosti pokrenut, na ekranu se pojavljuje prompt u kojem korisnik može unositi komande. U tom trenutku korisnik uopšte ne komunicira sa XV6 kernelom, već sa aplikacijom `sh` putem koje se mogu pokretati ostali programi koji su nalaze u XV6 fajl sistemu. Pored `sh`, aktiviran je još jedan program pod imenom `init` koji ima dva zadatka:

- da pokrene i održava učitanim program `sh`,
- da u određenim situacijama dealocira resurse od programa koji će biti pokretani u sistemu.

Da bi se stvorile pretpostavke za paralelno izvršavanje dvije inicijalne aplikacije, od koji jedna ima i sposobnost da komunicira sa korisnikom preko monitora i tastature, potrebno je sprovesti kompleksnu sekvencu koraka koja se označava kao proces pokretanja operativnog sistema, ili *boot* proces. Ovaj proces uključuje:

- detekciju i inicijalizaciju prosesorskih jezgri, kao i kontrolera vanjskih uređaja.
- tranziciju procesora u protected mod,
- konfiguraciju segmentiranja,
- učitavanje kernela,
- aktiviranje i konfiguraciju stranicenja,
- konfiguraciju uređaja,
- pokretanje dodatnih jezgri.

Nabrojane korake možemo podijeliti u dvije faze pokretanja sistema. Dok u drugoj fazi pokretanja učestvuje samo kernel operativnog sistema, u prvoj fazi učestvuju tri aktera:

1. Program BIOS, koji je pohranjen u permanentnoj memoriji matične ploče.
2. Program označen kao *bootloader*, koji se nalazi na specijalnoj lokaciji diska koji služi za pokretanje sistema.
3. Kernel operativnog sistema.

U ovom poglavlju analizirat ćemo sve korake prve faze pokretanja sistema. Primarni fokus će biti na programu koji učitava kernel i na inicijalnoj konfiguraciji segmentiranja i straničenja.

## Prva faza pokretanja sistema

Nakon što dobije napajanje, procesor počinje izvršavati instrukcije is segmenta fizičke memorije u kojem je mapiran BIOS program. BIOS detektuje i inicijalizira: memoriju, kontrolere uređaja spojenih na vanjsku sabirnicu i jezgra u procesoru. Rezultat ovog procesa se zapisuje u BIOS memorijskom prostoru, i to u obliku MP tabela. Nakon BIOS inicijalizacije, samo BSP jezgro procesora ostaje aktivno i dobija zadatak da izvrši pokretanje operativnog sistema.

BIOS ima i poseban mod rada u kojem dopušta da korisnik odabere određene postavke konfiguracije BIOS-a. Između ostalog, korisnik može i da odabere uređaj, a najčešće tvrdi disk, sa kojeg će se izvršiti pokretanje operativnog sistema. Diskovi se organiziraju u sektore, kontinuirane blokove za snimanje podataka sa kapacitetom od 512 bajta. Pojedinačni sektori se adresiraju na sličan način kako se adresiraju pojedinačni bajti u memoriji. Prvi sektor na disku ima specijalni status. Da bi se disk mogao koristiti za pokretanje operativnog sistema, u prvom sektoru diska mora biti zapisan program koji ima sposobnost da u memoriju učita kernel koji se nalazi na nekoj lokaciji na disku.

te faze pokretanja sistema samo kernel operativnog

permanentnoj memoriji

se nalazi na specijalnoj

prve faze pokretanja koji učitava kernel i smanjenja.

izvršavati instrukcije izvršen BIOS program. Uređaj uređaja spojenih rezultat ovog procesa se obliku MP tabela. Procesora ostaje aktivnovognog sistema.

da korisnik odaš. Između ostalog, koje tvrdi disk, sa kojeg će Diskovi se organiziraju u podataka sa kapacitetom na sličan način kako se sektor na disku ima za pokretanje operativnog biti zapisan program koji se nalazi na ne-

Kada završi sa procesom inicijalizacije, BIOS sa diska odabranog za pokretanje operativnog sistema na lokaciji 0x7c00 u memoriji učitava sadržaj prvog sektora diska. Kada završi učitavanje, BIOS izvršava instrukciju jmp na lokaciju u memoriji 0x7c00. Ovim BIOS predaje kontrolu programu za pokretanje operativnog sistema. U trenutku predaje kontrole, procesor je u real modu operacije sa uključenim prekidima. Učitani program ima zadatku da prebacuje procesor u protected mod operacije. U tu svrhu u memoriji mora kreirati i konfigurirati GDT tabelu za segmentiranje. Dodatno, program za pokretanje mora na neki način pronaći lokaciju na disku na kojoj se nalazi kernel, izvršiti učitavanje kornela na adekvatnu lokaciju u memoriji, a zatim predati kontrolu učitanom kernelu.

Iako postoje programi koji se mogu koristiti za pokretanje više različitih operativnih sistema sa jednog diska, npr. Grub ili SysLinux, XV6 uključuje poseban program za pokretanje svog kornela. Ovaj program podrazumijeva da je XV6 kernel kompajliran u ELF formatu i da će biti zapisan na istom disku, počevši od drugog sektora. Zbog praktičnosti, XV6 operativni sistem obično se izvršava pod emulatorom računara sa Intel procesorom u konfiguraciji sa dva diska. Na prvom disku se nalaze: program za pokretanje operativnog sistema i XV6 kernel. Drugi disk, čiji je sadržaj organizovan sa posebnim XV6 fajl sistemom, sadrži aplikacije koje se mogu izvršavati pod kontrolom XV6 kornela, i to nakon okončanja njegovog pokretanja. Umjesto fizičkih diskova, emulator koristi dva fajla čiji je sadržaj definiran putem Makefile skripte. Slijedeći segment iz skripte odnosi se na kreaciju prvog diska, na kojem se nalaze program za pokretanje i kernel.

```
xv6.img: bootblock kernel fs.img ①
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc ②
#
# ...
bootblock: bootasm.S bootmain.c ③
    $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c ④
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S ⑤
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7c00 \
```

```

-o bootblock.o bootasm.o bootmain.o ⑥
$(OBJDUMP) -S bootblock.o > bootblock.asm
$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock ⑦
./sign.pl bootblock

```

Linije od **①** do **②** sadrže upute za kreiranje fajla `xv6.img` koji se koristi kao prvi disk prilikom simulacije. Prvo se putem programa `dd` kreira fajl veličine 5MB koji je kompletan popunjen sa nulama. Sadržaj fajla se potom mijenja dva puta. Pri prvoj promjeni, sadržaj fajla `bootblock`, putem programa `dd`, upisuje se na sami početak fajla `xv6.img`. Nakon toga, sadržaj fajla `kernel`, ponovo putem programa `dd`, upisuje se u `xv6.img`, i to nakon prvog bloka od 512 bajta. Fajlovi koji se upisuju u sliku diska koja će se koristiti u procesu simulacije su:

- kompajlirani kernel XV6 operativnog sistema, u fajlu `kernel`,
- program za učitavanje kernela, u fajlu `bootblock`.

Fajl `bootblock`, nastaje na osnovu uputa datih između linija **③** i **⑦**. Prvo se u linijama **④** i **⑤** kreiraju dva objektna fajla `bootmain.o` i `bootasm.o`. Prvi fajl nastaje kompajliranjem `bootmain.c`, dok drugi fajl nastaje asembleriranjem `bootasm.S`. Objektni fajlovi u liniji **⑥**, putem linkera, uvezuju se u ELF fajl pod imenom `bootblock.o`, i to na način da se sve instrukcije u zajedničkoj `.text` sekcijsi odredišnog fajla organizuju tako da će sekcija biti učitavana u memoriju od lokacije `0x7c00`. U odredišni fajl, spram redoslijeda datog u liniji **⑥**, prvo se uvezuju sekcije iz `bootasm.o`, a zatim sekcije iz `bootmain.o`. Binarni sadržaj `.text` sekcije iz odredišnog ELF fajla izdvaja se u zaseban fajl `bootblock`, u liniji **⑦**.

Prva komponenta programa `bootblock` napisana je u asembleru i ima primarnu zadaću da izvrši tranziciju procesora u protected mod operacije. Slijedeći listing preuzet je iz fajla `bootasm.S` u kojem je implementirana ova komponenta:

```

#include "asm.h"
#include "memlayout.h"

```

6  
bootblock 7

je datota xv6.img koji se  
se putem programa  
popunjeno sa nulama.  
promjeni, sadržaj  
nasami početak fajla  
putem programa  
512 bajta. Fajlovi  
procesu simulacije

u fajlu kernel,  
bootblock.

medju linija 3 i 7.  
datota bootmain.o i bo-  
main.c, dok drugi fajl  
u liniji 6, putem  
bootblock.o, i to na na-  
određeni određenog fajla  
memoriju od lokacije  
liniji 6, prvo se  
bootmain.o. Binarni  
zvaja se u zaseban fajl

pisana je u asembleru i  
u protected mod  
asm.S u kojem je im-

```
#include "mmu.h"
.code16
.globl start
start:
    cli ①
    xorw %ax,%ax
    movw %ax,%ds
    movw %ax,%es
    movw %ax,%ss

seta20.1:
    inb $0x64,%al
    testb $0x2,%al
    jnz seta20.1
    movb $0xd1,%al
    outb %al,$0x64

seta20.2:
    inb $0x64,%al
    testb $0x2,%al
    jnz seta20.2
    movb $0xdf,%al
    outb %al,$0x60

    lgdt gdtdesc ②
    movl %cr0, %eax
    orl $CR0_PE, %eax
    movl %eax, %cr0 ③
    ljmp $(SEG_KCODE<<3), $start32 ④

.code32
start32:
    movw $(SEG_KDATA<<3), %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %ss
    movw $0, %ax
    movw %ax, %fs
    movw %ax, %gs
    movl $start, %esp ⑤
    call bootmain ⑥
    movw $0x8a00, %ax
    movw %ax, %dx
    outw %ax, %dx
    movw $0x8ae0, %ax
    outw %ax, %dx
spin:
    jmp spin

.p2align 2
gdt: ⑦
    SEG_NULLASM
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)
    SEG_ASM(STA_W, 0x0, 0xffffffff)

gdtdesc: ⑧
```

```
.word  (gdtdesc - gdt - 1)
.long  gdt
```

Na samom početku fajla nalaze se predprocesorske direktive za uključivanje fajlova `asm.h`, `memlayout.h` i `mmu.h`, u kojim se definirani makroi koji se kasnije koriste u asembler kodu.

Asembler kod se može podijeliti u dva dijela. U prvom dijelu koji počinje sa direktivom `.code16`, a traje do linije ④, koriste se instrukcije kodirane u 16-bitnom modu, što odgovara modu operacije u kojem se procesor pokreće. BIOS ostavlja uključene prekide prije nego što preda kontrolu programu za pokretanje kornela. Prvom instrukcijom `cli`, gase se prekidi i ostaju u tom stanju tokom cijele procedure pokretanja XV6 kornela. Nakon ovog, segmentni registri `es`, `ss` i `ds` postavljaju se na nulu.

Od oznake `set20.1` do linije ②, sa sekvencom `in` i `out` instrukcija, putem kontrolera tastature koji kontroliše portove `0x60` i `0x64`, sprovodi se procedura aktiviranja linije za adresiranje `A20`<sup>1</sup>, koja je zbog istorijskih razloga inicijalno onesposobljena.

Instrukcijom `lgdt`, register `gdtr` postavlja se da pokazuje na GDT tabelu koja je definirana nakon svih instrukcija unutar sekcije `.text`, i to nakon linije ③. Za definiranje tri 64-bitna deskriptora u tabeli, koriste se predprocesorski makroi `SEG_NULLASM` i `SEG_ASM`. Prvi red u tabeli, u skladu sa Intel konvencijama, je NULL deskriptor koji se popunjava sa nulama. Druga dva deskriptora, kreirana putem `SEG_ASM` makroa, opisuju dva segmenta koji pokrivaju kompletan memorijski prostor od 0 od 4GB. Prvi parametar u makrou `SEG_ASM` su dozvole za deskriptor koji se kreira, drugi parametar je `base` polje deskriptora, a zadnji parametar je `limit` polje deskriptora. Konfiguracija GDTa u kojoj se svi segmenti konfiguiraju sa `base` poljem postavljenim na 0, naziva se `flat` adresiranje. Ovakvim adresiranjem sve logičke adrese emitovane direktno iz instrukcija asem-

---

1. [https://en.wikipedia.org/wiki/A20\\_line](https://en.wikipedia.org/wiki/A20_line)

bler koda, u potpunosti odgovaraju linearnim adresama. Nadalje, sve dok je isključeno straničenje, logičke adresa će biti identične i konačnim fizičkim adresama.

Dakle, GDT uključuje dva deskriptora koji su asocirani sa dva segmenta koji se u potpunosti preklapaju. Razlog korištenja dva segmenta leži u potrebi za različitim dozvolama koje se primjenjuju za pristup memoriji kod različitih segmentnih registara. Drugi deskriptor u tabeli omogućava pristup memoriji radi pisanja i čitanja podataka, obzirom da je postavljena samo jedna dozvola pristupa STA\_W. Ovaj deskriptor koristit će se za sve selektore osim za CS register. Obzirom da CS selektor zahtijeva da njegov segment nema dozvole za pisanje, kreira se još jedan deskriptor koji omogućava pristup memoriji radi čitanja i izvršavanja koda, postavljanjem dozvola STA\_X i STA\_R. Ovim je konfiguriran GDT sa svim deskriptorima neophodnim za normalno funkcioniranje procesora u protected modu operacije. Obzirom da GDT sadži samo jedan validan deskriptor za CS register čije je DPL polje postavljeno na nulti nivo privilegija, nije moguće izvršiti tranziciju na niži nivo privilegija sa ovakvom konfiguracijom GDT-a.

Učitavanjem nove vrijednosti u register gdtr još uvijek nije napravljena nikakva promjena sa stanovišta moda operacije procesora. Protected mode operacije procesora aktivira se tek u liniji ②, eksplicitnim postavljanjem CR0\_PE bita u cr0 registru. Da bi se počelo koristiti adresiranje spram konfiguracije u GDT-u, potrebno je da se nove vrijednosti učitaju u sve segmentne registre, čime će se izvršiti keširanje deskriptora iz GDT-a u odgovarajuće segmetne selektore. Prvo se kešira deskriptor u CS selektoru putem instrukcije ljmp. Prvi parametar u instrukciji je nova vrijednost za CS register, dok drugi predstavlja lokaciju na koju se postavlja programski brojač EIP. Makro SEG\_KCODE ima vrijednost 1, koja se šifta za tri mesta u lijevo, a dobiveni broj upisuje se u register CS. Polje index u registru CS, asocira CS selektor sa drugim redom GDT-a. Pored keširanja GDT deksriptora u CS selektoru, postavljanjem регистра EIP na

lokaciju `start32` izvršen je skok putem kojeg se počinju preuzimati 32-bitne instrukcije. Nakon obavljenog skoka, vrši se asociranje svih neophodnih segmentnih registara sa odgovarajućim deskriptorom iz GDTa. Selektori ES, SS i DS, asociraju se sa deskriptorom broj dva, dok se FS i GS, postavljanjem na vrijednost 0, nadalje ne koriste.

Nakon postavljanja vrijednosti svih segmentnih registara, u liniji ⑤ mijenja se adresa za stek koju je inicijalno postavio BIOS. Nova adresa za stek postavlja se na lokaciju `start`. Obzirom da je `start` prvi simbol u `.text` sekciji koja se učitava na lokaciju 0x7c00, za stek se efektivno alocira 32KB prostora. Nakon toga, instrukcijom `call` poziva se funkcija `bootmain`. Funkcija je definirana u fajlu `bootmain.o` koji je linkerom uvezan u program `bootblock`. Ukoliko se poslije poziva funkcije dogodi povrat u asembler kod, to znači da je u pozvanoj funkciji nastala greška, zbog čega se procesor stavlja u beskonačnu petlju kod oznake `spin`. Pozvana funkcija ima zadatak da učita kernel u memoriju, a zatim da predala kontrolu ulaznoj funkciji u učitanom kernelu.

XV6 kernel je izvršna datoteka u ELF formatu koja je na disku zapisana počevši od drugog sektora. Svaki ELF fajl sa stanovišta linkera sastoji se od sekcija. Sa druge strane, sa stanovišta programa za učitavanje (loader), ELF datoteka organizirana je u programske segmente. Programski segment predstavlja jedan kontinuirani blok u memoriji čiji se sadržaj definiran u ELF fajlu. Za svaki programski segment unutar izvršne datoteke, obezbijeđuje se slijedeće informacije koje su relevantne za učitavanje tog programskog segmenta u memoriju:

- Lokacija početka i količine sadržaja programskog segmenta unutar ELF fajla.
- Lokacija gdje programski segment očekuje da bude učitan u memoriju.
- Veličina programskog segmenta u memoriji.

Na samom početku ELF fajla nalazi se jedan string (magic string) koji je prisutan u svim ELF fajlovima. String je sastavni dio ELF zaglavlja (header) koje ima fiksnu veličinu. Unutar zaglavlja mogu se naći osnovne informacije o izvršnoj datoteci, kao i lokacije u fajlu od kojih počinju druga zaglavlja koja opisuju pojedinačne programske segmenate. Putem programa `readelf`, moguće je pročitati razne informacije iz ELF fajlova. Na primjer, apliciranjem komande `readelf -h kernel`, dobijamo slijedeće informacije pročitane iz zaglavlja ELF fajla XV6 kernela.

## ELF Header:

```
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00 ①
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386 ②
Version: 0x1
Entry point address: 0x10000c ③
Start of program headers: 52 (bytes into file) ④
Start of section headers: 168856 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes) ⑤
Number of program headers: 2
Size of section headers: 40 (bytes)
Number of section headers: 17
Section header string table index: 14
```

- ① String koji potvrđuje da je riječ o ELF datoteci.
  - ② Platforma za koju je program namijenjen.
  - ③ Ulazna tačka u program, tj. lokacija u memoriji od koje počinje prva instrukcija u programu.
  - ④ Lokacija u fajlu, izražena u bajtima nakon ELF zaglavlja, na kojoj se nalaze pojedinačna zaglavlja za programske segmente.
  - ⑤ Veličina pojedinačnog zaglavlja za programske segmente.

Da bi ispravno učitala sve elemente XV6 kernela u memoriju, funkcija `bootmain` mora koristiti relevantne informacije iz ELF zaglavlja. Slijedi implementacija ove funkcije preuzeta iz fajla `bootmain.c`.

```
void bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; ①
    readseg((uchar*)elf, 4096, 0); ②
    if(elf->magic != ELF_MAGIC)
        return; ③

    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(; ph < eph; ph++){ ④
        pa = (uchar*)ph->paddr; ⑤
        readseg(pa, ph->filesz, ph->off); ⑥
        if(ph->memsz > ph->filesz)
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz); ⑦
    }

    entry = (void(*)(void))(elf->entry); ⑧
    entry(); ⑨
}
```

- ① Postavlja se adresa u memoriji ( $0x10000 \rightarrow 64KB$ ) na koju će biti učitano ELF zaglavlj.
- ② Na postavljenu adresu, putem funkcije `readseg`, učitava se 4096 bajta sa lokacije na kojoj XV6 kernel počinje na disku, tj. od drugog sektora.
- ③ U slučaju da nije riječ o ELF fajlu, nema smisla nastaviti dalje učitavanje, zbog čega se vrši povrat u `bootasm.S`.
- ④ Ulazak u petlju koja se aplicira na svakom pojedinačnom programskom segmentu ELF fajla.
- ⑤ Čita se adresa u memoriji na koju trenutni segment želi biti učitan.

- kernela u memoriju, informacije iz ELF za-  
iz fajla bootma-
- ⑥ Kompletan sadržaj trenutnog segmenta veličine `ph->filesz`, pročitan iz ELF fajla od lokacije na kojoj počinje ovaj segment `ph->off`, učitava se u memoriju na lokaciju dobivenu u prethodnom koraku.
- ⑦ U slučaju da je veličina sadržaja segmenta na disku manja od veličine segmenta u memoriji, npr. zbog `.bss` sekcijs, sadržaj segmenta u memoriji koji nije preuzet sa diska popunjava se nulama.
- ⑧ Nakon što su u memoriju učitani svi segmenti iz programa, iz zaglavlja se čita lokacija ulazne funkcije u program.
- ⑨ Pozivom ulazne funkcije, predaje se kontrola upravo učitanim XV6 kernelu.

## Inicijalna konfiguracija straničenja u kernelu

Kada program za pokretanje operativnog sistema preda kontrolu kernelu, procesor je u protected modu operacije u kojem koristi dva segmenta konfiguirirana za flat adresiranje. U tom trenutku procesor radi sa nultim nivoom privilegija, a svaka logička adresa ujedno je i fizička. Da bi analizirali dalje pokretanje operativnog sistema osvrnut ćemo se na proces nastanka ELF datoteke kernela. Slijedeći kod iz Makefile skripte koristi se za kreiranje ELF fajla kernela.

```
OBJS = \ ①
        bio.o\
        console.o\
        exec.o\
        file.o\
        fs.o\
        ide.o\
        ioapic.o\
        kalloc.o\
        kbd.o\
        lapic.o\
        log.o\
        main.o\
        mp.o\
        picirq.o\
        pipe.o\
```

```

proc.o\
spinlock.o\
string.o\
swtch.o\
syscall.o\
sysfile.o\
sysproc.o\
timer.o\
trapasm.o\
trap.o\
uart.o\
vectors.o\
vm.o

# ...

CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing \ ②
-fvar-tracking -fvar-tracking-assignments \
-Of0 -g -Wall -MD -gdwarf-2 -m32 -Werror \
-fno-omit-frame-pointer

# ...

kernel: $(OBJS) entry.o entryother initcode kernel.ld
$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) \ ③
-b binary initcode entryother ④
# ...

```

① Lista objektnih fajlova koji čine kernel, a koji nastaju kompajliranjem istoimenih C fajlova.

Komandne opcije koje se koriste za kompajliranje pojedinačnih C fajlova. Između ostalog, kompajliranje se vrši staticki  
② za 32-bitnu Intel platformu, sa simbolima za debagiranje, bez optimizacije i ovisnosti od standardne biblioteke koja dolazi uz kompajler.

XV6 ELF kernel se formira uvezivanjem svih objektnih fajlova iz liste OBJS sa fajлом `entry.o`, koji nastaje asembleranjem `entry.S`. Uvezivanje se kontroliše putem linker skripte pod imenom `kernel.ld`.

Sadržaju fajla `kernel` dodaju se i dva binarna fajla: `initcode`  
④ i `entryother`, čiju ćemo svrhu analizirati u narednim poglavljima.

Linker skripta `kernel.ld` kontroliše imena i sadržaj sekcija u izlaznom kernel fajlu. Slijedi kompletan sadržaj ove skripte.

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)

ENTRY(_start) ①

SECTIONS
{
    . = 0x80100000; ②
    .text : AT(0x10000) { ③
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }

    PROVIDE(etext = .);

    .rodata : {
        *(.rodata .rodata.* .gnu.linkonce.r.*)
    }

    .stab : {
        PROVIDE(__STAB_BEGIN__ = .);
        *(.stab);
        PROVIDE(__STAB_END__ = .);
        BYTE(0)
    }

    .stabstr : {
        PROVIDE(__STABSTR_BEGIN__ = .);
        *(.stabstr);
        PROVIDE(__STABSTR_END__ = .);
        BYTE(0)
    }

    . = ALIGN(0x1000); ④
    PROVIDE(data = .);

    .data : {
        *(.data)
    }

    PROVIDE(edata = .);

    .bss : {
        *(.bss)
    }

    PROVIDE(end = .);

    /DISCARD/ : { ⑤
        *(.eh_frame .note.GNU-stack)
    }
}
```

```
    }  
}
```

Na početku skripte, opcijama `OUTPUT_FORMAT` i `OUTPUT_ARCH`, linker se instruira da proizvede izlazni fajl u ELF formatu za IA32 platformu. Zatim se u liniji ❶ određuje da vrijednost simbola `_start` буде ulazna tačka u ELF zaglavljtu. Ovaj simbol ће бити дефиниран у једном од објектних фај洛va који се увежују у kernel.

Blok `SECTIONS`, дефинира све секције које ће бити присутне у излазном ELF фајлу. Пojedinačна секција дефинира се у формату:

```
.ime_sekcije : {  
    sadrzaj_sekcije!  
}
```

Kernel ће се састојати од секција у следећем redoslijedu: `.text`, `.rodata`, `.stab`, `.stabstr`, `.data` и `.bss`. Садржав сваке секције у kernelu, настаје од садржаја секција из улазних објектних фајлова. На пример, садржав комплетне секције `.text` у kernel фајлу дефиниран је изразом:

```
*(.text .stub .text.* .gnu.linkonce.t.*)
```

Gornja линија interpretira се на следећи начин. Симбол `*` испред заграде означава било који објектни фајл који је улаз у linkeru. Унутар заграде именују се секције из датог фајла чији садржаји преузимају. Током креiranja секције `.text` излазног фајла, из сваког улазног објектног фајла преузимају се садржаји секција: `.text`, `.stub` и свих секција чија имена почињу са `.text.` или `.gnu.linkonce.t..`. Након овог, садржаји преузетих секција из сваког pojedinačnog објектног фајла, надовезат ће се један на други у складу са redoslijedom преузimanja, а потом injektirati у једном комаду у секцију `.text` излазног фајла. Redoslijed процесирања pojedinačnih објектних фајлова путем оператора `*`, усклађен је са redoslijedom улазних фајлова који је дат приликом pozivanja linkera.

Simbol . u linker skripti je brojač koji predstavlja memorisku adresu na koju će se dodavati budući sadržaj. Ukoliko se ne postavi na neku vrijednost, počinje od nule. Pri svakom dodavanju sadržaja iz određenog objektnog fajla, brojač se inkrementira za vrijednost koja predstavlja količinu bajta koja je dodata u izlazni fajl. Nakon dodavanja svake sekcije u izlazni fajl, putem linker direktive PROVIDE, definira se vrijednost za simbole koji predstavljaju lokaciju kraja i početka date sekcije u memoriji. Simboli se koriste se u pojedinačnim ulaznim objektnim fajlovima, a njihove vrijednosti u izlaznom fajlu definira linker tokom uvezivanja.

Pri uvezivanju, linker modifcira sve instrukcije i simbole preuzete iz ulaznih objektnih fajlova, spram njihove nove lokacije u izlaznom fajlu koja je određena sa aktuelnom vrijednošću brojača. Za XV6 kernel, prije dodavanja bilo kakvog sadržaja u izlazni fajl, inicijalna vrijednost brojača postavlja se na vrijednost 0x80100000 u liniji ②. Ovo znači da je adresa uvezivanja kernela postavljena visoko u memoriji na lokaciji 2GB+1MB. Kernel će očekivati da se njegove instrukcije izvršavaju sa ove memoriske adrese. Istovremeno, u liniji ③, putem linker direktive AT, određuje se adresa za učitavanje trenutnog segmenta izlaznog ELF fajla. Adresa za učitavanja XV6 kernela postavljena je nisko u memoriji na lokaciji 1MB (0x100000). Program bootblock koristit će ovu informaciju za učitavanje kernela u memoriju.

Kada je XV6 sistem u potpunosti pokrenut, proizvoljan broj aplikacija izvršavat će se istovremeno. Svaka aplikacija će imati svoj virtualni adresni prostor od 4GB. Sa lokacije u fizičkoj memoriji na kojoj je učitan, kernel će putem straničenja biti mapiran u adresni prostor svake pojedinačne aplikacije iznad adresu KERNBASE (2GB). Ovakva konfiguracija korisna je sa stanovišta ostvarivanje protekcijske pojedinačnih aplikacija uz istovremeno zadržavanje visokog stepena performansi u procesiranju prekida koji nastaju tokom izvršavanja pojedinačnih aplikacija. Ovo je primarni razlog zbog čega se kernel u toku uvezivanja priprema za izvršenje sa lokacije iznad

OUTPUT\_ARCH, lin-  
atu za IA32 plat-  
simbola\_start bu-  
ti definiran u jed-

prisutne u izlaz-  
formatu:

tedoslijedu: .text,  
svake sekcije u ker-  
nel fajlova. Na pri-  
definiran je iz-

Symbol \* ispred za-  
dat linkeru.  
se sadržaji pre-  
fajla, iz svakog  
sekcija: .text, .stub  
linkonce.t.. Na-  
pojedinačnog objek-  
ta redoslijedom  
u sekciju .text  
ulaznih objektnih fajlo-  
redom ulaznih fajlova

2GB. Sa druge strane, adresa na koju se kernel učitava razlikuje se od adrese na kojoj se kernel izvršava. Računari na kojim se instalira operativni sistem potencijalno neće imati više od 2GB fizičke memorije, ali će zasigurno imati memoriske resurse neophodne za učitavanje kernela na lokaciju od 1MB u fizičkoj memoriji.

Kada u potpunosti učita kernel u memoriju, bootblock program izvršava skok na lokaciju simbola `_start` u učitanom programu. Ovim je kontrola predana kernelu, i to u trenutku kada još uvijek nije uključeno straničenje. Konfiguracija i aktivacija straničenja kojom se formira inicijalni virtualni adresni prostor kernela primarni je zadatak koda na lokaciji određenoj simbolom `_start`. Slijedeći listing preuzet je iz fajla `entry.S` u kojem se nalazi definicija simbola `_start`.

```
#include "asm.h"
#include "memlayout.h"
#include "mmu.h"
#include "param.h"

.p2align 2
.text

.globl multiboot_header
multiboot_header:
#define magic 0x1badb002
#define flags 0
.long magic ①
.long flags
.long (-magic-flags)

.globl _start
_start = V2P_W0(entry) ②

.globl entry
entry:
    movl %cr4, %eax
    orl $(CR4_PSE), %eax
    movl %eax, %cr4 ③

    movl $(V2P_W0(entrypgdir)), %eax
    movl %eax, %cr3 ④

    movl %cr0, %eax
    orl $(CR0_PG|CR0_WP), %eax
    movl %eax, %cr0 ⑤
```

kernel učitava razlikuje se učitati na kojim se instaliraju više od 2GB fizičke memorije neophodne za vlastoj memoriji.

Na početku, bootblock program je učitanom programu. Na početku kada još uvijek aktivacija straničenja kopija prostor kernela primarni adresi \_start. Slijedeći list će definicija simbola

```
    movl $(stack + KSTACKSIZE), %esp 6
    mov $main, %eax
    jmp *%eax 7
.comm stack, KSTACKSIZE 8
```

XV6 kernel je pripremljen tako da se njegovo učitavanje može obaviti putem bootblock programa, ali i putem ostalih specijaliziranih programa za startanje operativnih sistema. Ovi programi očekuju da učitavani kernel uključuje multiboot zaglavljkoje je u specijaliziranom formatu. XV6 kernel od lokacije **1** definira ovo zaglavljkoje je veličine 12B.

Prva instrukcija kernela počinje od simbola entry. Cilj je da bootblock, kada predaje kontrolu kernelu, izvrši skok na ovu instrukciju. Ukoliko bi simbol \_start bio postavljen direktno na vrijednost simbola entry, obzirom na link adresu kernela, skok bi se izvršio na lokaciju u fizičkoj memoriji iznad 2GB na kojoj nije učitan kernel. Zbog toga se \_start simbol inicijalizira putem makro, koji je definiran u fajlu memlayout.h na slijedeći način:

```
#define KERNBASE 0x80000000
//...
#define V2P_W0(x) ((x) - KERNBASE)
#define P2V_W0(x) ((x) + KERNBASE)
```

Makro V2P\_W0 radi konverziju virtualne adrese u fizičku adresu oduzimanjem vrijednosti KERNBASE, dok P2V\_W0 radi obrnutu operaciju.

Nakon obavljenog skoka u kernel, počinje se sa konfiguracijom straničenja. U liniji **3** aktivira se podrška za straničenje sa stranicama veličine 4MB, i to postavljanjem bita CR4\_PSE registra cr4. Zatim se registar cr3 puni adresom na kojoj se nalazi PD tabela prvog nivoa straničenja. Tabela je određena lokacijom simbola entrypgdir, globalnom varijablu definiranom u fajlu main.c putem slijedećeg listinga:

```

#define NPDENTRIES      1024
//...
#define PDXSHIFT        22
//...
typedef uint pde_t;
//...
pde_t entrypgdir[NPDENTRIES] = {
    [0] = (0) | PTE_P | PTE_W | PTE_PS,
    [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};

```

Varijabla `entrypgdir` je niz od 1024 elementa. Svi elementi su 32-bitni cijeli brojevi postavljeni na vrijednost 0, osim elemenata sa indeksima: 0 i 512 (`KERNBASE>>PDXSHIFT`) čija je vrijednost zadata u binarnom formatu kao:

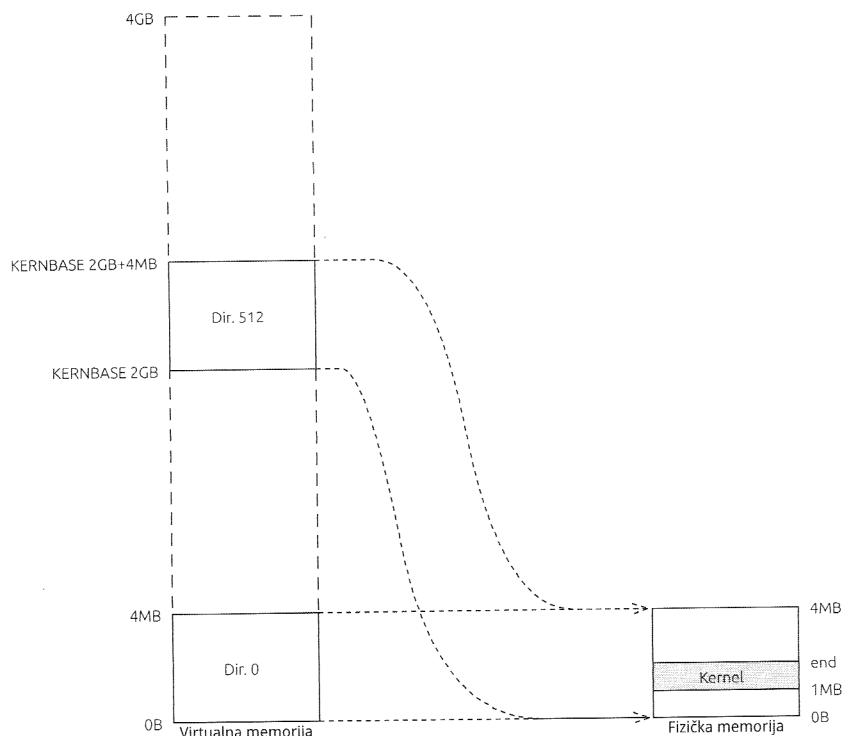
`000000000000000000000000_000010000011`

Gornja vrijednost, definira mapiranje u nulti okvir fizičke memorije veličine 4MB (`PTE_P | PTE_PS`), sa postavljenom dozvolom za pisanje (`PTE_W`). Dakle, direktoriji sa rednim brojevima 0 i 512, putem PD tabele `entrypgdir`, mapiraju se u prvu stranicu fizičke memorije veličine 4MB. Svi ostali direktoriji nemaju definirano mapiranje. Po uključivanju straničenja, pristup adresama koje pripadaju tim direktorijima proizveo bi iznimku.

Kao što je prikazano na slici 5-1, kada se u liniji ④ aktivira straničenje, fizičkim adresama od 0 do 4MB moguće je pristupiti sa dvije lokacije u definiranoj virtualnoj memoriji,

U liniji ③ putem asembler direktive `.comm`, alociran je prostor veličine `KSTACKSIZE` (4096). Sve dok se ne inicijalizira kernel alokator stranica, alocirani prostor koristit će se kao trenutni stek. Prelazak na ovaj stek izvodi se u liniji ⑥, postavljanjem registra `esp` na vrh alociranog prostora.

Konačno, u liniji ⑦ vrši se skok na lokaciju asociranu sa simbolom `main`. Ovim je okončana prva faza pokretanja operativnog sistema. Druga faza pokretanja kompletno se odvija unutar C funkcije `main` koja je definirana u fajlu `main.c`.



Slika 5-1. Inicijalni virtualni adresni prostor kernela

PS,

Svi elementi su  
osim elemenata sa  
jednost zadata u

okvir fizičke me-  
mljenom dozvolom  
brojevima 0 i 512,  
stranicu fizičke  
nemaju definirano  
adresama koje pri-

aktivira strani-  
je pristupiti sa dvije

ociran je prostor ve-  
zira kernel alokator  
enutni stek. Prelazak  
registra esp na vrh

asocirano sa simbo-  
etanja operativnog sis-  
cija unutar C funkcije

avije 5, Učitavanje kernela

Inicijalna konfiguracija straničenja u kernelu

## POGLAVLJE 6

# Inicijalizacija kernela

Druga faza pokretanja XV6 operativnog sistema kompletna se odvija u kernelu. U okviru ove faze kernel treba da:

1. Pripremi straničenje sa stranicama od 4KB.
2. Inicijalizira i konfigurira hardverske uređaje.
3. Podesi IDT tabelu za tretman prekida.
4. Pripremi za izvršavanje prvi proces.
5. Pokrene sva raspoloživa jezgra.

Nakon što se obave navedeni koraci sva detektovana procesorska jezgra angažuju se na izvršavanju korisničkih aplikacija.

## Funkcija main

Funkcija `main` ima ključnu ulogu u implementaciji druge faze pokretanja XV6 operativnog sistema. U trenutku poziva ove funkcije, kernel se izvršava sa inicijalnim virtualnom adresnim prostorom zasnovanom na dvije stranice od 4MB. BSP jezgro procesora izvodi kod ove funkcije, dok su AP jezgra u zaustavljenom stanju.

Slijedi kompletan kod ove funkcije preuzet iz fajla `main.c`.

```
int
main(void)
{
    kinit1(end, P2V(4*1024*1024));
    kvmalloc(); ①
    mpinit();
    lapicinit(); ②
    seginit(); ③
```

```

    cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
    picinit(); ④
    ioapicinit(); ②
    consoleinit(); ⑤
    uartinit(); ⑤
    pinit();
    tvinit(); ⑥
    binit(); ⑦
    fileinit(); ⑦
    iinit(); ⑦
    ideinit(); ⑦
    if(!ismp)
        timerinit(); ④
    startothers(); ⑧
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
    userinit(); ⑨
    mpmain();
}

```

Pozivom `kinit1` na samom početku funkcije inicijalizira se alokator stranica radi korištenja preostalog prostora od zadnjeg bajta na kojem je učitan kernel (end), do kraja prvog okvira u fizičkoj memoriji (4MB). Rasploživi memoriski prostor za alokator, pozivom `kinit2` pred kraj funkcije, proširit će se do fizičke memoriske lokacije određene sa `PHYSTOP` (224MB).

U liniji **①** pozivom `kvmalloc`, prvo se konfigurira, a zatim počinje i koristiti, novi sistem straničenja baziran na stranicama veličine 4KB.

Zatim se putem funkcije `mpinit` pretražuje BIOS memorija radi pronalaženja MP tabele. Ukoliko se tabela pronađe u validnoj konfiguraciji, sistem na kojem se kernel izvodi je višeprocesorski. Za ovaj slučaj koristit će se LAPIC i IOAPIC kontroleri za generiranje prekida. U linijama **②**, konfiguiraju se postavke za ove kontrolere. LAPIC kontroler svakog jezgra konfigurira se tako da putem tajmera integriranog na jezgru proizvodi prekide u jednakim vremenskim intervalima. U slučaju da je detektovan jednoprocesorski sistem, PIC kontroler se koristi za generiranje prekida. Za ovaj slučaj, prekidi u jednakim vremenskim intervalima za procesor dolazit će od PIT tajmera povezanog na liniju `irq0`. Inicijalizacija PIC kontrolera i PIT tajmera obavlja se u linijama **④**.

U liniji ③ prelazi se na novi GDT koji za BSP procesor aktivira segmentiranje sa 6 deskriptora i to: dva sa dozvolama za kernel, dva sa dozvolama za korisničke programe, jedan za TSS i poseban segment za dvije CPU varijable.

U linijama ⑤ inicijalziraju se vanjski uređaji spram IOAPIC ili PIC kontrolera prekida.

IDT tabela kojom XV6 asocira svoje ISR rutine sa pojedinačnim vektorima prekida konfigurira se u liniji ⑥.

XV6 fajl sistem se inicijalizira pozivima u linijama ⑦, a AP jezgra pokreću u liniji ⑧.

Nakon proširivanja raspoloživog prostora za alokator stranica, vrši se inicijalizacija prvog procesa u liniji ⑨. Nakon ovog, obzirom da su obavljeni svi neophodni zadaci pokretanja operativnog sistema, BSP jezgro može se pridružiti AP jezgrama u izvršavanju korisničkih aplikacija, pozivom funkcije `mpmain`.

## Alokator stranica

Nakon učitavanja i preuzimanja kontrole, kernel ima potrebu da koristi memoriju koja je na raspolaganju, kako za svoje interne potrebe, tako i za potrebe aplikacija koje se izvršavaju pod kontrolom operativnog sistema.

Slobodna fizička memorija koja ostaje iznad prostora na koju je kernel učitan dijeli se na komade jednakih veličina od po 4KB koji odgovaraju okvirima za straničenje. Radi praćenja stanja zauzetosti pojedinačnih okvira u kernelu se formira jednostruko povezana lista slobodnih fizičkih okvira, koja se naziva alokator stranica. Alokator ima zadatak da, po potrebi, za ostale komponente kernela obezbijedi slobodne fizičke okvire. Isto tako, kada određena komponenta u kernelu prestane sa radom, alokator ima zadatak da reciklira fizičke okvire koje je koristila data komponenta.

U  
① su,  
koj  
② Ko  
③ Ne  
④ Na  
slip  
⑤ No  
alo

Obrn  
tjeva m  
se puten  
adresu s

char\* k  
{  
struct  
if(k  
acq  
r = k  
if(r)  
kne  
if(k  
rel  
retu  
}

① N  
② U  
③ D  
④ V  
al

Za implementaciju alokatora okvira, XV6 koristi globalnu strukturu `kmem`, koja u fajlu `kalloc.c` ima slijedeću definiciju:

```
struct run {  
    struct run *next;  
};  
  
struct {  
    struct spinlock lock;  
    int use_lock;  
    struct run *freelist;  
} kmem;
```

Polja `lock` i `use_lock` koriste se za sinhronizaciju pristupa alokatoru kada više jezgri procesora izvršava kernel kod. Ključno polje u strukturi je polje `freelist` koje predstavlja lokaciju u memoriji na kojoj se nalazi prva slobodna stranica u virtualnoj memoriji koja je data na upravljanje alokatoru. Inicialno, ovo polje ima vrijednost 0, tj. alokator na početku svog funkcionisanja nema na raspolaganju niti jednu stranicu.

Slobodna stranica se daje na upravljanje alokatoru putem funkcije `kfree` koja kao parametar uzima pointer na početak stranice koja se predaje na upravljanje alokatoru.

```
void kfree(char *v)  
{  
    struct run *r;  
  
    if((uint)v % PGSIZE || v < end || v2p(v) >= PHYSTOP)  
        panic("kfree"); ①  
  
    memset(v, 1, PGSIZE); ②  
  
    if(kmem.use_lock)  
        acquire(&kmem.lock); ③  
    r = (struct run*)v; ④  
    r->next = kmem.freelist; ④  
    kmem.freelist = r; ⑤  
    if(kmem.use_lock)  
        release(&kmem.lock); ③  
}
```

koristi globalnu strukturu definiciju:

zajednicu pristupa alokatoru kod. Ključno polje u lokaciju u memoriji na fizičkoj memoriji koja je polje ima vrijednost 0, te nema na raspaganju

alokatoru putem funkcije početak stranice koja

= STOP)

- U slučaju da proslijedena adresa nije poravnata na 4KB adresu, ili ako data virtualna adresa ne korespondira validnoj fizičkoj adresi, zaustavlja se dalje izvođenje koda.
- ❶ Kompletna stranica popunjava se sa vrijednosti 1.
  - ❷ Neophodno radi sinhronizacije.
  - ❸ Na samom početku stranice od 4KB postavlja se pointer na sljedeću slobodnu stranicu u alokatoru.
  - ❹ Nova stranica stavlja se na početak liste slobodnih stranica u alokatoru.

Obrnuta operacija kojom se nekoj komponenti kernela koja zahitjeva memoriju daje na korištenje jedna slobodna stranica, obavlja se putem funkcije kalloc. Funkcija ne uzima parametre, a vraća adresu stranice koja je data na korištenje.

```
char* kalloc(void)
{
    struct run *r;

    if(kmem.use_lock)
        acquire(&kmem.lock); ❶
    r = kmem.freelist; ❷
    if(r)
        kmem.freelist = r->next; ❸
    if(kmem.use_lock)
        release(&kmem.lock); ❹
    return (char*)r;
}
```

- ❶ Neophodno radi sinhronizacije.
- ❷ Uzima se pointer na prvu slobodnu stranicu u alokatoru.
- ❸ Druga slobodna stranica u alokatoru postaje prva.
- ❹ Vraća se adresa prve stranice koja više nije pod upravljanjem alokatora.

Funkcije `kfree` i `kalloc` koriste se na različitim lokacijama u kodu, uključujući i funkciju `main` prilikom pokretanja operativnog sistema. Na samom početku funkcije `main`, alokatoru se daju na upravljanje okviri u fizičkoj memoriji iznad lokacije na koju je učitan kernel (`v2p(end)`), a do adrese 4MB. Ovo se ostvaruje putem slijedećeg poziva funkcije `kinit1`.

```
kinit1(end, P2V(4*1024*1024));
```

Funkcija `kinit1` i sa njom asocirana funkcija `freerange` definirane su sa slijedećim kodom:

```
void kinit1(void *vstart, void *vend)
{
    initlock(&kmem.lock, "kmem"); ①
    kmem.use_lock = 0; ②
    freerange(vstart, vend);
}

// ...

void freerange(void *vstart, void *vend)
{
    char *p;
    p = (char*)PGROUNDUP((uint)vstart); ③
    for(; p + PGSIZE <= (char*)vend; p += PGSIZE) ④
        kfree(p);
}
```

- ① Inicijalizira se brava koja se koristi za sinhronizaciju pristupa alokatoru.
- ② Isključuje se upotreba sinhronizacije.
- ③ Prva adresa iz datog raspona se zaokružuje na prvu gornju adresu djeljivu sa 4096.
- ④ Za sve adrese u zadanom rasponu, sa korakom od 4096, poziva se `kfree`.

Efekat poziva funkcije `kinit1` prikazan je na slici 6-1.

KERNBASE

Slika 6-1.

Ukolik  
ka kompo  
jio jednu  
je alokator  
vidljivo o  
stranice u  
kon pozic  
zatražila

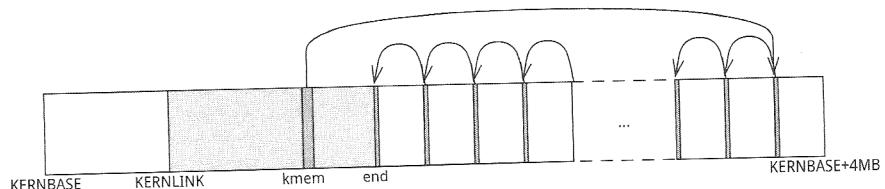
KERNBASE

Slika 6-2.

Nakon  
alokator  
jedećeg p

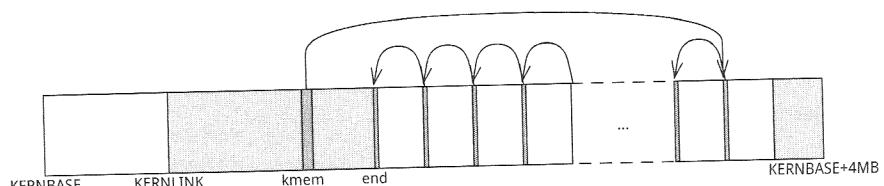
```
kinit1(
// ...
void kfree(
{
    free(
    knem.
})
```

Alokator st



Slika 6-1. Slobodne stranice kojim upravlja alokator nakon poziva `kinit1`

Ukoliko zahtijeva stranicu slobodne memorije na korištenje, neka komponenta kernela bi pozvala `kalloc`. Ovim bi alokator izdvojio jednu slobodnu stranicu. Rezultat ove operacije, i to nakon što je alokator inicijaliziran sa `kinit1`, prikazan je na slici 6-2. Sa slike je vidljivo da `kmem.freelist` uvijek pokazuje na adresu prve slobodne stranice u alokatoru, a stranica koja je bila prva na alokator listi, nakon poziva `kalloc`, počinje se koristiti u komponenti kernela koja je zatražila alokaciju.



Slika 6-2. Slobodne stranice kojim upravlja alokator nakon poziva `kalloc`

Nakon što se unutar funkcije `main` izvrši pokretanje AP jezgri, alokatoru se dodaju ostale slobodne stranice iznad 4MB putem slijedećeg poziva funkcije `kinit2`:

```

kinit2(P2V(4*1024*1024), P2V(PHYSTOP));

// ...

void kinit2(void *vstart, void *vend)
{
    freerange(vstart, vend); ①
    kmem.use_lock = 1; ②
}

```

Alokator stranica

Unutar funkcije ponovo se poziva `freerange` sa datim rasponom adresa u liniji ①, a zatim uključuje sinhronizacija u liniji ②, obzirom da su tada aktivirane sve procesorske jezgre.

## Kernel konfiguracija za straničenje

Inicijalno straničenje uspostavljeno u prvoj fazi pokretanja operativnog sistema dovoljno je za učitavanje kernela i konfiguriranje alokatora stranica. Postavka straničenja od dvije stranice veličine 4MB koje se mapiraju u isti okvir fizičke memorije, uspostavlja se putem samo jedne tabele prvog nivoa. Tabela je definirana kao globalni niz od 1024 elementa, pod imenom `entrypgdir`. Svi elementi niza u potpunosti su određeni u procesu kompajliranja.

Ovakva konfiguracija straničenja, iako jednostavna za kreiranje, neadekvatna je spram racionalne upotrebe više od 4MB fizičke memorije. Zbog toga, nakon što se inicijalizira alokator stranica, pozivom `kvmalloc`, uspostavlja se novo straničenje za BSP jezgru koje je bazirano na stranicama veličine 4KB. Slijedi kod funkcije `kvmalloc` koja je implementirana u fajlu `vm.c`.

```
void kvmalloc(void)
{
    kpgdir = setupkvm(); ①
    switchkvm();
}
```

Postavka za novo straničenje, koja uključuje jednu PD tabelu i sve neophodne PT tabele, sprovodi se unutar `setupkvm`. Rezultat ove funkcije je adresa konfiguiriranog PDa, koja se u liniji ① koristi za inicijalizaciju globalne varijable `kpgdir`. Aktivacija formiranog straničenja za BSP jezgro događa se u sljedećoj liniji koda, pozivom funkcije `switchkvm`. Ova funkcija u registar `cr3` učitava fizičku adresu okvira sa kojom je asocirana globalna varijabla `kpgdir`, čime se aktivira novo straničenje. Ovu funkciju, neposredno nakon buđenja, pozivaju i AP jezgre da bi imale identičan virtualni adresni prostor kao i BSP jezgra.

datim rasponom u liniji ②, obzirom

pokretanja oper-kernela i konfiguriranje stranice veličine memorije, uspostavlja se definirana kao globalna pgdir. Svi elementi su definirani.

predstavna za kreiranje, od 4MB fizičke memorije, kator stranica, pozicija na BSP jezgru koje je funkcije kvmalloc

jednu PD tabelu i sve se setupkvm. Rezultat ove se u liniji ① koristi za formiranog stranicijski liniji koda, pozivom ③ učitava fizičku adresu tabla kpgdir, čime se posredno nakon buđenja virtualni adresni

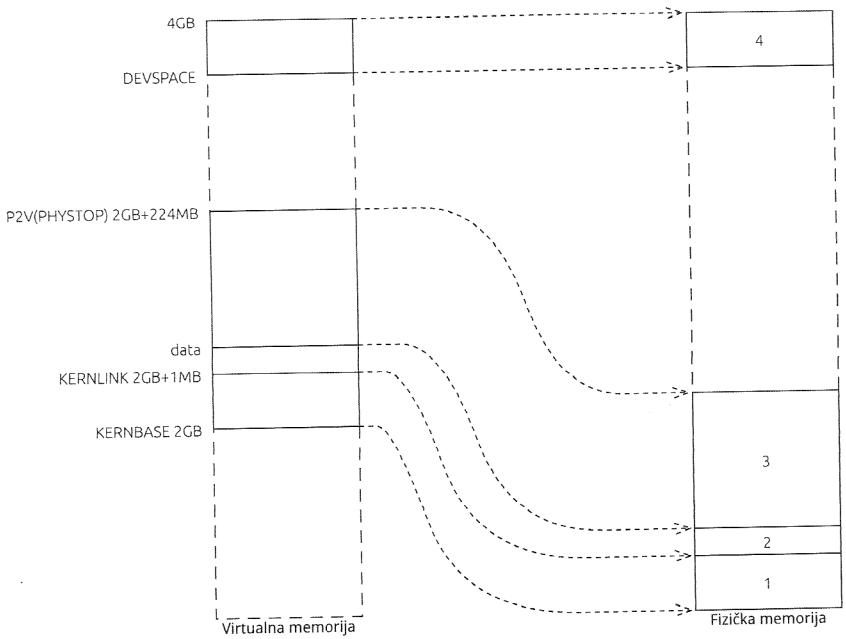
Prilikom formiranja novog adresnog prostora, setupkvm se oslanja na konfiguraciju virtualnog adresnog prostora formiranu putem globalnog niza kmap. Ova varijabla definirana je u fajlu vm.c na slijedeći način:

```
static struct kmap {  
    void *virt;  
    uint phys_start;  
    uint phys_end;  
    int perm;  
} kmap[] = {  
    { (void*)KERNBASE, 0,           EXTMEM,      PTE_W}, ①  
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, ②  
    { (void*)data,   V2P(data),     PHYSTOP,     PTE_W}, ③  
    { (void*)DEVSPACE, DEVSPACE,    0,           PTE_W}, ④  
};
```

Svaki element niza kmap predstavlja kontinuirani blok u fizičkoj memoriji koji se mapira na određenu lokaciju u virtualnom adresnom prostoru kernela. Početak i kraj bloka fizičke memorije koji se mapira, definiraju polja phys\_start i phys\_end. Polje virt predstavlja adresu u virtualnoj memoriji na koju se dati fizički blok mapira, a polje perm definira dozvole koje se apliciraju za mapiranje. Putem kmap određuje se kompletan virtualni adresni prostor kernela od četiri kontinuirana bloka, čiji je izgled prikazan na slici 6-3.

Blokovi ① i ③ predstavljaju slobodnu memoriju ispod i iznad adresa na kojim je učitan kernel. Dio bloka ③ koristi se za .bss i .data sekcije iz kernel ELF fajla. Blok memorije ②, u koji su iz kernel ELF datoteke učitane sekcije .text, .rodata, .stab i .stabstr, označen je bez dozvola za pisanje, obzirom da se u ovom segmentu fizičke memorije nalaze kernel kod i simboli za debagiranje. Postavljenjem dozvole PTE\_W u ostalim definiranim blokovima kernel virtualna memorije dozvoljava se pisanje. Pristup fizičkim adresama na koje se mapiraju kontroleri vanjskih uređaja omogućen je u kernel virtualnoj memoriji putem bloka ④.

Da bi se mapiranje kernel virtualne memorije sprovelo u djelo, fizički blokovi definirani u varijabli kmap moraju se prvo izdijeliti na



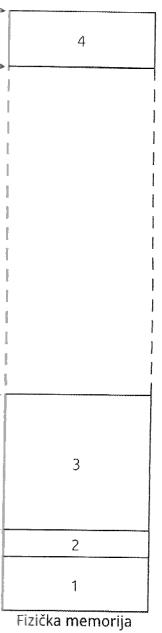
Slika 6-3. Kernel virtualni adresni prostor

okvire veličine 4KB. Svaki pojedinačni okvir zatim se mapira u odgovarajuću stranicu u virtualnom adresnom prostoru. Ova mapiranja bilježe se u PT tabelama, i to po jedna tabela za svaki direktorij koji je uključen u mapiranje. Svaki definirani PT zahtijevaće jedan okvir fizičke memorije koji se alocira putem alokatora stranica pozivom funkcije `kalloc`. Dodatno, za tabelu prvog nivoa, u koju se smještaju informacije o adresama fizičkih okvira tabela drugog nivoa, biće također potreban jedan fizički okvir.

Slijedeći listing prikazuje opisani proces koji se implementira unutar funkcije `setupkvm`.

```
pde_t* setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;

    if((pgdir = (pde_t*)kalloc()) == 0) ①
        return 0;
    memset(pgdir, 0, PGSIZE); ②
    if (p2v(PHYSTOP) > (void*)DEVSPEC)
```



```

        panic("PHYSTOP too high");
    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
        if(mappages(pgdир, k->virt, k->phys_end - k->phys_start,
                    (uint)k->phys_start, k->perm) < 0) ❸
            return 0;
    return pgdир;
}

```

- ❶ Alokacija okvira za PD.
- ❷ Inicijalizacija svih elemenata u PDu na 0.
- ❸ Za svaki blok u kmap varijabli, putem funkcije mappages, vrši se mapiranje pojedinačnih stranica iz bloka.

Kompletan posao oko mapiranja pojedinačnih stranica koje pripadaju jednom kontinuiranom bloku memorije održuje se unutar funkcije mappages. Ova funkcija, po potrebi, vrši alokaciju neophodnih PTova, kao i odgovarajuće modifikacije PDa spram alociranih PTova.

Kada se pozivom switch kvm aktivira mapiranje definirano u funkciji setup kvm, bilo kakav pristup stranicama u virtualnoj memoriji koje nemaju mapiranje u fizičkoj memoriji rezultira iznimkom.

## Kernel konfiguracija za segmentiranje

Inicijalna konfiguracija segmentiranja uspostavljena u prvoj fazi pokretanja operativnog sistema postavlja dva deskriptora u GDTu. Segmenti definirani putem ovih deskriptora mogu se koristiti samo dok je procesor u nultom nivou privilegija. Nakon obavljenih koraka neophodnih za konfiguraciju kernela, konačni cilj pokretanja operativnog sistema je da se sva procesorska jezgra zaposle na izvršavanju koda pokrenutih aplikacija. Kada izvršava kod od neke aplikacije, jezgro treba da bude u stanju najnižeg nivoa privilegija. U tom trenutku svi segmentni selektori moraju biti asocirani sa odgovarajućim deskriptorima čije je DPL polje postavljeno na vrijednost

se mapira u odnosu. Ova mapiranja se rađaju za svaki direktorij PT zahtijevaće jedan direktor stranica po nivoa, u koju se takođe tabeli drugog nivoa.

se implementira

3. Obzirom da ne sadrži niti jedan deskriptor koji zadovoljava ovaj kriterij, inicijalni GDT nije pogodan za izvršavanje običnih aplikacija.

Nakon što uključi konačno kernel straničenje, pozivom funkcije `seginit`, BSP jezgro uspostavlja novi GDT. Slijedeći listing preuzet je iz fajla `vm.c` u kojem je implementirana funkcija `seginit`.

```
void seginit(void)
{
    struct cpu *c;

    c = &cpus[cpunum()];
    ❶
    c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0); ❷
    c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0); ❷
    c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER); ❸
    c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER); ❸
    c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0); ❹

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

    cpu = c; ❻
    proc = 0; ❻
}
```

Gornja funkcija koristi globalnu varijablu `cpus` u kojoj XV6 za svaku pojedinačnu procesorsku jezgru drži informacije ključne za segmentiranje. Slijedeća dva listinga, preuzeta iz fajlova `mp.c` i `proc.h`, relevantni su za definiciju niza `cpus`.

```
struct cpu cpus[NCPUs];

struct cpu {
    uchar id;
    struct context *scheduler;
    struct taskstate ts;
    struct segdesc gdt[NSEGS];
    volatile uint started;
    int ncli;
    int intena;

    struct cpu *cpu;
    struct proc *proc;
};


```

Svako de...  
kro NCPU pr...  
ujedno i ma...  
rira. Elemen...  
strukture im...

**id**

Brojčani

**scheduler**

Posebna

**ts**

Komplet

**gdt**

Komplet

**started**

Indikator

**ncli**

Varijabla

**intena**

Varijabla

**cpu**

Pointer k...  
koji je as...  
lje id.

**proc**

Pointer k...  
procesa k...  
identifik

Kernel konfig

koji zadovoljava ovaj  
izravanje običnih aplikacija.  
pozivom funkcije  
Sljedeći listing preuzet  
sa seginit.

```
    ffff, 0); ②
    0); ②
    ffff, DPL_USER); ③
    DPL_USER); ③
```

④

cpus u kojoj XV6 za  
informacije ključne za  
iz fajlova mp.c i

Svako detektovano jezgro će imati svoj element u nizu cpus. Makro NCPU prestavlja maksimalni broj elemenata u ovom nizu, a to je ujedno i maksimalan broj jezgri sa kojim XV6 kernel može da operira. Elementi niza cpus su strukture tipa cpu. Pojedinačna polja ove strukture imaju slijedeća značenja:

#### **id**

Brojčani identifikator za jezgro dobiven iz MP tabele.

#### **scheduler**

Posebna struktura koja se koristi tokom promjene konteksta.

#### **ts**

Kompletna TSS struktura.

#### **gdt**

Kompletan GDT sa ukupno NSEGS (7) deskriptora.

#### **started**

Indikator da li je jezgro u potpunosti pokrenuto.

#### **ncli**

Varijabla koja se koristi u procesu sinhronizacije.

#### **intena**

Varijabla koja se koristi u procesu sinhronizacije.

#### **cpu**

Pointer koji uvijek treba da pokazuje na onaj element u nizu cpus koji je asociran sa procesorskom jezgrom čiji je identifikator polje id.

#### **proc**

Pointer koji uvijek treba da pokazuje na strukturu proc od onog procesa koji u datom trenutku izvršava procesorsko jezgro čiji je identifikator polje id.

Od svih polja strukture `cpu`, funkcija `seginit` manipulira samo poljima `gdt`, `proc` i `cpu`. U liniji ① ove funkcije, iz niza `cpus` prvo se izdvaja element spram identifikatora jezgra koje trenutno izvodi kod ove funkcije. Identifikator se dobija pozivom funkcije `cpunum`.

Potom se za datu jezgru konfigurira GDT tabela, koja je kompletan sadržana u polju `gdt` elementa odabranog iz niza `cpus`. Deskriptori u GDTu koji će se koristiti dok se izvršava kernel kod, inicijaliziraju se u linijama ②. Pri čemu će se deskriptor sa indeksom `SEG_KCODE` koristiti za CS selektor, a `SEG_KDATA` za sve ostale selektore. Dva deskriptora koji će se koristiti dok procesor izvršava kod od običnih aplikacija, konfiguriraju se u linijama ③. Konfiguracije su slične onima za kernele deskriptore, sa jednom ključnom razlikom, a to je da se dozvole za upotrebu deskriptora postavljaju na najnižu razinu (`DPL_USER → 3`).

Poseban deskriptor unutar GDTa, pod rednim brojem `SEG_KCPU`, kreira se u liniji ④. Polje `base` ovog deskriptora nije postavljeno na 0, već na adresu polja `cpu` elementa iz niza `cpus` asociranog sa jezgrom koja trenutno izvršava `seginit`. Polje `limit` ovog deskriptora definira segment veličine 8 bajta. Na nultoj adresi unutar definiranog segmenta pristupa se polju `cpu` od elementa iz niza `cpus`, dok se na adresi 4 pristupa polju `proc` od istog elementa. Deskriptor pod rednim brojem `SEG_KCPU` keširat će se u selektoru `gs`, koji se koristi pri definiciji dvije globalne kernel varijable: `cpu` i `proc`, i to na slijedeći način:

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

Varijable `proc` i `cpu` neće imati memoriju lokaciju u `.data` sekciji kernela, kao što je slučaj sa klasičnim globalnim varijablama. Putem `asm` direktiva pozicija ovih varijabli vezana je za segment koji je u datom trenutku asociran sa selektorom `gs`. Obzirom na definiciju segmenta sa indeksom `SEG_KCPU` u GDTu, koji je asociran sa `gs` selektorom u svakom jezgru, zaključujemo da se na ovaj način jed-

je manipulira samo iz niza cpus prvo je trenutno izvodi funkcije cpunum.

GDT tabela, koja je komponig iz niza cpus. Desavala kernel kod, inicijator sa indeksom za sve ostale selektore procesor izvršava kod od ③. Konfiguracije su uključnom razlikom, postavljaju na najnižu

brojem SEG\_KCPU, nije postavljeno na asociranog sa jezgrom ovog deskriptora unutar definiranog niza cpus, dok se na Deskriptor pod redoslijedom, koji se koristi pri izvršec, i to na slijedeći

lokaciju u .data sekciju varijablama. Puna je za segment koji je Obzirom na definiciju je asociran sa gsegmentom se na ovaj način jed-

na globalna varijabla proc mapira u više polja proc u odgovarajućim elementima niza cpus, i to u ovisnosti od jezgra koje pristupa varijabli u datom trenutku. Isto vrijedi i za globalnu varijablu cpu koja se mapira u polja cpu. Dakle, svako jezgro preko globalne variabile cpu ima direktni pristup njemu dodjeljenom elementu unutar niza cpus, dok preko globalne variabile proc ima pristup proc strukturi koja je asocirana sa procesom koje dato jezgro izvršava u tom trenutku.

Učitavanjem adrese GDTa u gdtr registar aktivne procesorske jezgre aktivira se segmentiranje u liniji ⑤, dok se u liniji ⑥ vrši kesiiranje cs deskriptora iz nove GDT tabele. Pred sam kraj funkcije, globalna varijabla cpu inicijalizira se u liniji ⑦, dok se globalna varijabla proc, obzirom da jezgro još uvijek nije dobilo proces kojeg treba da izvršava, postavlja na 0.

## Pokretanje AP jezgri

Proces pokretanje operativnog sistema, izvršavanjem koda funkcije main, sprovodi striktno BSP jezgro. Sastavni zadatak ovog procesa je aktiviranje svih procesorskih jezgri koje su na raspolaganju datom računaru. AP jezgra se pokreću tako da im BSP jezgro pošalje sekvencu među-procesorskih prekida (IPI) putem koje se jezgra bude iz zaustavljenog stanja i počinju izvršavati kod sa lokacije koju im obezbijeđuje BSP jezgro.

Odmah po buđenju, AP jezgra su u sličnom stanju u kojem je bilo BSP jezgro kada se počeo izvršavati program bootblock. Svako AP jezgro treba proći proces inicijalizacije, u kojem jezgro:

1. prelazi u protected mod operacije,
2. uključuje inicijalno straničenje,
3. izvršava funkciju kojom se za dato jezgro podešava konačno segmentiranje i straničenje identično BSP jezgru.

Početna dva koraka inicijalizacije AP jezgra definirani su u programu `entryother`, koji je uvezan skupa sa objektnim fajlovim kernela u zajedničku kernel ELF datoteku. Treći korak inicijalizacije definiran je u kernel funkciji `mpenter`. BSP jezgro obavlja buđenje svih AP jezgri u kodu funkcije `startothers`. Unutar ove funkcije, BSP jezgro treba da pripremi kod koje se proslijedi na izvršenje svakom AP jezgru nakon buđenja. Slijedeći listing preuzet je iz fajla `main.c` u kojem je implementirana ova funkcija.

```
static void startothers(void)
{
    extern uchar _binary_entryother_start[], _binary_entryother_size[];
    uchar *code;
    struct cpu *c;
    char *stack;

    code = p2v(0x7000); ①
    memmove(code, _binary_entryother_start,
            (uint)_binary_entryother_size); ②

    for(c = cpus; c < cpus+ncpu; c++){ ③
        if(c == cpus+cputnum())
            continue; ④

        stack = kalloc(); ⑤
        *(void**)(code-4) = stack + KSTACKSIZE; ⑥
        *(void**)(code-8) = mpenter; ⑥
        *(int**)(code-12) = (void *) v2p(entrypgdir); ⑥

        lapicstartap(c->id, v2p(code)); ⑦

        while(c->started == 0) ⑧
            ;
    }
}
```

- Pointer `code` se postavlja na lokaciju u virtualnoj memoriji
- ① koja korespondira fizičkoj lokaciji 0x7000. Ovo je ujedno i adresa na koju program `entryother` očekuje da bude učitan.
  - Sa lokacije u memoriji na kojoj je učitan u okviru kernela, sa
  - ② držaj fajla `entryother` kopira se tako da u fizičkoj memoriji počinje od lokacije 0x7000.
  - ③ Petlja se izvršava za svaku detektovanu jezgru.

- definirani su u programnim fajlovim kernele. Korak inicijalizacije obavlja buđenje unutar ove funkcije, određuje na izvršenje i preuzet je iz fajla `_binary_entryother_size[];`
- ④ U slučaju da trenutni element iz niza `cpus` korespondira BSP jezgri, preskače se na sljedeći element niza.
  - ⑤ Alocira se jedan fizički okvir za stek koji će koristiti AP jezgro koje se budi u dатој iteraciji petlje.

Postavljaju se argumenti za program `entryother` kojeg će izvršavati AP jezgro kada se probudi. Argumenti su: lokacija steka za jezgro, lokacija funkcije koju jezgro treba da počne izvršavati nakon što pređe u protected mode operacije i lokacija inicijalne tabele za straničenje. Argumenti se smještaju ispod adrese 0x7000, poštujući njihov redoslijed.
  - ⑥ Šalje se IPI procesorskoj jezgri koja se budi, argument za ovaj među-procesorski prekid je fizička lokacija u memoriji sa koje dato jezgro treba da počne izvršavati kod.
  - ⑦ BSP prelazi u mod čekanja unutar beskonačne petlje. BSP može nastaviti dalje, tek kada AP jezgro koje se budi promjeni stanje polja `started` u svojoj strukturi iz niza `cpus`.

Lokacije u kernelu na kojem počinje (`_binary_entryother_start`) i završava (`_binary_entryother_size`) kod koji treba da inicijalno izvršavaju AP jezgra, definira linker u procesu uvezivanja kernel ELF datoteke. Datoteka `entryother`, koja sadrži ovaj kod, uvezana je u kernel ELF fajl u binarnom formatu na sami kraj sadržaja kernel fajla.

Slijedeći segment preuzet iz Makefile skripte, određuje formiranje datoteke `entryother`.

```
entryother: entryother.S
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c entryother.S
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7000 \
-o bootblockother.o entryother.o
$(OBJCOPY) -S -O binary -j .text bootblockother.o entryother
```

Prvo se asemblira datoteka `entryother.S`, čime nastaje ELF fajl `entryother.o`. Ovaj fajl se zatim uvezuje u izvršnu datoteku `boot-`

blockother.o, pri čemu se kod .text sekciјe pozicionira za izvršenje sa adresi 0x7000. Konačna dodatoteka entryother, koja se uvezuje u kernel, nastaje izdvajanjem u binarnom formatu sadržaja sekciјe .text iz ELF fajla bootblockother.o.

Slijedi sadržaj fajla entryother.S u kojem je definirana izdvojena .text sekciјa.

```
.code16
.globl start
start:
    cli
    xorw    %ax,%ax
    movw    %ax,%ds
    movw    %ax,%es
    movw    %ax,%ss
    lgdt    gdtdesc
    movl    %cr0, %eax
    orl    $CR0_PE, %eax
    movl    %eax, %cr0
    ljmpl   $(SEG_KCODE<<3), $(start32) ①

.code32
start32:
    movw    $(SEG_KDATA<<3), %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss
    movw    $0, %ax
    movw    %ax, %fs
    movw    %ax, %gs

    movl    %cr4, %eax
    orl    $(CR4_PSE), %eax
    movl    %eax, %cr4 ②

    movl    (start-12), %eax
    movl    %eax, %cr3 ③

    movl    %cr0, %eax
    orl    $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0 ④

    movl    (start-4), %esp ⑤
    call    *(start-8) ⑥
    movw    $0x8a00, %ax
    movw    %ax, %dx
    outw    %ax, %dx
    movw    $0x8ae0, %ax
    outw    %ax, %dx

spin:
```

cionira za izvršenje  
ter, koja se uvezuje  
atu sadržaja sekciјe

je definirana izdvojena

```
    jmp      spin

    .p2align 2
gdt:
    SEG_NULLASM
    SEG_ASM(STA_X|STA_R, 0, 0xffffffff)
    SEG_ASM(STA_W, 0, 0xffffffff)
gddesc:
    .word   (gddesc - gdt - 1)
    .long   gdt
```

Program kojeg izvršavaju AP jezgra prilikom pokretanja, sličan je dijelu program za učitavanje kernela kojeg pokreće BIOS. Nakon linije ①, AP jezgro prelazi u protected mod operacije. Pošto se završi inicijalizacija selektora spram GDT tabele definirane simbolom `gdt`, u liniji ②, aktivira se podrška za straničenje sa stranicama veličine 4MB.

Treći parametar koji je dat od strane BSP jezgra, u liniji ③, učitava se u register `cr3`. Ovim se određuje lokacija za PD tabelu inicijalnog straničenja AP jezgra. Straničenje se aktivira u liniji ④, nakon čega se postavlja lokacija za stek, i to spram prvog parametra kojeg je obezbijedilo BSP jezgro. Konačno, u liniji ⑤, skače se u funkciju koju je kao drugi parametar dalo BSP jezgro.

Do linije ⑥, AP jezgro koje se budi, izvršilo je tranziciju u protected mod i postavilo inicijalno straničenje od dvije stranice veličine 4MB, koje je određeno kernel varijablom `entrypgdir`. Ovim je AP jezgro postavljeno u poziciju da može izvršavati bilo koju funkciju koja je sastavni dio ranije učitanog kernel koda. Funkcija `mpenter`, drugi parametar kojeg obezbijeduje BSP jezgro, koristi se za skok kojeg AP jezgro izvodi u liniji ⑥. Slijedeći kod preuzet je iz fajla `main.c` u kojem je implementirana ova funkcija.

```
static void mpenter(void)
{
    switch kvm(); ①
    seginit(); ②
    lapicinit(); ③
    mpmain(); ④
}
```

AP jezgro, kada skoči u funkciju `mpenter`, prvo aktivira konačnu konfiguraciju kernel straničenja u liniji ❶. Nakon ovog, u liniji ❷, aktivira konačnu konfiguraciju segmentiranja. Kako bi moglo primati prekide i da bi integrirani tajmer prekida u jednakim vremenski intervalima, jezgro inicijalizira sopstveni LAPIC kontroler u liniji ❸. Konačno, u liniji ❹ putem funkcije `mpmain`, jezgro počinje da izvršava kod od trenutno aktiviranih aplikacija. Dakle, sva AP jezgra završavaju svoj proces pokretanja u funkciji `mpmain`. Nakon što izvrši pokretanje operativnog sistema, `mpmain` je konačna odrednica i za BSP jezgro.

Kada su u potpunosti pokrenuta sva jezgra, postoji mogućnost da dva ili više jezgra u istom trenutku pokušavaju pristupiti istom resursu unutar kernela. Ova situacija izaziva problem koji se naziva stanje utrke. Problem se rješava putem sinhronizacijskih primitiva koji će biti tema posebnog poglavlja.

Kroz sist...  
sistem ost...  
obične apl...  
je su impl...  
kom izvrš...  
ti bez posl...

Posebni...  
prekidanje...  
mogu kon...  
šenje neke...  
način da o...  
ge aplikac...  
već u slijed...  
ti sa svojim...  
u vremenom...  
kratak inter...  
među izvršen...  
od da sve...  
ostvaruju...  
kurentno...  
dok se na...  
vanje već...  
tiraju ovu...  
koje spada...

Da bi ...  
sistem tre...

## Tretman prekida

Kroz sistem prekida, koji je implementiran u hardveru, operativni sistem ostvaruje efikasnu komunikaciju sa vanjskim uređajima, a obične aplikacije dobijaju mogućnost poziva određenih funkcija koje su implementirane u kernelu. Dodatno, greške koje nastaju tokom izvršavanja korisničkih aplikacija moguće je adekvatno tretirati bez posljedica po ostale aplikacije koje se izvršavaju istovremeno.

Posebno bitni prekidi dolaze od tajmera, uređaja koji omogućava prekidanje procesora u jednakim vremenskim intervalima koji se mogu konfigurirati. Kada se zbog servisiranja tajmera prekine izvršenje neke aplikacije, operativni sistem može tretirati ovaj prekid na način da od tog trenutka procesor zaposli izvršavanjem neke druge aplikacije. Stanje od aplikacije koja je prekinuta se snima, kako bi već u slijedećoj instanci prekida od tajmer aplikacija mogla nastaviti sa svojim izvršenjem. Tajmer se konfigurira da prekida procesor u vremenskim intervalima perioda 10ms – 100ms, što je dovoljno kratak interval da korisnik ne primijeti prebacivanje procesora između izvršavanja različitih aplikacija, a ujedno i dovoljno dug period da sve aplikacije, između kojih se preklapanje procesora događa, ostvaruju napredak u izvršenju. Na ovaj način omogućava se konkurentno izvršavanje aplikacija čak i ako procesor ima jedno jezgro, dok se na procesoru sa više jezgri omogućava konkurentno izvršavanje većeg broja aplikacija od broja jezgri. Sistemi koji implementiraju ovu sposobnost nazivaju se *multitasking* operativni sistemi u koje spada i XV6.

Da bi na Intel platformi funkcionsao sistem prekida, operativni sistem treba implementirati sve neophodne ISR rutine, te kreirati

i konfigurirati IDT tabelu. U ovom poglavlju analizirat ćemo XV6 kernel funkcije koje služe ovoj namjeni.

## Konfiguracija IDT tabele

IDT tabela se sastoji od maksimalno 256 elemenata koji se označavaju kao gate deskriptori. U fajlu `trap.c` XV6 defnira globalni niz `idt` koji predstavlja implementaciju IDTa. Niz je definiran na slijedeći način:

```
struct gatedesc idt[256];
```

Pojedinačni elementi u nizu su strukture tipa `gatedesc` koji se na jednostavan način mogu konfigurirati putem makroa `SETGATE`. Ovaj makro uzima slijede parametre:

1. Element u `idt` nizu koji se modificira.
2. Tip deskriptora: 1 → `trap gate`, 0 → `interrupt gate`. Za `interrupt gate` ISR rutina se izvodi sa svim prekidima maskiranim na procesoru. Dok su za `trap gate` prekidi uključeni u toku izvođenja ISR rutine.
3. Redni broj deskriptora u GDTu koji će se keširati u CS selektoru tokom tretmana prekida.
4. Adresa ISR rutine unutar segmenta odabranog putem CS selektora.
5. Minimalni nivo privilegija potreban da se pokrene ISR rutina putem instrukcije `int`.

Kompletna IDT tabela konfigurira se u funkciji `tvinit` u toku pokretanja operativnog sistema. Slijedi kod ove funkcije koja je implementirana u fajlu `trap.c`.

```
void tvinit(void)
{
    int i;

    for(i = 0; i < 256; i++)
```

```

    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0); ①
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
             vectors[T_SYSCALL], DPL_USER); ②
    initlock(&tickslock, "time");
}

```

U liniji ① funkcija konfigurira elemente u IDTu koji su asocirani sa vektorima prekida uređaja i iznimki, dok se u liniji ② postavlja jedan element asociran sa vektorom T\_SYSCALL koji služi za implementaciju sistemskih poziva na XV6 kernelu.

Za sve elemente definiraju se pointeri na njihove ISR rutine (`vectors[i]`) uz postavku da se sve ISR rutine izvršavaju sa najvišim nivoom privilegija (SEG\_KCODE<<3). Za vektor T\_SYSCALL postoje bitne razlike u postavkama u odnosu na sve ostale vektore. Ovaj vektor može se koristiti sa instrukcijom `int`. Nadalje, kad se počne izvršavati njegova ISR rutina, prekidi na procesoru neće biti maskirani.

Sva procesorska jezgra koriste istu IDT tabelu, obzirom da pozivaju funkciju `idtinit` koja učitava adresu niza `idt` u registar `idtr` onog jezgra koje izvršava datu funkciju. BSP jezgro poziva ovu funkciju pred kraj pokretanja operativnog sistema, a AP jezgra na kraju sekvence buđenja. Dakle, dok tretiraju prekide koji im prosleđuju njihovi LAPIC kontroleri, jezgra izvršavaju isti kernel kod.

U bilo kojem trenutku jezgro može biti zaposleno izvršavanjem jednog od dva zadatka:

- servisiranje prekida,
- izvršavanje koda nekog procesa.

Zadatak koji obavlja neko jezgro u datom trenutku nije unaprijed određen, već ovisi od nastale situacije. Na primjer, u sistemu od četiri jezgra u nekom trenutku dva jezgra mogu izvršavati kod od dvije aplikacija, dok druga dva jezgra servisiraju prekide od različitih uređaja. U nekoj drugoj instanci vremena, sva četiri jezgra mogu biti zaposlena servisiranjem prekida od njihovog lokalnog tajmera.

Dok izvršava kod nekog procesa, jezgro je u najnižem nivou privilegija i može primati prekide. Sa druge strane, na osnovu konfiguracije IDT tabele, kad servisira bilo koji prekid, jezgro prelazi u najviši nivo privilegija sa onemogućenim prekidima, osim u slučaju prekida uslijed sistemskog poziva kad novi prekidi ostaju omogućeni.

## ISR rutine

Svaki elemenat IDTa sadrži adresu ISR rutine koja se pokreće kada se dogodi prekid sa odgovarajućim vektorom. XV6 definira sve ISR rutine u fajlu `vectors.S`. U slijedećem listingu prikazan je sadržaj ovog fajla gdje su izostavljeni suvišni segmenti koda na lokacijama na kojim se nalaze komentari.

```
.globl vector0
vector0: ①
    pushl $0
    pushl $0
    jmp alltraps

.globl vector1
vector1: ②
    pushl $0
    pushl $1
    jmp alltraps

.globl vector2
vector2:
    pushl $0
    pushl $2
    jmp alltraps

# ...

.globl vector8
vector8: ③
    pushl $8
    jmp alltraps

.globl vector9
vector9:
    pushl $0
    pushl $9
    jmp alltraps

.globl vector10
```

nižem nivou pri-  
na osnovu konfi-  
ljezgro prelazi u  
osim u slučaju  
ostaju omoguće-

pokreće kada  
definira sve ISR  
je sadržaj  
na lokacijama

```
vector10: ④
    pushl $10
    jmp alltraps

# ...

.globl vector254
vector254: ⑤
    pushl $0
    pushl $254
    jmp alltraps

.globl vector255
vector255: ⑥
    pushl $0
    pushl $255
    jmp alltraps

.data
.globl vectors
vectors:
    .long vector0
    .long vector1
    .long vector2
    .long vector3

# ...
    .long vector254
    .long vector255
```

Kao što je vidljivo u linijama od ① do ⑥, svaka ISR rutina je kratka sekvenca asembler instrukcija koja počinje sa oznakom vektor $K$  ( $K$  iz intervala  $0 - 255$ ), a završava bezuslovnim skokom u funkciju asociranu sa oznakom alltraps. Prije nego što izvrši skok u funkciju, svaki ISR postavlja argumente na stek. Većina rutina, npr. ①, ②, ⑤, na stek stavlja dva argumenta. Prvi argument uvijek je nula, dok drugi argument predstavlja redni broj ISR rutine. Neke rutine, npr. ③, ④, na stek stavljuju samo redni broj rutine. Obzirom da će redni broj svake rutine korespondirati vektoru prekida koji pokreće tu rutinu, ISR na steku kao drugi argument proslijedi vektor prekida koji se tretira u datom trenutku u funkciji alltraps. Prvi parametar predstavlja Error Code prekida koji potiče od određenih iznimki za koje procesor na steku automatski postavlja ovaj parametar. Za sistemski poziv, sve hardverske prekide, te određene iznimke koje ne-

maju Error Code, radi uniformnosti se prvi parametar postavlja na vrijednost 0, i to ručno putem ISR.

Od linije ⑥ počinje definicija niza od 256 elemenata pod imenom vectors. Elementi niza su oznake ISR rutina, tj. lokacije u memoriji od kojih počinje prva instrukcija ISR rutine. Niz vectors, putem makroa SETGATE, koristi se u funkciji tvinit kod konfiguracije redova IDT tabele. Time se svaki pojedinačni vektor prekida uvezuje sa odgovarajućom ISR rutinom iz fajla vectors.S.

Kao što je objašnjeno u ranijim poglavljima, kada nastane prekid, procesor u skladu sa deskriptorom iz IDT tabele mijenja nivo privilegija, a na stek snima ključne registre i eventualni Error Code. Tačan broj snimljenih registara i lokacija steka na koju se snimanje vrši ovise od nivoa privilegija koje je procesor imao u trenutku nastanka prekida. Potom, u skladu sa vektorom prekida, procesor iz IDT uzima odgovarajući ISR, kojeg zatim i izvršava. Svaki ISR na stek dodaje vektor prekida i eventualni nulti argument, a zatim vrši skok u funkciju alltraps. Slijedi kod ove funkcije koja je implementirana u asembler fajlu trapasm.S.

```
#include "mmu.h"

.globl alltraps
alltraps:
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs ①
    pushal ②

    movw $(SEG_KDATA<<3), %ax ③
    movw %ax, %ds
    movw %ax, %es
    movw $(SEG_KCPU<<3), %ax
    movw %ax, %fs
    movw %ax, %gs ④

    pushl %esp ⑤
    call trap ⑥
    addl $4, %esp

.globl trapret
trapret:
```

```
popal
popl %gs
popl %fs
popl %es
popl %ds
addl $6
iret
```

Funkcija koje je kreira kida, nakon tira prekida je potrebna gram, kako

Od početku segmenta ju se svi registri u pletnog struktura koja je u man prekida tri sa odgovarajućim iznimki će biti procesorom ja tretirati to je početak

Po treći argument lokaciju prekida tao program će addl instrukcija prekinuti i preuzimati putem programne funkcije, dodala funkcije, ljeni na

```
popal ⑦  
popl %gs  
popl %fs  
popl %es  
popl %ds ⑧  
addl $0x8, %esp ⑨  
iret
```

Funkcija ima prvenstveni zadatak da na stek snimi sve registre koje je koristio program koji se izvršavao u trenutku nastanka prekida, nakon čega poziva specijaliziranu funkciju koja adekvatno tretira prekid. Po povratku iz funkcije koja je tretirala prekid, sa steka je potrebno vratiti na procesor cijelokupno stanje prekinutog program, kako bi isti mogao nastaviti sa izvršenjem.

Od početka funkcije do linije ①, putem push instrukcije se snimaju segmentni registri, a zatim u liniji ② jednom instrukcijom snimaju se svi registri opšte namjene. Ovim je obavljen snimanje kompletног stanja prekinutog programa, a na steku je formirana struktura koja se označava kao `trapframe`. Od ovog trenutka počinje tretman prekida. Između linija ③ i ④ inicijaliziraju se segmentni registri sa odgovarajućim kernel deskriptorima, obzirom da se tretman iznimki obavlja putem kernel koda sa najvišim nivoom privilegija procesora. Prije nego što se u liniji ⑥ izvrši skok u funkciju `trap` koja tretira prekid, na stek se dodaje jedni argument za ovu funkciju, a to je pointer na početak `trapframe` strukture.

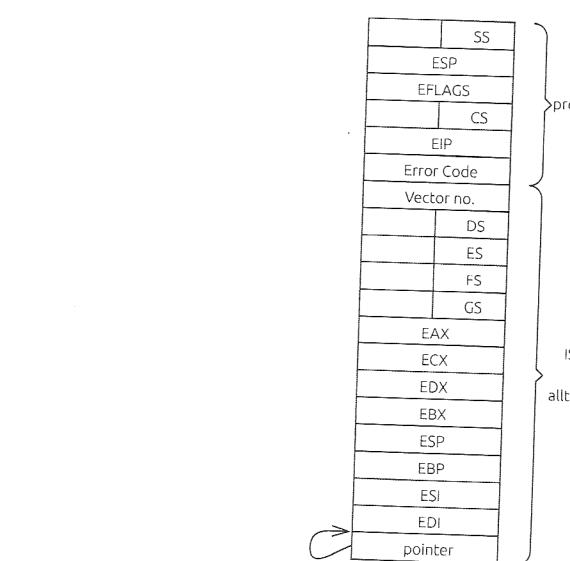
Po tretmanu prekida, procesor se vraća u funkciju `alltraps` na lokaciju prve instrukcije nakon linije ⑤. Obzirom da je na steku ostao programski brojač od poziva `trap`, isti se prvo uklanja putem `addl` instrukcije, nakon čega počinje vraćanje na procesor stanja prekinutog programa. Prvo se u liniji ⑦ jednim potezom sa steka preuzimaju svi snimljeni registri opšte namjene, zatim do linije ⑧ putem pojedinačnih `pop` instrukcija preuzimaju se snimljeni segmentni registri. U liniji ⑨ sa steka se uklanjuju dva argumenta koje je dodala ISR rutina. Konačno, putem `iret` instrukcije na kraju funkcije, na procesor se vraćaju svi registri koji su automatski snimljeni na stek prilikom nastanka prekida. Ovim prestaje izvršenje

kernel koda koji je tretirao prekid, a program nastavlja dalje kao da nije ni prekidan.

varaju raspored  
strukture koja

## Struktura trapframe i procesiranje prekida

Svi prekidi tretiraju se u kernel funkciji trap. Od trenutka nastanka prekida do trenutka poziva ova funkcije, kod kojeg izvršava procesor ima jedinu namjenu da na steku pripremi argument kojeg očekuje ova funkcija.



Slika 7-1. Izgled steka prije poziva funkcije trap

Slika 7-1 prikazuje sadržaj steka koji nastao uslijed automatskog snimanja stanja nekih registara prilikom nastanka prekida i izvršavanja ISR rutine i funkcije alltraps.

Kreirani sadržaj na steku može se posmatrati kao da se sastoji iz dva elementa. Prvi element je struktura tipa trapframe, a drugi, čija je adresa trenutno nalazi u registru esp, je pointer na prvi element. Polja u strukturi trapframe pažljivo su postavljena da odgo-

```
struct trapframe
{
    uint edi;
    uint esi;
    uint ebp;
    uint oesp;
    uint ebx;
    uint edx;
    uint ecx;
    uint eax;

    ushort gs;
    ushort paddin;
    ushort fs;
    ushort paddin;
    ushort es;
    ushort paddin;
    ushort ds;
    ushort paddin;
    uint trapno;

    uint err;
    uint eip;
    ushort cs;
    ushort paddin;
    uint eflags;

    uint esp;
    ushort ss;
    ushort paddin
};
```

- ① Polje kreirano
- ② Polje kreirano trukcija pu
- ③ Polje kreirano
- ④ Polje kreirano strane progr
- ⑤ Polje kreirano da.

varaju rasporedu snimljenih registara na steku. Slijedi definicija ove strukture koja je preuzeta iz fajla x86.h.

```
struct trapframe {  
    uint edi;          ①  
    uint esi;          ①  
    uint ebp;          ①  
    uint oesp;         ①  
    uint ebx;          ①  
    uint edx;          ①  
    uint ecx;          ①  
    uint eax;          ①  
  
    ushort gs;          ②  
    ushort padding1;   ②  
    ushort fs;          ②  
    ushort padding2;   ②  
    ushort es;          ②  
    ushort padding3;   ②  
    ushort ds;          ②  
    ushort padding4;   ②  
    uint trapno;       ③  
  
    uint err;           ④  
    uint eip;           ⑤  
    ushort cs;          ⑤  
    ushort padding5;   ⑤  
    uint eflags;        ⑤  
  
    uint esp;           ⑤  
    ushort ss;          ⑤  
    ushort padding6;   ⑤  
};
```

- ① Polja kreirana u funkciji `alltraps` putem instrukcije `pusha`.
- ② Polja kreirana u funkciji `alltraps` putem pojedinačnih instrukcija `pushl`.
- ③ Polje kreirano u ISR rutini putem instrukcije `pushl`.
- ④ Polje kreirano u ISR rutini putem instrukcije `pushl` ili od strane procesora u trenutku nastanka prekida.
- ⑤ Polje kreirana od strane procesora u trenutku nastanka prekida.

Kernel funkcija `trap` dolazi do strukture `trapframe` putem jedinog argumenta kojeg uzima, a to je pointer na tu strukturu. Unutar ove strukture, funkcija ima sve neophodne informacije da bi adekvatno adresirala prekid. Na osnovu vektora prekida koji se nalazi u polju `trapno`, funkcija može odrediti razlog nastanka prekida, a zatim pozvati specijalizirane funkcije za tretman konkretnog prekida. Ukoliko je prekid nastao uslijed neke iznimke, kada je vektor prekida u intervalu 0 — 31, dodatne informacije o iznimci mogu se dobiti u polju `err`. Ukoliko je prekid nastao zbog sistemskog poziva, kada vektor prekida ima vrijednost `T_SYSCALL` (64), argumenti za sistemski poziv nalaze se na steku procesa koji je izazvao prekid. Konačno, ukoliko je prekid nastao od uređaja, dodatne informacije mogu se dobiti od kontrolera uređaja koji je izazvao prekid, putem `in` i `out` instrukcija ili pristupanjem memorijiski mapiranim adresama registara kontrolera.

Slijedeći listing preuzet je iz fajla `trap.c` u kojem je definirana funkcija `trap`.

```
void trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){ ❶
        if(proc->killed)
            exit();
        proc->tf = tf;
        syscall();
        if(proc->killed)
            exit();
        return;
    }

    switch(tf->trapno){ ❷
        case T_IRQ0 + IRQ_TIMER: ❸
            if(cpu->id == 0){
                acquire(&tickslock);
                ticks++;
                wakeup(&ticks);
                release(&tickslock);
            }
            lapiceoi();
            break;
        case T_IRQ0 + IRQ_IDE:
            ideintr();
            lapiceoi();
            break;
    }
}
```

```
case T_I
break;
case T_I
kbdtint
lapice
break;
case T_I
uartin
lapice
break;
case T_I
case T_I
cprint
lapice
break;

default:
if(proc
cpri
pani
}
cprint
```

```
proc->
}
if(proc
exit()
if(proc
proc
yield(
if(proc
exit()
}

)
```

Funkcija  
nosti `tf->t`  
tretman ko  
bloku ❶, pr  
❷.

Sistemsk  
čega se, pov  
izvršio prek

trapframe putem jedinu strukturu. Unutar informacije da bi adek- rekida koji se nalazi u stanka prekida, a za- i konkretnog prekida. kada je vektor preki- znimci mogu se dobiti stemskog poziva, kada argumenti za sistem- zvao prekid. Konačno, e informacije mogu se prekid, putem in i out ranim adresama regis-

u kojem je definirana

```
case T_IRQ0 + IRQ_IDE+1:  
    break;  
case T_IRQ0 + IRQ_KBD:  
    kbdintr();  
    lapiceoi();  
    break;  
case T_IRQ0 + IRQ_COM1:  
    uartintr();  
    lapiceoi();  
    break;  
case T_IRQ0 + 7:  
case T_IRQ0 + IRQ_SPURIOUS:  
    cprintf("cpu%d: spurious interrupt at %x:%x\n",  
            cpu->id, tf->cs, tf->eip);  
    lapiceoi();  
    break;  
  
default: ④  
    if(proc == 0 || (tf->cs&3) == 0){  
        cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%lx)\n",  
                tf->trapno, cpu->id, tf->eip, rcr2());  
        panic("trap");  
    }  
    cprintf("pid %d %s: trap %d err %d on cpu %d "  
           "eip 0x%lx addr 0x%lx--kill proc\n",  
           proc->pid, proc->name, tf->trapno,  
           tf->err, cpu->id, tf->eip,  
           rcr2());  
    proc->killed = 1;  
}  
if(proc && proc->killed && (tf->cs&3) == DPL_USER)  
    exit(); ⑤  
if(proc &&  
    proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)  
    yield(); ⑥  
if(proc && proc->killed && (tf->cs&3) == DPL_USER)  
    exit();  
}
```

Funkcija konceptualno radi kao dispečer, tj. na osnovu vrijednosti `tf->trapno` određuje koju će specijaliziranu funkciju pozvati za tretman konkretnog prekida. Funkcija procesira sistemski poziv u bloku ①, prekide od uređaja u bloku ②, dok iznimke tretira u bloku ④.

Sistemski poziv kompletno se tretira pozivom `syscall`, nakon čega se, povratkom u `alltraps`, nastavlja izvršavanje procesa koji je izvršio prekid.

Iznimke se tretiraju u bloku ④ u ovisnosti od izvora iznimke. Ukoliko je iznimka nastala dok se izvršavao kod od operativnog sistema, tj. u toku tretmana nekog prekida, nakon ispisivanja adekvatne poruke vrši se zaustavljanje cjelokupnog operativnog sistema pozivom funkcije `panic`. Međutim, ukoliko je iznimka nastala dok se izvršavala neka aplikacija, tada se u bloku ④ aplikacija priprema za terminiranje postavljanjem polja `killed` u strukturi `proc` na vrijednost 1. Terminacija aplikacija se događa u liniji ⑤.

Hardverski prekidi koji se tretiraju u bloku ② dolaze od: tajmera, diska, tastature i serijskog porta. Dodatno, u ovom bloku tretiraju se i lažni hardverski prekidi koji imaju vektor `IRQ_SPURIOUS`. Tretman tajmera u bloku ③ posebno je interesantan, jer omogućava multitasking.

Kada na nekom jezgru nastane prekid od tajmera, u okviru bloka ③ jezgro šalje EOI signal LAPIC kontroleru kako bi se nastavilo slanje budućih tajmer prekida. Samo ono jezgro čiji je APIC identifikator 0, u okviru bloka ③, inkrementira globalnu varijablu `ticks` i pozivom funkcije `wakeup_budi` one procese koji spavaju na kanalu čiji je identifikator adresa `variable_ticks`. Globalna varijabla `ticks` u kernelu služi za održavanje koncepta prolaska vremena. Procesi od kernela mogu zatražiti da se ne izvršavaju, tj. da budu uspavani, sve do isteka određenog vremenskog perioda. Nakon buđenja uslijed prekida od tajmera, inspekциjom nove vrijednosti varijable `ticks`, moguće je utvrditi istek željenog vremenskog intervala, nakon čega je procesu omogućen daljnji nastavak izvršenja. Pored bloka ③, prilikom tretmana prekida tajmera svako jezgro izvršava i liniju koda ⑥. Pozivom funkcije `yield` događa se *promjena konteksta*, uslijed koje jezgro u narednom vremenskom periodu počinje izvršavati kod nekog drugog procesa, a proces koji se izvršavao na datom jezgru u trenutku nastanka prekida ostaje u zamrznutom stanju u liniji ⑥, sve dok ga neko jezgro u budućnosti ne preuzme na izvršavanje. Detalje vezane za promjenu konteksta predstavit ćemo u posebnom poglavlju posvećenom procesima.

Nakon  
je pravi  
ratnom  
ces. Me  
mogu d  
jednog  
nja kod  
globaln  
manipu  
rezultat  
situacij  
pu zajec  
osnovn

## Stanje

Posljedi  
ćemo n  
definira

```
struct
int d
struct
};
```

```
struct
struct
} my_l
```

```
void i
struct
n->da
```

## Sinhronizacija

Nakon pokretanja AP jezgri unutar operativnog sistema omogućen je pravi paralelizam, obzirom da svako pojedinačno jezgro u separatnom adresnom prostoru može izvršavati njemu dodjeljen proces. Međutim, u toku tretmana prekida koji se na dvije ili više jezgre mogu dogoditi istovremeno, jezgra počinju izvršavati kod unutar jednog adresnog prostora koji pripada kernelu. Tokom izvršavanja koda za tretman prekida, svako jezgro može slobodno pristupati globalnim varijablama kernela. Ukoliko dva ili više jezgra pokušaju manipulirati istom memorijskom lokacijom istovremeno, konačni rezultat ovih operacija nije moguće predvidjeti. Nastanak ovakvih situacija neophodno je spriječiti uvođenjem sinhronizacije u pristupu zajedničkim resursima. U ovom poglavlju analizirat ćemo *brave*, osnovni primitiv za sinhronizaciju unutar XV6 kernela.

### Stanje utrke

Posljedice istovremene manipulacije zajedničkog resursa razmotrit ćemo na primjeru jednostruko povezane liste cijelih brojeva koja je definirana na slijedeći način.

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

struct list {
    struct node *first;
} my_list;

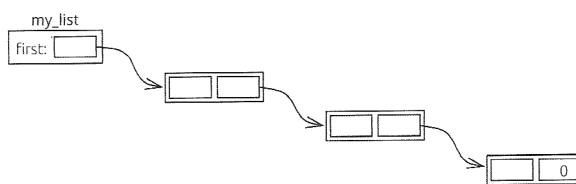
void insert(int data) {
    struct node *n = malloc(sizeof(struct node));
    n->data = data; ①
    n->next = my_list.first;
    my_list.first = n;
}
```

```

    n->next = my_list.first; ②
    my_list.first = n; ③
}

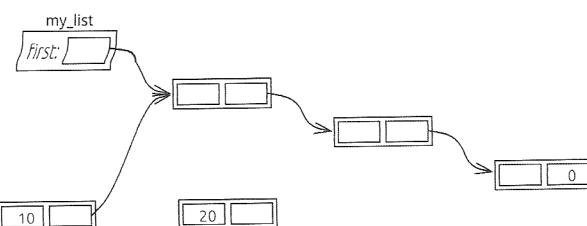
```

Početak liste definiran je globalnom kernel varijablu `my_list`, dok se novi elementi u listu mogu dodavati putem funkcije `insert`. Pretpostavimo da su se na dva procesorska jezgra istovremeno dogodili prekidi, a da tretman oba prekida zahtijeva manipulaciju liste definirane putem globalne varijable `my_list`. Inicijalno stanje liste prikazano je na slici 8-1.



Slika 8-1. Inicijalno stanje liste

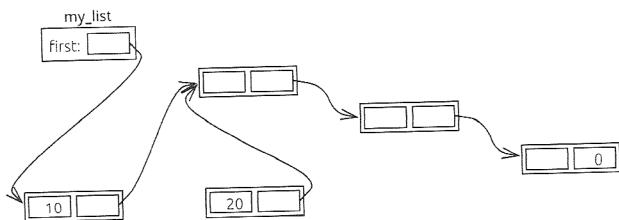
Pretpostavimo da oba jezgra pozivaju funkciju `insert`, pri čemu prvo jezgro poziva `insert(10)`, nakon čega sa malim kašnjenjem drugo jezgro poziva `insert(20)`. Neka prvo jezgro izvršava liniju ② u trenutku kada drugo jezgro izvršava kod iz linije ①. Stanje kreiranih čvorova i globalne liste nakon ovog trenutka prikazano je na slici 8-2.



Slika 8-2. Stanje liste u procesu dok dva jezgra dodaju čvorove — trenutak 1

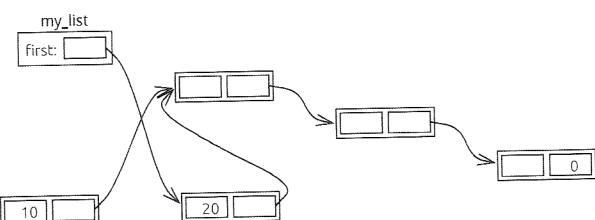
Nadalje, pretpostavimo da u toku izvršavanja linije ②, drugo jezgro pročita vrijednost polja `my_list.first` prije nego što ovo polje

promjeni prvo jezgro izvršavajući liniju ❸. Stanje kreiranih čvorova i globalne liste nakon ovog trenutka prikazano je na slici 8-3.



Slika 8-3. Stanje liste u procesu dok dva jezgra dodaju čvorove — trenutak 2

Konačno, kada drugo jezgro izvrši liniju ❸, stanje kreiranih čvorova i globalne liste prikazano je na slici 8-4.



Slika 8-4. Stanje liste nakon dodavanja čvorova

Istovremenim izvršavanjem funkcije `insert` od strane dva jezgra kreirana su dva elementa tipa `node`. Element nastao djelovanjem drugog jezgra pozicioniran je na početak globalne liste, dok je drugi element, rezultat operacije prvog jezgra, ostao van globalne liste. Obzirom da lista nije uvećana za dva elementa, istovremeni pristup globalnoj listi proizveo je neispravan rezultat. Dodatno, element kojeg je dinamički kreiralo prvo jezgro nije moguće dealocirati, obzirom da ne postoji pointer na njegovu lokaciju.

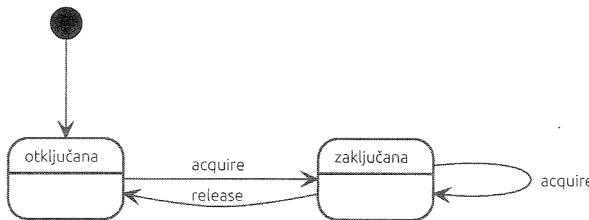
Ukoliko bi jezgra istovremeno izvršavala funkciju `insert` sa redoslijedom različitim od opisanog, konačni rezultat bi potencijalno bio drugačiji, a za neke sekvence izvršavanja rezultat bi bio i ispravan. U opštem slučaju, situacije u kojem dva ili više jezgra istovre-

meno pristupaju dijeljenom resursu nazivaju se *stanja utrke* (race conditions). Kada se utrka jezgri završi, manipulirani resurs ostaje u proizvoljnem stanju.

Da bi se izbjegle situacije u kojem nastaju stanja utrke, potrebno je uvesti redoslijed koji mora biti poštovan kada više jezgri istovremeno zahtijeva pristup istom resursu. Pozicioniranjem sinhronizacijskih primitiva na adekvatnim lokacijama u kodu moguće je uređiti pristup pojedinačnim dijeljenim resursima. Brava je najčešći sinhronizacijski mehanizam kojeg implementiraju operativni sistemi, uključujući i XV6.

## Brave

Brave su podatkovne strukture dizajnirane da omoguće ekskluzivni pristup nekom resursu. Svaka brava ima dva stanja i dvije operacije koje podržava. Na slici 8-5 prikazane su tranzicije između stanja brave spram podržanih operacija.



Slika 8-5. Dijagram stanja brave

Ukoliko neko jezgro pozove funkciju `acquire` na otključanoj bravi, jezgro odmah prestaje izvršavati kod funkcije `acquire`, a brava mijenja stanje, tj. postaje zaključana. Sa druge strane, ukoliko jezgro pozove funkciju `acquire` dok je brava zaključana, brava ostaje u istom stanju, a jezgro nastavlja izvršavati kod funkcije `acquire` sve dok brava ne promjeni stanje. Brava postaje otključana ukoliko se na njoj pozove funkcija `release`. Otključavanje brave treba da obavi ono jezgro koje je zaključalo bravu. Segment koda između pozici

stanja utrke (race condition) resurs ostaje u

utrke, potrebno  
više jezgri istovremeno  
moguće je uređati je najčešći sin-  
operativni sistemi,

oguće ekskluzivni  
dvije operacije  
među stanja bra-

acquire

ptključanoj bra-  
acquire, a brava  
ekoliko jezgro  
ava ostaje u is-  
je acquire sve  
čana ukoliko se  
treba da oba-  
da između pozicija

va funkcija `acquire` i `release` često se označava kao *kritična sekcija* (critical section). Unutar kritične sekcije vrši se manipulacija određenim zajedničkim resursom.

Da bi se spriječilo stanje utrke na nekom dijeljenom resursu, za taj resurs potrebno je kreirati jednu bravu, te adekvatno pozicionirati operacije `acquire` i `release` sa tom bravom na lokacijama u kodu koje predstavljaju kritične sekcije za dati resurs. Na primjer, slijedeći kod rješava problem istovremenog pristupa jednostruko povezanoj listi iz prethodne sekcije.

```
struct node {  
    int data;  
    struct node *next;  
};  
  
struct list {  
    struct node *first;  
    struct lock list_lock; ①  
} my_list;  
  
void insert(int data) {  
    acquire(&my_list.list_lock); ②  
    struct node *n = malloc(sizeof(struct node));  
    n->data = data;  
    n->next = my_list.first;  
    my_list.first = n;  
    release(&my_list.list_lock); ③  
}
```

Svaka povezana lista dobija svoju bravu u liniji ① kao polje strukture `list` pod imenom `list_lock`, dok se kompletan kod funkcije `insert` postavlja između operacija `acquire` i `release` na bravi koja je polje globalne varijable `my_list`. Ovo se čini da bi različita procesorske jezgra sinhronizirala pristup linijama koda između oznaka ② i ③, u kojim se vrši manipulacija stanjem globalne varijable `my_list`. Jezgro koje prvo pozove funkciju `acquire` u liniji ② dobija ekskluzivni pristup kritičnoj sekciji koda unutar koje može manipulirati stanjem varijable `my_list` bez bojazni da to istovremeno radi i neko drugo jezgro.

Postoje različite implementacije brave, u slijedećem kodu data je *spinlock* varijanta brave:

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

Funkcija `xchg` uzima neku adresu i vrijednost koju treba postaviti na tu adresu, a vraća staru vrijednost pročitanu sa date adrese. Ova funkcija u linijama ❶ i ❷ koristi se za manipulaciju polja `locked` u strukturi `lock` koja predstavlja neku bravu. Petlja u liniji ❶ ima zadatak da u funkciji `acquire` zadržava ono jezgro koje ovu funkciju pozove na zaključanoj bravu, i to sve dok neko drugo jezgro eksplicitno ne otključa bravu pozivom funkcije `release`.

Brave su dijeljeni resurs, obzirom da različita jezgra izvršavajući funkcije `acquire` i `release` pokušavaju manipulirati unutrašnjim stanjem brave. Zbog toga, funkcija `xchg` mora biti implementirana tako da se čitanje stare i postavljanje nove vrijednosti na adresi koja se proslijedi kao prvi parametar funkcije, obavlja atomski, tj. kao jedna operacija, bez ikakvih smetnji. Ovo je moguće realizirati putem slijedećeg koda:

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

Funkcija `xchg` implementira se kao kombinacija C koda i asemblera. Putem `asm` direktive u liniji ❶ u funkciju se injektiraju dvije

asembler i  
ku jezgru  
dato jezgru  
zivni prist  
kojom se v  
sult, a zat

## Zastoji

Brave pre  
posebnu p  
racija koje  
ćemo slijed

```
struct lock  
{  
    struct lock *next;  
};  
  
void f1()  
{  
    acquire(&lock);  
    acquire(&lock);  
    // manip...  
    release(&lock);  
    release(&lock);  
}  
  
void f2()  
{  
    acquire(&lock);  
    acquire(&lock);  
    // manip...  
    release(&lock);  
    release(&lock);  
}
```

Neka je  
brave koje  
U funkcija  
se na poče  
u činjenici  
prvo zakl

u kodu data je asembler instrukcije `lock` i `xchgl`. Izvršavanje instrukcije `lock` za neku jezgru ima posljedicu da tokom izvršavanja slijedeće instrukcije dato jezgro na hardverskom nivou spram ostalih jezgri ima ekskluzivni pristup memoriji. Naredna instrukcija u `asm` direktivi je `xchgl` kojom se vrijednost pročitana sa adresi `addr` kopira u varijablu `result`, a zatim se sadržaj varijable `newval` postavlja na adresu `addr`.

## Zastoji

Brave predstavljaju koristan sinhronizacijski primitiv. Međutim, posebnu pažnju treba obratiti prilikom pozicioniranja u kodu operacija koje manipuliraju različitim bravama. Kao ilustraciju koristit ćemo slijedeći kod:

```
struct lock A;
struct lock B;

void f1() {
    acquire(&A); ①
    acquire(&B); ②

    // manipulacija resursima

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

void f2() {
    acquire(&B); ③
    acquire(&A); ④

    // manipulacija resursima

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

Neka je gornji kod sastavni dio kernela. Globalne varijable `A` i `B` su brave koje služe za sinhronizaciju pristupa za dva različita resursa. U funkcijama `f1` i `f2` vrši se manipulacija sa oba resursa, zbog toga se na početku obje funkcije zaključavaju obje brave. Jedina razlika je u činjenici da se u liniji ① prvo zaključava brava `A`, dok se u liniji ③ prvo zaključava brava `B`.

Prepostavimo da na dva procesorska jezgra istovremeno nastaju različiti prekidi i da prvo jezgro tokom tretmana svog prekida poziva funkciju f1, dok drugo jezgro za tretman svog prekida poziva funkciju f2. Nadalje, prepostavimo da u isto vrijeme kada prvo jezgro izvrši kod linije ①, drugo jezgro izvrši liniju ③. Od tog trenutka obje brave su u zaključanom stanju, a prvo jezgro izvršavanjem linije ② ostaje unutar funkcije acquire u beskonačnoj petlji, dok isto vrijedi za drugo jezgro uslijed izvršavanja linije ④.

Opisana sekvenca zaključavanja brava dovela je do činjenice da oba jezgra ostaju zauvijek zarobljena u tretmanu nastalih prekida, a operativni sistem više nema mogućnost da datim jezgrima dodjeli neki proces na izvršenje. Nastalo stanje označava se kao *zastoj* (deadlock). Da bi se izbjegli zastoji, sukcesivne operacije zaključavanja brava na više lokacija u kodu moraju se svaki put obavljati u istom redoslijedu.

Ukoliko se koristi implementacija brave lock opisana u prethodnoj sekciji, zastoji su mogući i zbog izvršavanja koda na samo jednoj jezgri. Ilustrirat ćemo ovaj problem putem slijedećeg koda:

```
struct lock A;

void f1() {
    acquire(&A);
    f2(); ①
    release(&A);
}

void f_isr() {
    acquire(&A); ②
    // servisiranje prekida
    release(&A);
}
```

Prepostavimo da je neki proces izazvao sistemski poziv za čiji tretman jezgro koje je izvršavalo proces treba da izvrši funkciju f1. Tokom tretmana sistemskih poziva, XV6 kernel ostavlja uključene prekide na jezgri koja adresira prekid. Neka je u toku tretmana sis-

simultanovremeno nastaju i u svog prekida poziva se prekida poziva funkcije kada prvo jezgro ulazi u izvršavanjem linije ②, dok isto vrijeme u drugom jezgru se izvršava funkcija f1. Od tog trenutka do kada jezgrima dodjeli se kao zastoj (deaktivacija zaključavanja) obavljati u istom vremenu.

Opisana u prethodnom koda na samo jednoj redoslijed koda:

simultani poziv za čiji je izvrši funkciju f1. Ostavlja uključene toku tretmana sistema

Da bi se izbjegao ovaj problem, XV6 koristi slijedeću modificiranu implementaciju spinlock brave:

```
struct lock {  
    uint locked;  
};  
  
void acquire(struct lock *lk) {  
    pushcli(); ①  
    while(xchg(&lk->locked, 1) != 0)  
        ;  
}  
  
void release(struct lock *lk) {  
    xchg(&lk->locked, 0);  
    popcli(); ②  
}
```

Za razliku od prethodne varijante, dodata se u nove linije koda na lokacijama ① i ②. Zadatak linije ① je da onemogući prekid na jezgru koji pozove funkciju `acquire` prilikom zaključavanja brave, dok linija ② ima zadatku da omogući prekide prilikom otključavanja brave. Na ovaj način, dok izvršava kod neke kritične sekcije, jezgro ima maskirane prekide.

Funkcije `popcli` i `pushcli` imaju slijedeće implementacije:

```
void pushcli(void) {  
    int eflags;  
    eflags = readeflags();  
    cli(); ①  
    if(cpu->ncli++ == 0) ②  
        cpu->intena = eflags & FL_IF; ③  
}
```

```

void popcli(void) {
    if(--cpu->ncli < 0) ④
        panic("popcli");
    if(cpu->ncli == 0 && cpu->intena) ⑤
        sti(); ⑥
}

```

Maskiranje i demaskiranje prekida konkretno se odvija se u linijama ① i ⑥. Maskiranje prekida je bezuslovno, dok za demaskiranje prekida postoji uslov uspostavljen linijom ⑤. Ovo se čini radi situacija u kojim tretman nekog prekida zahtijeva zaključavanje više brava. Za ove situacije, hardverski prekidi moraju biti maskirani sve dok se ne otključa zadnja zaključana brava. Zbog toga, svako jezgro u svojoj strukturi `cpu` ima polje `ncli` koje predstavlja brojač brava koje u datom trenutku to jezgro drži u zaključanom stanju. Funkcija `pushcli` inkrementira ovaj brojač u liniji ②, dok ga funkcija `popcli` dekrementira u liniji ④. Prilikom zaključavanja prve brave, u polju `intena` strukture `cpu`, bilježi se prvobitno stanje IF polja registra `EFLAGS`. Ovo polje indicira da li su prekidi na datom jezgru u tom trenutku maskirani. Da bi se putem `popcli` prekidi demaskirali, potreban uslov je da se funkcija poziva u kontekstu otključavanja zadnje brave (`cpu->ncli == 0`) i da prekidi pri zaključavanju prve brave nisu bili u maskiranom stanju (`cpu->intena`).

Modifikovane varijante funkcija `acquire` i `release` koje koristi XV6, pored opisanog zastoja, sprečavaju i eventualna stanja utrke koja mogu biti prouzrokovana nastankom hardverskog prekida na jezgru koje tretira neki sistemski poziv.

Svaki dijeljeni resurs u XV6 kernelu štiti sa pojedinačnom bravom koja se u procesu pokretanja operativnog sistema inicijalizira u otključano stanje.

Osnovna  
unutar k  
nja uklju  
aplikaci  
vati odre

Apstr  
za aplika  
broj pro  
sursima  
đaji. Zad  
pleksira  
čitim pr  
glavlju p  
koristi za

## Struktura

XV6 ker

```
// preuzeto iz xv6

struct pde_t {
    uint32_t valid;
    pde_t* parent;
    char* name;
    enum { PDE_TYPE_DIRECTORY = 0, PDE_TYPE_FILE = 1 };
    int prot;
    struct file* file;
    struct file* child;
    void* offset;
    int kbytes;
}
```

# Procesi

Osnovna funkcija operativnih sistema je obezbijedivanje okruženja unutar kojih se mogu izvršavati korisničke aplikacije. Ova okruženja uključuju separatne virtualne adresne prostore za pojedinačne aplikacije i infrastrukturu putem koje svaka aplikacija može zahtijevati određene usluge koje implementira kernel.

Apstrakcija putem koje operativni sistemi modeliraju okruženja za aplikacije naziva se *proces*. U određenom trenutku proizvoljan broj procesa je aktivan u sistemu, a procesi se natječu za pristup resursima kao što su memorija, procesorska jezgra i hardverski uređaji. Zadatak operativnog sistema je da zahtjeve za resursima multiplexira u vremenu, tj. da određeni resurs daje na raspolaganje različitim procesima u različitim vremenskim intervalima. U ovom poglavljtu predstaviti ćemo programske strukture i funkcije koje XV6 koristi za podršku procesima.

## Struktura proc

XV6 kernel implementira procese putem slijedeće strukture:

```
// preuzeto iz fajla proc.h

struct proc {
    uint sz;          ①
    pde_t* pgdir;    ②
    char *kstack;    ③
    enum procstate state; ④
    int pid;         ⑤
    struct proc *parent; ⑥
    struct trapframe *tf; ⑦
    struct context *context; ⑧
    void *chan;       ⑨
    int killed;      ⑩
};
```

Struktura proc

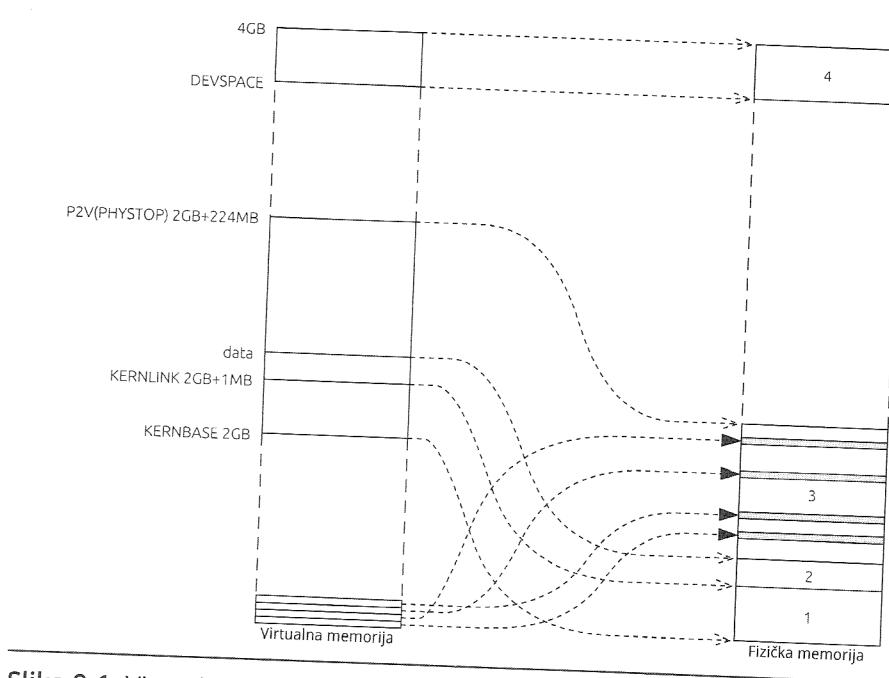
```

struct file *ofile[NFILE]; ⑪
struct inode *cwd; ⑫
char name[16]; ⑬
};

```

Aplikacija koja se izvršava u sistemu reprezentirana je jednom strukturu tipa `proc`. U polju `proc` globalne varijable `ptable` XV6 drži niz struktura tipa `proc`, koji reprezentira tabelu sa svim procesima u sistemu. Broj elemenata u tabeli je fiksan (`NPROC → 64`) i definiran u procesu kompajliranja. Kada neko jezgro izvršava određeni proces, globalni pointer `proc` pokazuje na element unutar tabele koji reprezentira dati proces.

Svaki proces ima svoj virtualni adresni prostor, a polje `pgdir` definirano u liniji ②, predstavlja pointer na PD tabelu tog adresnog prostora. Na slici 9-1 je prikazan virtualni adresni prostor XV6 procesa.



Slika 9-1. Virtualni adresni prostor procesa

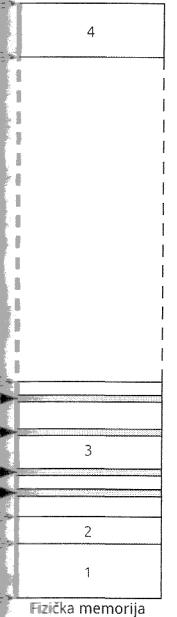
Iznad adrese `KERNBASE` virtualnog adresnog prostora stranice su mapirane u fizički adresni prostor na identičan način kao kod kernel

virtualno  
modu pri  
ko da je  
ovaj pros  
je kreiran  
sve XV6  
nje od ad  
nog prost  
okvire ko  
likom kre  
adresnog  
od trenutl  
lje `sz` defin  
koristi dat

Svako j  
privilegija  
figuraciji K  
stek koji j  
phodni pr  
alokacijom  
koje je def  
stek koji se  
kada se taj  
za određen  
vršenje, vr  
za tretman  
sa. Kada na  
matu trapf  
finirano u  
vanje proce  
uključuje p  
cesa na pre  
vršavati na  
u nekom bu

ana je jednom  
e ptable XV6  
sa svim proce-  
 $C \rightarrow 64$ ) i defi-  
sava određeni  
utar tabele ko-

olje pgdir de-  
tog adresnog  
stor XV6 pro-



ra stranice su  
ao kod kernel

avljie 9, Procesi

virtualnog prostora i dostupne su samo dok je procesor u najvišem modu privilegija. Adresni prostora ispod KERNBASE konfigurira se tako da je dostupan dok je procesor u najnižem nivou privilegija. U ovaj prostor učitava se sadržaj ELF datoteke od aplikacije za koju je kreiran proces. Učitavanje se vrši od nulte adrese, obzirom da se sve XV6 aplikacije kompajliraju tako da njihova .text sekcija počinje od adrese 0. Kao što je prikazano na slici, dio virtualnog adresnog prostora koji je zauzet sadržajem ELF fajla mapira se u fizičke okvire koje kernel alocira po potrebi putem alokatora stranica. Prilikom kreiranja procesa, XV6 alocira zadnju stranicu u ovom dijelu adresnog prostora sa namjenom da se stranica koristi kao stek, i to od trenutka kada se aplikacija počne izvršavati na nekoj jezgri. Pojme sz definirano u liniji ❶ indicira količinu memorije u bajtima koju koristi dati proces.

Svako jezgro izvršava procesa sa omogućenim prekidima, a nivo privilegija koje jezgro ima u tom trenutku je najniži. U ovakvoj konfiguraciji kada nastane prekid na jezgru vrši se transfer na poseban stek koji je konfiguriran putem TSS deskriptora tog jezgra. Neophodni prostor za stek XV6 obezbijeđuje u toku kreiranja procesa alokacijom posebne stranice čija se adresa zapisuje u polju kstack koje je definirano u liniji ❸. Dakle, proces dobija poseban kernel stek koji se koristi prilikom tretmana prekida nastalih u trenutku kada se taj proces izvršava na nekom jezgru. Svaki put kada kernel za određeni vremenski interval nekom jezgru dodjeli proces na izvršenje, vrši se modifikacija TSS segmenta datog jezgra kako bi se za tretman prekida na tom jezgru koristio kernel stek od tog procesa. Kada nastane prekid, stanje prekinutog procesa snima se u formatu trapframe strukture na kernel steku tog procesa, a polje tf definirano u liniji ❷ pokazuje na datu strukturu. Ukoliko je izvršavanje procesa prekinuto uslijed djelovanja tajmera, tretman prekida uključuje promjenu konteksta kojom prestaje izvršavanje tog procesa na prekinutom jezgru, a kernel bira novi proces koji će se izvršavati na datom jezgru. Da bi se prekinuti proces mogao nastaviti u nekom budućem trenutku, prilikom promjene konteksta na ker-

nel steku vrši se snimanje stanja prezerviranih registara procesora u formatu strukture tipa `context`. Polje `context`, definirano u liniji ❸, predstavlja pointer na lokaciju na kojoj je snimljena ova struktura.

Postoji samo jedan način da se na XV6 operativnom sistemu kreira novi procesa, a to je da proces koji je u izvršenju pozove sistemski poziv `fork`. Prilikom tretmana ovog sistemskog poziva sadržaj `proc` strukture procesa koji je obavio sistemski poziv kopira se u `proc` strukturu novog procesa. Kreirani proces efektivno je klon već postojećeg procesa, a između procesa se formira relacija u kojoj proces koji izvršio operaciju `fork` predstavlja roditelja za novoformirani proces. Svaki proces ima pointer na `proc` strukturu svog roditelja putem polja `parent` koje je definirano u liniji ❹. Kada se kreira novi proces, kernel mu dodjeljuje unikatni identifikator koji se zapisuje u polje `pid` iz linije ❺.

Kernel vodi računa o svim fajlovima koje u nekom trenutku proces drži otvorenim putem niza `ofile` definiranim u liniji ❻. Maksimalan broj otvorenih fajlova ograničen je u procesu kompajliranja na vrijednost `NOFILE` (16). Proses se izvršava u nekom trenutnom direktoriju koji je određen poljem `cwd` iz linije ❼. Polje `name`, definirano u liniji ➋, predstavlja ime procesa koje se primarno koristi u procesu debagiranja.

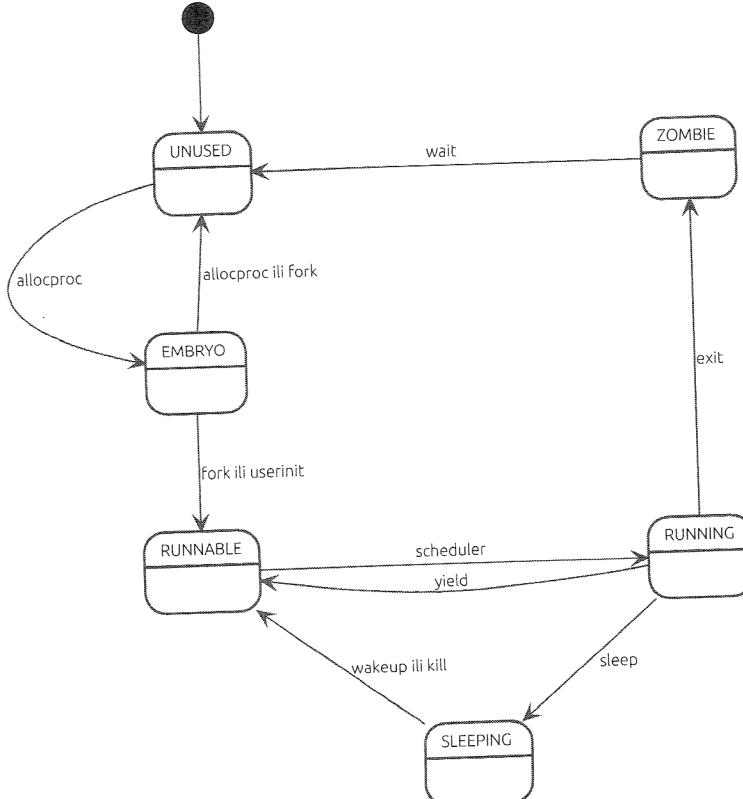
Tokom izvršavanja proces mijenja stanja u skladu sa dijagramom prikazanim na slici 9-2. Trenutno stanje procesa bilježi se u polju `state` koje je definirano u liniji ❾. Svi procesi u tabeli `pTable` inicijalno su u `UNUSED` stanju koje označava da struktura `proc` nema asocirani aplikaciju za izvršavanje. Kada se kreira novi proces, kernel u tabeli `pTable` pronađe element koji je u stanju `UNUSED`. Dok traje priprema njegovog adresnog prostora, odabrani proces je u stanju `EMBRYO`. U slučaju bilo kakve greške tokom pripreme, proces se vraća u `UNUSED` stanje.

ra procesora u  
jano u liniji ⑧,  
va struktura.

n sistemу kre-  
ju pozove sis-  
og poziva sadr-  
ziv kopira se u  
vno je klon već  
ija u kojoj pro-  
a novoformira-  
u svog roditelja  
a se kreira novi  
oji se zapisuje u

n trenutku pro-  
liniji ⑪. Maksi-  
u kompajliranja  
kom trenutnom  
olje name, defini-  
marno koristi u

sa dijagramom  
iližezi se u polju  
eli ptable inici-  
a proc nema aso-



Slika 9-2. Dijagram stanja procesa

Kada je proces u potpunosti spreman za izvršavanje prelazi u RUNNABLE stanje u kojem konkuriše na procesorski resurs.

Prilikom promjene konteksta, proces odabran za izvršavanje mijenja stanje iz RUNNABLE u RUNNING. Broj procesa koji u nekom trenutku mogu biti u ovom stanju određen je brojem procesorskih jezgri. Procesi koji su u RUNNING stanju troše procesorski resurs izvršavajući likacija, a u toku tretmana prekida izvršavajući kernel

ku u stanju RUNNABLE. Na ovaj način, procesi mogu biti prekinuti u izvršavanju na jednom procesorskom jezgru, a nastavljeni sa izvršavanjem u nekom drugom trenutku na bilo kojem od raspoloživih procesorskih jezgri.

Tokom izvršavanja procesa mogu nastati situacije u kojim nastavak izvršavanja određenog procesa nema smisla dok se ne ispuni neki uslov, npr. čekanje na paket sa mreže ili unos sa tastature. Ukoliko bi ostao u RUNNING ili RUNNABLE stanju, proces bi bespotrebno trošio procesorske resurse čekajući na ispunjenje uslova za nastavak izvršavanja. Zbog toga, proces se prebacuje u SLEEPING stanje i to na kanalu određenom putem polja `chan` iz linije ❾. Dok je u ovom stanju proces ne konkuriše na procesorske resurse. Kada se ispuni uslov za dalji nastavak izvršavanja procesa, njegovo stanje vraća se u RUNNABLE.

Ukoliko nastane iznimka u izvršavanju koda aplikacije ili ukoliko dođe do kraja programa, proces prelazi u stanje ZOMBIE u kojem čeka na dealokaciju svih resursa koje je koristio. Oslobođanje resursa vrši roditelj od datog procesa, ili proces `init` za slučaj da je roditelj proces već ranije terminiran. Nakon dealokacije korištenih resursa, proces se vraća u UNUSED stanje. Proses se markira za terminaciju aktiviranjem polja `killed` iz linije ❿.

## Implementacija sistemskih poziva

Korisničke aplikacije koje se izvršavaju u procesima imaju potrebu da komuniciraju sa hardverskim uređajima. Čak i elementarna aplikacija koja na ekranu ispisuje neki tekst zahtijeva komunikaciju sa kontrolerima video kartice i tastature. Obzirom da registri ovih kontrolera nisu dostupni u virtualnom adresnom prostoru procesa i da procesi nemaju dozvole da koriste instrukcije `in` i `out`, jedini način da aplikacije komuniciraju sa vanjskim uređajima je putem posredovanja kernela. Iako je sav sadržaj kernela mapiran u adresnom prostoru svakog procesa iznad adrese KERNBASE, aplikacija ne mo-

že jednostavno izvršiti poziv neke kernel funkcije putem instrukcije `call`, obzirom da su sve stranice iznad KERNBASE dostupne samo kada je procesor u najvišem nivou privilegija.

Putem mehanizma sistemskog poziva kernel omogućava da aplikacija pozove odabrane funkcije implementirane u kernelu putem kojih se dio funkcionalnosti iz kernela kontrolirano eksportuje na korištenje aplikacijama. Da bi pokrenula ovaj mehanizam, aplikacija pokreće instrukciju:

```
int T_SYSCALL
```

gdje `T_SYSCALL` ima vrijednost 64.

Prije nego što instrukcijom `int` izvrši sistemski poziv, aplikacija mora odabratи коју од eksportovanih функција у kernelu ће да покrene путем системског poziva. Izbor se vrši postavljanjem odgovarajućег broja у registru `eax`. Putem globalnog niza `syscalls`, kernel asocira бројеве системских poziva са функцијама које се путем instrukcije `int` извозе на кориштење корисниčким aplikacijama:

```
// preuzeto iz fajla syscall.c

static int (*syscalls[])(void) = {
[1] sys_fork,
[2] sys_exit,
[3] sys_wait,
[4] sys_pipe,
[5] sys_read,
[6] sys_kill,
[7] sys_exec,
[8] sys_fstat,
[9] sys_chdir,
[10] sys_dup,
[11] sys_getpid,
[12] sys_sbrk,
[13] sys_sleep,
[14] sys_uptime,
[15] sys_open,
[16] sys_write,
[17] sys_mknod,
[18] sys_unlink,
[19] sys_link,
[20] sys_mkdir,
```

```
[21] sys_close,  
};
```

Niz ima 21 element, a pojedinačni elementi su pointeri na kernel funkcije koje ne uzimaju nikakve parametre i vraćaju cjelobrojnu vrijednost. Konkretnе argumente korištene prilikom sistemskog poziva, pozvane kernel funkcije će preuzimati sa korisničkog steka procesa koji je izvršio sistemski poziv. Dakle, pored izbora funkcije koja će biti pozvana u kernelu, aplikacija prije nego što izvrši sistemski poziv na svom steku treba postaviti i argumente za kernel funkciju koju poziva.

Kada se pokrene instrukcija `int`, nakon tranzicije u kernel i snimanja stanja procesa koji je izvršio ovu instrukciju, počinje izvršavanje funkcije `trap` koja radi tretmana nastalog prekida poziva funkciju `syscall` definiranu na slijedeći način.

```
// preuzeto iz fajla syscall.c  
  
void syscall(void)  
{  
    int num;  
    num = proc->tf->eax; ❶  
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {  
        proc->tf->eax = syscalls[num](); ❷  
    } else {  
        sprintf("%d %s: unknown sys call %d\n",  
                proc->pid, proc->name, num);  
        proc->tf->eax = -1; ❸  
    }  
}
```

U liniji ❶ čita se stanje polja `eax` iz `trapframe` strukture prekinutog procesa. Pročitani broj `num` predstavlja proslijedjeni broj sistemskog poziva. Ukoliko je broj sistemskog poziva validan, u liniji ❷ poziva se željena kernel funkcija dereferenciranjem odabranog pointer na funkciju iz niza `syscalls`. Rezultat poziva snima se u `trapframe` strukturu procesa, tako da se u registru `eax` nađe rezultat sistemskog poziva kada se nastavi izvršavanje aplikacije nakon tretmana prekida. U slučaju da broj prekida nije validan, povratna vrijednost sis-

temskog po-  
nju sistemsk

Najjedno-  
aplikaciji un-  
šava kod od-  
ovaj sistem

```
// preuzeto  
  
int sys_get  
{  
    return pro
```

Funkcija  
procesa koji

Da bi za k-  
obične funk-  
objektnim fa-  
nalaze se fun-  
temski poziv  
sistemske po-  
za pozivanje  
Ovim je svak  
temskog poz-  
obične funk-  
funkcija:

```
int getpid(
```

Koja ima s

```
// preuzeto  
  
.globl getpid:  
getpid:  
    movl $11,  
    int $64;  
    ret ❸
```

temskog poziva je -1 čime se procesu signalizira greška u izvršavanju sistemskog poziva.

Najjednostavniji sistemski poziv pod rednim brojem 11, vraća aplikaciji unikatni identifikator procesa koji u datom trenutku izvršava kod od aplikacija. Kernel funkcija `sys_getpid` koja se poziva za ovaj sistemski poziv implementirana je na slijedeći način:

```
// preuzeto iz fajla sysproc.c

int sys_getpid(void)
{
    return proc->pid;
}
```

Funkcija na mjestu poziva vraća vrijednost polja `pid` iz strukture procesa koji se trenutno izvršava na datom jezgru.

Da bi za korisničke aplikacije sistemski poziv izgledao kao poziv obične funkcije, svaka XV6 korisnička aplikacija uvezuje se sa objektnim fajlom nastalim asembleriranjem `usys.S`. Unutar ovog fajla nalaze se funkcije, implementirane u asembleru, koje izvršavaju sistemski poziv neposredno nakon postavljanja odgovarajućeg broja sistemskog poziva u registar `eax`. U fajlu `user.h` nalaze se deklaracije za pozivanje svake pojedinačne funkcije implementirane u `usys.S`. Ovim je svaka kernel funkcija koja je izvezena kroz mehanizam sistemskog poziva dobila svoj ekvivalent na strani aplikacije u obliku obične funkcije. Na primjer, u fajlu `user.h` deklarirana je slijedeća funkcija:

```
int getpid(void);
```

Koja ima slijedeću implementaciju:

```
// preuzeto iz fajla usys.S

.globl getpid;
getpid:
    movl $11, %eax; ①
    int $64; ②
    ret ③
```

Obzirom da se sve XV6 aplikacije uvezuju sa ELF fajlom `usys.o`, funkcija `getpid` se može pozvati u bilo kojoj aplikaciji nakon uključivanja zaglavlja `user.h`. Kada neka aplikacija izvrši poziv ove funkcije, u liniji ❶ postavlja se broj sistemskog poziva na 11. Zatim se generira prekid sa vektorom 64, što za kernel predstavlja sistemski poziv.

Prelaskom u kernel izvršava se slijedeća sekvenca funkcija:

- ISR → `alltraps` → `trap` → `syscall` → `sys_getpid`

Po povratku u funkciju `alltraps`, nakon postavljanja vrijednosti registara procesora pročitanih iz `trapframe` strukture, izvršava se instrukcija `iret` kojom prestaje izvršavanje kernela, a izvršavanje procesa nastavlja se u liniji ❸. U tom trenutku, u registru `eax` nalazi se povratna vrijednost sistemskog poziva. Ovo će ujedno biti i povratna vrijednost funkcije `getpid` nakon izvršetka linije ❷. Dakle, pozivom obične funkcije `getpid`, implicitno je pozvana funkcionalnost koju obezbeđuje kernel u funkciji `sys_getpid`. Na sličan način se implementiraju sve funkcije iz C standardne biblioteke koje zahtijevaju komunikaciju sa vanjskim uređajima.

Ukoliko sistemski poziv uzima argumente, aplikacija ih ostavlja na korisničkom steku njenog procesa. Prilikom tretmana sistemskog poziva, kernel ima pristup svim dijelovima adresnog prostora procesa, uključujući i korisnički stek. Prije upotrebe argumenata kernel mora provjeriti validnost istih jer eventualna iznimka prilikom tretmana prekida, npr. zbog pristupa nevažećoj stranici, može izazvati zaustavljanje cijelog operativnog sistema. Za ovu namjenu XV6 koristi veliki broj funkcija i to spram tipa argumenta koji se pruzimaju sa steka. Kao primjer analizirat ćemo funkciju `argint` definiranu na slijedeći način:

```
// preuzeto iz fajla syscall.c
int fetchint(uint addr, int *ip)
{
```

```

if(addr >= proc->sz || addr+4 > proc->sz) ①
    return -1; ②
*ip = *(int*)(addr); ③
return 0; ④
}

int argint(int n, int *ip) ⑤
{
    return fetchint(proc->tf->esp + 4 + 4*n, ip); ⑥
}

```

Funkcija `argint`, definirana u liniji ⑤, koristi za preuzimanje cje-lobrojne vrijednosti koja je kao `n`-ti argument proslijedena sistemskom pozivu na korisničkom steku. Pročitanu vrijedost funkcija upisuje na lokaciju u memoriji određenu drugim argumentom, posredstvom `ip`. Ukoliko je operacija uspješna funkcija vraća 0, u suprotnom vraća -1.

Kompletan posao provjere validnosti obavlja se u funkciji `fetchint` koju `argint` poziva u liniji ⑥. Prvi argument poziva je adresa sa koje treba preuzeti vrijednost izračunata na osnovu adrese dna steka prekinutog procesa `proc->tf->esp` i broja proslijedenog parametra `n`. Drugi argument je lokacija u kernel memoriji gdje se treba upisati eventualno pročitana vrijednost sa steka procesa. Provjera validnosti proslijedene adrese vrši se u liniji ①. Ako adresa nije ispravna, odmah se prekida sa dalnjim izvršavanjem funkcije, te se u liniji ② signalizira greška putem negativne povratne vrijednosti funkcije. Sa druge strane, ukoliko je adresa validna, vrijednost pročitana sa te adrese u liniji ③ kopira se u kernel memoriju na lokaciju `ip`, a funkcija signalizira uspjeh vraćajući vrijednost 0 u liniji ④.

## Kreiranje procesa

Kao što je već naglašeno, svi procesi nastaju izvršavanjem sistemskog poziva `fork`. Neophodan preduslov za kreiranje novog procesa je postojanje nekog drugog procesa koji će izvršiti sistemski poziv `fork` i koji će za novi proces biti roditelj. Kreirani proces je egzaktna kopija svog roditelja, a oba procesa nastavljaju izvršavanje na istoj lokaciji u kodu neposredno nakon poziva `fork`. Jedina razlika izme-

ajlom usys.o,  
nakon uklju-  
ziv ove funk-  
11. Zatim se  
sistemski

nkacija:

a vrijednosti  
izvršava se  
izvršavanje  
u eax nalazi  
mo biti i po-  
Dakle, po-  
kacionalnost  
an način se  
toje zahtije-

ih ostavlja  
na sistem-  
og prostora  
rgumenata  
imka prili-  
nici, može  
u namjenu  
ta koji se  
argint de-

đu procesa je u vrijednosti rezultata sistemskog poziva koju kernel ostavlja u registru `eax`, koja je za kreirani proces 0, a za roditelj proces predstavlja identifikacioni broj novog procesa.

Slijedeća kernel funkcija implementira sistemski poziv `fork`:

```
// preuzeto iz fajla proc.c

int fork(void)
{
    int i, pid;
    struct proc *np; ①

    if((np = allocproc()) == 0) ②
        return -1; ③

    if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){ ④
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1; ⑥
    }
    np->sz = proc->sz; ⑦
    np->parent = proc; ⑧
    *np->tf = *proc->tf; ⑨

    np->tf->eax = 0; ⑩

    for(i = 0; i < NOFILE; i++) ⑪
        if(proc->ofile[i])
            np->ofile[i] = filedup(proc->ofile[i]);
    np->cwd = idup(proc->cwd); ⑫

    safestrncpy(np->name, proc->name, sizeof(proc->name)); ⑬

    pid = np->pid; ⑭

    acquire(&pstable.lock);
    np->state = RUNNABLE; ⑮
    release(&pstable.lock);

    return pid; ⑯
}
```

Funkcija ne uzima nikakve argumente, a vraća rezultat sistemskog poziva. Sastavni dio izvršenja funkcije je kreacija novog procesa reprezentiranog strukturuom `proc` na koju pokazuje pointer definiran u liniji ①. Inicijalizacija ove strukture događa se putem funkcije `allocproc` u liniji ②, dok se u liniji ④ putem funkcije `copyuvm` vr-

ši kreacija virtuelne memorije kopiranjem koda iz novog procesa u liniju ⑤ terminira u liniju ⑥ povratka iz funkcije `fork`.

Funkcija u liniji ⑦ zauzima novi prostor u memorijskoj strukturi roditelja. Kod u liniji ⑧ kopira `trapframe` strukturu, čime su dva procesa identični. Različit je samo identitet vrijednosti sisteških varijabli.

Novi proces dobija identitet od ⑩ do ⑫, čime je i novi identitet direktorij i imena u liniji ⑬.

Izvršavanje novog procesa nakon čega bilo je moguće da novi proces obavlja operacije na vlastitom računalu, ali je to u ovom slučaju obavešteno eventualno.

Sistemski poziv `fork` za roditelj procesa vrati novi identitet.

Nakon istog poziva, kod istog programu se trenutno izvršava skog poziva. Dostupnost novog procesa u adresnom prostoru roditelja. Ukoliko novi proces dobije šansu da se izvrši, onda će se u njemu početi izvršavati.

ova koju kernel  
za roditelj pro-

poziv fork:

④

⑯

ezultat sistem-  
novog proce-  
pointer defi-  
putem funk-  
copyuvn vr-

avljje 9, Procesi

ši kreacija virtualnog adresnog prostora novog procesa, koji nastaje kopiranjem kompletног sadržaja adresnog prostora roditelja definiranog putem `proc->pgdir`. U slučaju bilo kakve greške, funkcija se terminira u linijama ⑩ ili ⑪ vraćajući negativnu vrijednost, a prije povratka iz funkcije putem linije ⑫ vrši se dealokacija resursa novog procesa.

Funkcija u liniji ⑦ postavlja informaciju o količini memorije koju zauzima novi proces, dok u liniji ⑧ novi proces eksplisitno dobija roditelja. Kompletно stanje roditelja u liniji ⑨ kopira se putem `trapframe` strukture u novi proces, nakon čega dva procesa postaju identični. Razlika između procesa nastaje u liniji ⑩, gdje se povratna vrijednost sistemskog poziva za novi proces postavlja na 0.

Novi proces dobija kopije fajl deskriptora od roditelja u linijama od ⑪ do ⑫, čime će oba procesa imati iste otvorene fajlove. Trenutni direktorij i ime novog procesa kopiraju se iz roditelja u linijama ⑬ i ⑭.

Izvršavanjem linije ⑮, novi proces postavlja se u RUNNABLE stanje, nakon čega bilo koje jezgro odmah može početi izvršavati kod novog procesa. Obzirom da se mijenja stanje elementa u `ptable` nizu, operacija se obavlja pod kontrolom brave `ptable.lock` da bi se izbjegla eventualna stanja utrke.

Sistemski poziv završava se u liniji ⑯ vraćanjem rezultata poziva za roditelj proces koji je u liniji ⑭ pročitan iz polja `pid` strukture `proc` dijete procesa.

Nakon sistemskog poziva `fork`, dijete i roditelj proces izvršavaju kod istog programa. Program može identificirati proces u kojem se trenutno izvršava, analizirajući povratnu vrijednost od sistemskog poziva. Dijete proces inicijalno ima sve varijable u sopstvenom adresnom prostoru kopirane iz adresnog prostora roditelja i čim dobije šansu za izvršavanje može da ih mijenja neovisno od roditelja. Ukoliko dijete proces treba da izvršava kod neke druge aplikacije, mora se izvršiti dodatna migracija.

kacije, program se piše tako da dijete proces odmah po obavljenoj operaciji `fork` pozove sistemski poziv `exec` čime se adresni prostor djeteta u potpunosti mijenja učitavanjem sadržaja nekog programa, čije ime se daje kao argument sistemskom pozivu `exec`. Obavljanjem sistemskog poziva `wait`, roditelj proces ima mogućnost da bez trošenja procesorskih resursa sačeka na završetak izvršavanja dijete procesa. Povratna vrijednost sistemskog poziva `wait` je identifikator dijete procesa koji je prestao sa izvršenjem.

Slijedeći segment programa demonstrira upotrebu sistemskih poziva `fork` i `wait`:

```
char *argv[] = { "prog", 0 }; ①  
if (fork()) ②  
    wait(); ③  
else  
    exec("/bin/prog", argv) ④
```

Definiraju se argumenti za funkciju `main` programa koji će se

- ① putem `exec` učitati u dijete proces. Prvi argument po konvenciji je ime ELF fajla programa koji se učitava.
- ② Obavlja se `fork` poziv i vrši grnanje na osnovu dobivene povratne vrijednosti.

Ako je povratna vrijednost veća od nule, program se izvršava u roditelj procesu koji poziva sistemski poziv `wait` i čeka na završetak dijete procesa. Kada se probudi nakon operacije

- ③ `wait`, roditelj nastavlja izvršavati kod programa nakon linije ④. Sastavni dio operacije buđenja procesa nakon sistemskog poziva `wait` je i dealokacija resursa jednog dijete procesa koji je u stanju ZOMBIE.

Ako je povratna vrijednost sistemskog poziva 0, kod se izvršava u dijete procesu, te se učitava program iz ELF datoteke

- ④ `prog`, koja se nalazi u `bin` direktoriju. Izvršavanje učitanog programa počinje od funkcije `main`, koja će iznad svog aktiva-

cijskog okvira dobiti argumente date u skladu sa `argv`. Dijete proces više neće imati šansu da nastavi izvršavanje prvobitnog programa.

Ukoliko bi se izbacio poziv `wait` u liniji ③, roditelj bi nastavio izvršavati prvobitni program dok dijete paralelno izvršava kod programa `prog`.

## PROCES INIT

Uspostavljanje novih procesa putem mehanizma `fork` stvara suštinski problem nastanka prvog procesa u operativnog sistemu. Ovaj proces označava se kao `init` proces i ima specijalan tretman od strane kernela. Inicijalni program koji se izvršava u prvom procesu nastaje putem slijedećeg dijela Makefile skripte.

```
initcode: initcode.S
    $(CC) $(CFLAGS) -nostdinc -I. -c initcode.S ①
    $(LD) $(LDFLAGS) -N -e start -Ttext 0 \ ②
        -o initcode.out initcode.o
    $(OBJCOPY) -S -O binary initcode.out initcode ③
```

① Asemblira se fajl `initcode.S`, čime nastaje objektni fajl `initcode.o`

② Kreira se program u ELF fajlu `initcode.out` uvezivanjem objektnog fajla `initcode.o`, na način da `.text` sekcija počinje od adrese 0, i to bez korištenja standardne biblioteke koja se distribuira sa kompjuterom.

③ Program se iz ELF formata konvertuje u binarni format ko-piranjem sadržaja svih sekacija fajla `initcode.out` u novi fajl `initcode`

Sadržaj binarne datoteke `initcode`, nastale putem linije ❸, ubacuje se kernel ELF fajl tokom kompajliranja i uvezivanja kernela. Datoteka `initcode.S` asemblerirana u liniji ❶ ima slijedeći sadržaj:

```
.globl start
start:
    pushl $argv
    pushl $init
    pushl $0
    movl $$SYS_exec, %eax
    int $T_SYSCALL ①

exit:
    movl $$SYS_exit, %eax
    int $T_SYSCALL ②
    jmp exit

init: ③
    .string "/init\0"

.p2align 2
argv: ④
    .long init
    .long 0
```

Linije asembler koda gornjeg programa do linije ❶ imaju svoj ekvivalent u jednoj liniji C koda:

```
exec("/init", argv);
```

Gdje je prvi argument za sistemski poziv exec definiran u liniji ❸, a drugi u liniji ❹.

Ukoliko uspije sistemski poziv exec, proces počinje izvršavati novi program učitan iz ELF datoteke `init` koja se unutar fajl sistema nalazi u direktoriju `/`. U suprotnom, program u liniji `2` izvršava sistemski poziv `exit`, čime se terminira trenutni proces.

Nakon uspješnog poziva exec, proces init počinje izvršavati slijedeći program:

```
// preuzeto iz fajla init.c  
#include "types.h"  
#include "stat.h"  
#include "user.h"
```

```

#include "fcntl.h"

char *argv[] = { "sh", 0 };

int main(void)
{
    int pid, wpid;

    if(open("console", O_RDWR) < 0){ ❶
        mknod("console", 1, 1); ❷
        open("console", O_RDWR); ❸
    }
    dup(0); ❹
    dup(0); ❺

    for(;;){
        printf(1, "init: starting sh\n");
        pid = fork(); ❻
        if(pid < 0){
            printf(1, "init: fork failed\n");
            exit(); ❼
        }
        if(pid == 0){
            exec("sh", argv); ❽
            printf(1, "init: exec sh failed\n");
            exit();
        }
        while((wpid=wait()) >= 0 && wpid != pid) ❾
            printf(1, "zombie!\n");
    }
}

```

Program u liniji ❶ iz fajl sistema pokušava otvoriti fajl /console. Ukoliko ne postoji ovaj fajl, program ga prvo kreira u liniji ❷, a zatim i otvara u liniji ❸. Obzirom da se prilikom kreiranja koristi sistemski poziv mknod, kreira se uređaj čiji je tip određen argumentima za sistemski poziv, a korišteni argumenti impliciraju kreiranje konzole. Proces konzoli dodjeljuje fajl deskriptor pod rednim brojem 0, koji se zatim u linijama ❹ i ❺ sistemskim pozivom dup kopira u fajl deskriptore pod rednim brojevima 1 i 2. Na XV6 operativnom sistemu fajl deskriptor pod rednim brojem 0 predstavlja standarni ulaz (standard input), deskriptor 1. predstavlja standardni izlaz (standard output), dok se deskriptor 2. koristi kao izlaz za greške (standard error). Ukoliko program čita iz prvog fajl deskriptora, kernel putem konzole preusmjerava zahtijev kontroleru tastature, a ukoliko pro-

linije ❻, ubacujući  
u kernela. Da-  
sadržaj:

maju svoj ek-

miran u liniji ❸

je izvršavati  
fajl sistema  
izvršava sis-

izvršavati sli-

lavije 9, Procesi

gram vrši upisivanje u drugi fajl deskriptor, kernel, preko konzole, podatke preusmjerava na prikaz video kartici.

Nakon linije ⑤, program ulazi u beskonačnu petlju, u kojoj se u liniji ⑥ prvo odrađuje operacija fork, tj. proces init se klonira u novi proces. Za slučaj greške u operaciji fork, proces se odmah terminira u liniji ⑦.

Kreirani proces u liniji ⑧ obavlja sistemski poziv exec čime se u njegov adresni prostor učitava program sh. Ovaj proces ima kopirane fajl deskriptore od init procesa, tj. može da komunicira sa konzolom preko standardnog ulaza i izlaza. Čim pređe u stanje RUNNING, proces počinje komunicirati sa korisnikom na način da sa tastature traži unos neke validne sh komande, koja se zatim i interpretira.

Sa druge strane, proces init nastavlja izvođenje prvobitnog programa te u liniji ⑨ sistemskim pozivom wait prelazi u stanje čekanja na završetak nekog procesa kojem je neposredni roditelj. Korisnički procesi koji u budućnosti nastanu u nasljednoj hijerarhiji na čijem se vrhu nalazi proces sh, potencijalno mogu izgubiti svoje neposredne roditelje. XV6 kernel određuje init proces kao neposrednog roditelja za sve procese koji prije svog završetka izgube roditelje. Operacija promjene roditelja događa se u funkciji exit, koju svi procesi pozivaju neposredno pred kraj izvršavanja.

Ukoliko se u liniji ⑩, zbog završetka nekog dijete procesa koji ne izvršava program sh, probudi init proces, oslobađaju se resursi terminiranog procesa, ispisuje se odgovarajuća poruka u liniji ⑪, pa se init proces ponovo vraća u mod čekanja u liniji ⑨. Međutim, ako je terminirani proces nastao putem operacije fork iz linije ⑥, program init procesa izlazi iz petlje u liniji ⑩, te se vraća u liniju ⑥. Ovim se pokreće novi proces koji nanovo izvršava program sh.

Dakle, izvršavanjem programa init, prvi proces ima ključnu ulogu u dealociranju resursa onih procesa koji izgube neposredne rodi-

eko konzole, u kojoj se u klonira u no- dmah termi- evec čime se u cesima kopira- cira sa kon- stanje RUNNING, da sa tastature interpretira.

bitnog pro- stanje čekanja. Korisnički na čijem se neposredne srednjog rodi- ditelje. Ope- su svi procesi

procesa koji ne se resursi ter- liniji ⑩, pa se vedutim, ako je ⑥, program ⑥. Ovim se

ključnu ulo- posredne rodi-

glavlje 9, Procesi

telje, te u održavanju sistema u stanju da uvijek postoji barem jedan proces koji izvršava program sh. Za kreiranje prvog procesa ne koristi se sistemski poziv fork, već kernel na poseban način vrši premenu njegovog stanja, tako da kad počne njegovo izvršavanje proces ima iluziju da je nastao putem operacije fork.

BSP jezgro u procesu pokretanja operativnog sistema poziva funkciju userinit kojom se kreira prvi proces. Funkcija ima sljedeću implementaciju:

```
// preuzeto iz fajla proc.c

void userinit(void)
{
    struct proc *p;
    extern char _binary_initcode_start[], _binary_initcode_size[]; ①

    p = allocproc(); ②
    initproc = p;
    if((p->pgdir = setupkvm()) == 0) ③
        panic("userinit: out of memory?");
    inituvm(p->pgdir, _binary_initcode_start,
            (int)_binary_initcode_size); ④
    p->sz = PGSIZE; ⑤
    memset(p->tf, 0, sizeof(*p->tf));
    p->tf->cs = (SEG_UCODE << 3) | DPL_USER; ⑥
    p->tf->ds = (SEG_UDATA << 3) | DPL_USER; ⑦
    p->tf->es = p->tf->ds;
    p->tf->ss = p->tf->ds;
    p->tf->eflags = FL_IF; ⑧
    p->tf->esp = PGSIZE; ⑨
    p->tf->eip = 0; ⑩

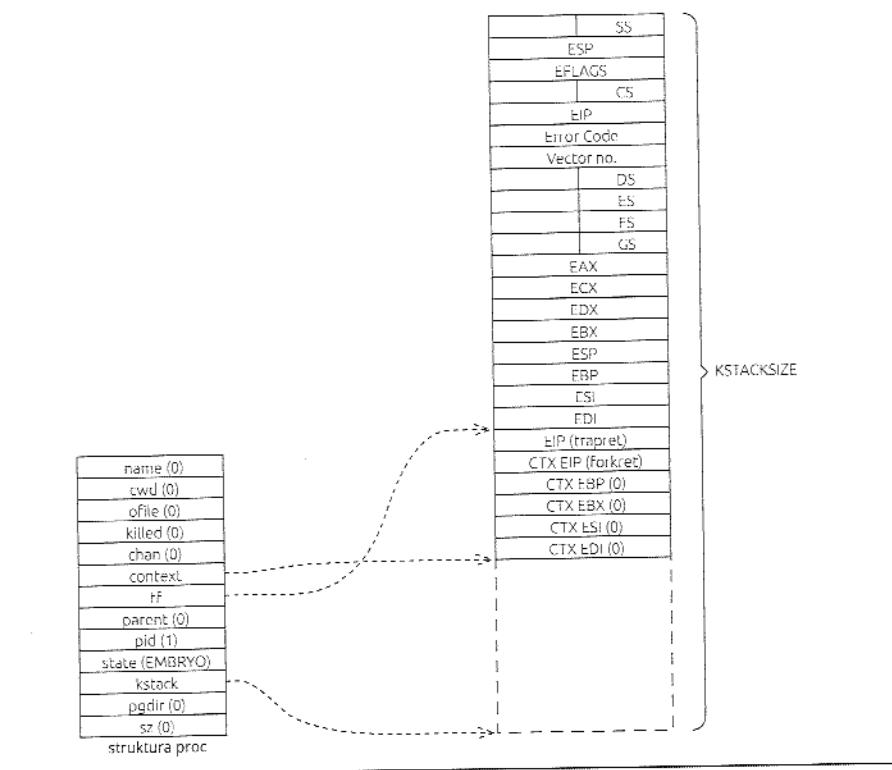
    safestrncpy(p->name, "initcode", sizeof(p->name));
    p->cwd = namei("/");
    p->state = RUNNABLE; ⑪
}
```

Eksterne varijable deklarirane u liniji ① inicijalizirane su od strane linkera tokom uvezivanja kernel ELF datoteke, i to spram lokacije u memoriji od koje počinje sadržaj binarnog fajla initcode i spram veličine tog fajla.

Alokacija strukture proc za init proces obavlja se u liniji ② putem iste funkcije koja se koristi za alokaciju novog procesa u sistem-

Kreirani  
cira se u lin  
ra procesa n  
ra stranica.  
binarnog fa  
početak nul

Između l  
lih polja str  
da se dobija



Slika 9-3. Struktura proc od procesa init nakon izvršenja funkcije allocproc

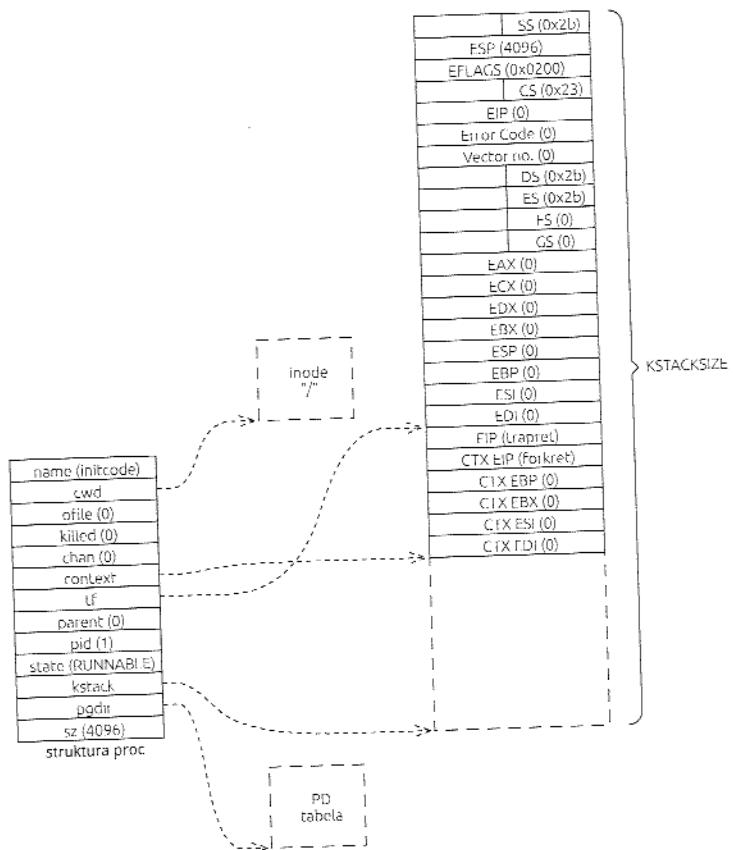
skom pozivu `fork`. Funkcija `allocproc` u tabeli procesa `ptable` pronalazi prvi element u `UNUSED` stanju, a zatim inicijalizira polja unutar odabrane strukture i alocira kernel stek za novi proces. Na alociranom steku procesa, funkcija kreira strukturu `trapframe` čije elemente inicijalizira tako da sa stanovišta procesa izgleda kao da je upravo obavio sistemski poziv `fork`. Inicijalno stanje procesa nakon izvršenja linije ② prikazano je na slici 9-3.

Proces dobija inicijalni virtualni adresni prostor putem funkcije `setupkvm` u liniji ③, čime se formiraju mapiranja raspoloživih stranica iznad adrese `KERNBASE` identično kao u kernel virtualnom adresnom prostoru.

Slika 9-4.  
rinit

Kreirani adresni prostor procesa putem funkcije `inituvm` modifika se u liniji ④ na način da se prvo nulta stranica adresnog prostora procesa mapira u okvir fizičke memorije alociran putem alokatora stranica. Nakon mapiranja, vrši se kopiranje kompletног sadržaja binarnog fajla `initcode` sa lokacije određene putem linije ① na sami početak nulte stranice adresnog prostora procesa.

Između linija ⑤ i ⑥, funkcija `inituvm` nastavlja inicijalizaciju ostalih polja strukture `proc` i modifikaciju stanja procesa na steku, tako da se dobija konačno stanje procesa prikazano na slici 9-4.



Slika 9-4. Struktura `proc` od procesa `init` nakon izvršenja funkcije `use_rinit`

Alocirani proces spreman je za izvršavanja nakon linije ⑪. Vrijednost za CS segmentni selektor, koja će biti učitana na procesor kada u nekom trenutku počne izvršavanje procesa, postavljena je u liniji ⑥, a za ostale segmentne selektore između linija ⑦ i ⑧. Ove vrijednosti su izabrane tako da se nivo privilegija procesora prebaci na najniži nivo kada počne izvršavanje procesa.

Adresa od koje počinje izvršavanje procesa postavljena je u liniji ⑩, tako da odgovara početku .text sekcije koja se postavlja na nultu adresu prilikom uvezivanja svih XV6 programa. Inicijalni stek procesa postavljen je u liniji ⑨ na vrh alocirane stranice.

Obzirom na vrijednost eflags registra postavljenu u liniji ⑧, kada počne izvršavanje procesa, svi prekidi su omogućeni na procesoru.

## Promjena konteksta

Nakon završenog buđenja, svako AP jezgro odmah je u stanju da počne izvršavati kod nekog raspoloživog procesa. BSP jezgro kada završi proces pokretanja operativnog sistema također treba da počne izvršavati korisničke proceze. Dakle, sva raspoloživa jezgra u osnovi će u krenelu izvoditi jednu funkciju putem koje se prvo vrši izbor prvog procesa koji je u stanju RUNNABLE, a zatim počinje i izvršavanje odabranog procesa na jezgru koje izvodi kod ove funkcije. Funkcija koja vrši izbor procesa za izvršavanje na određenom jezgru u XV6 kernelu, zove se *scheduler*.

AP jezgra izvršavanjem slijedeće sekvence funkcija dolaze u poziciju da izvršavaju funkciju *scheduler*:

- buđenje → mpenter → mpmain → scheduler

Svako AP jezgro dok izvršava gore navedene funkcije koristi *sopstveni* stek kojeg je BSP jezgro alociralo u procesu buđenja AP jezgra.

BSP jezgro da bi došlo do izvršavanja funkcije `scheduler` izvodi nešto drugčiju sekvencu funkcija:

- `bootblock → main → mpmain → scheduler`

Dok izvršava svoju sekvencu funkcija, BSP jezgro koristi *sopstveni* stek alociran u BSS sekciji kernel ELF fajla ustanovljenoj u procesu uvezivanja, koja je alocirana u memoriji putem programa `boot-block` prilikom učitavanja kornela.

Okruženje koje se sastoji od funkcije koja se u nekom trenutku izvršava i steka kojeg ta funkcija koristi unutar nekog adresnog prostora, naziva se *nit* (thread). Dok izvršavaju bilo koji dio XV6 kornela, aktivirana jezgra koriste dva tipa niti. Svako jezgro ima tačno jednu nit koja izvršava kod funkcije `scheduler`, a koja koristi sopstveni stek jezgre. Sve ostale niti izvršavaju kernel funkcije putem kojih se tretira prekid nastao tokom izvršavanja nekog procesa na tom jezgru. Ove niti koriste kernel stek prekinutog procesa koji je alociran u funkciji `allocproc` prilikom kreiranja procesa.

Sva jezgra inicijalno izvršavaju nit sa funkcijom `scheduler`. Unutar ove funkcije jezgro bira proces kojeg će izvršavati u narednom vremenskom periodu. Po startanju sistema postoji samo jedan proces koji se može izvršavati u sistemu, a kojeg je BSP jezgro pripremilo u funkciji `userinit`.

Nakon odabira procesa za izvršavanje, jezgro mora "zamrznuti" `scheduler` nit te početi izvršavati nit za tretman prekida u odabranom procesu. Ukoliko je odabrani proces tek nastao, njegova kernel nit je pripremljena za povratak procesa iz sistemskog poziva `fork`. Tranzicija kojom procesor prestaje izvršavanje jedne i počinje izvršavanje druge niti, naziva se *promjena konteksta*. Promjena konteksta se razlikuje se od klasičnog poziva funkcije, jer pored promjene funkcije koja se u tom trenutku izvršava, uključuje i promjenu steka kojeg koristi procesor.

Nakon prve promjene konteksta, na jednom od jezgri aktivira se kernel nit kojom se tretiraju prekidi u procesu `init`. Ova nit prvo izvršava funkciju `forkret`, a zatim `trapret`. Izvršavanjem funkcije `trapret` prestaje izvršavanje kernel niti, a jezgro počinje izvršavati odabrani proces od adrese koja je postavljena u `trapframe` strukturi procesa `init`.

Kernel funkcija `scheduler` ima slijedeću implementaciju:

```
// preuzeto iz fajla proc.c

void scheduler(void)
{
    struct proc *p;

    for(;;){ ❶
        sti(); ❷

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

            proc = p; ❻
            switchuvn(p); ❼
            p->state = RUNNING; ❽
            swtch(&cpu->scheduler, proc->context); ❾
            switchkv(); ❿

            proc = 0; ❿
        }
        release(&ptable.lock);
    }
}
```

- ❶ Funkcija se obavlja u beskonačnoj petlji.
- ❷ Za trenutak se omogućavaju prekidi da bi jezgro moglo procesirati prekid koji je eventualno nastao u toku zadnje iteracije.
- ❸ Potreban je ekskluzivan pristup elementima niza `ptable.proc`, i to sve do odabira prvog `RUNNABLE` procesa.
- ❹ Ovim se ponovo onemogućavaju prekidi na jezgru sve dok se ne otpusti zaključana brava.

- ④ Skeniraju se svi procesi u nizu.
- ⑤ Ukoliko trenutni proces nije RUNNABLE, odmah se prelazi na sljedeći element u nizu.
- ⑥ Na jezgru koje izvršava kod funkcije `scheduler` odabrani proces postaje aktuelni proces.  
Jezgro aktivira virtualni adresni prostor aktualnog procesa koji je definiran putem polja `proc->pgdir`. Dodatno, u TSS segmentu jezgre vrši se promjena polja `SS0` i `ESP0` spram lokacije kernel steka aktuelnog procesa koja je određena putem izraza `proc->kstack+KSTACKSIZE`.
- ⑦ Proces se označava da je počeo sa izvršavanjem.
- ⑧ Vrši se promjena konteksta kojom jezgro prestaje izvršavati nit `scheduler`, a počinje izvršavati kernel nit koja tretira prekid u odabranom procesu. U ovom trenutku, `scheduler` nit je u potpunosti zamrznuta. Nit koja se promjenom kontekst počela izvršavati na ovom jezgru, mora otpustiti zaključanu bravu `ptable.lock` prije nego što kontrolu jezgra prepusti kodu odabranog procesa.

Neki proces koji se počne izvršavati nakon izvršenja promjene konteksta u liniji ⑨, u nekom budućem trenutku će biti prekinut uslijed djelovanja tajmera. Tretman tajmer prekida uključuje poziv kernel funkcije `yield` koja odraduje novu promjenu konteksta. Nakon što se prekinuti proces postavi u RUNNABLE stanje, promjena konteksta u funkciji `yield` obavlja se ponovo putem funkcije `swtch`, ali u obrnutom smjeru. Ovaj put će se zamrznuti kernel nit kojom je u prekinutom procesu tretiran prekid od tajmera, a probudit će se `scheduler` nit datog jezgra na lokaciji u kodu gdje je prethodno zaustavljena tj. od linije ⑩. Izvršavanjem `switch kvm`, jezgro aktivira kernel virtualni prostor, nakon čega se u liniji ⑪ označava da jezgro u tom trenutku ne izvršava niti jedan proces. Potom se vraća na početak beskonačne petlje, te se vrši izbor novog procesa za izvršavanje.

nje na datom jezgru, i to za idući vremenski interval tajmera. Ovim nije isključena mogućnost da jezgro ponovo odabere upravo prekinuti proces kao novi aktuelni proces.

Funkcija `swtch` kojom se obavlja promjena konteksta zaslužuje poseban tretman. Slijedeći asembler kod preuzet iz fajla `swtch.S` implementira ovu funkciju:

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

    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi ①

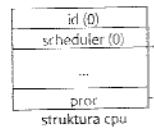
    movl %esp, (%eax) ②
    movl %esp, %esp ③

    popl %edi
    popl %esi
    popl %ebx
    popl %ebp ④
    ret ⑤
```

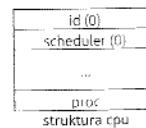
Na slici 9-5 prikazano je inicijalno stanje svih elemenata koji, prilikom poziva `swtch` unutar funkcije `scheduler`, učestvuju u promjeni konteksta. Registr `ESP` jezgra u tom trenutku pokazuje na vrh steka niti koja izvršava `scheduler` funkciju. Na steku se nalazi lokacija programskog brojača za povrat u funkciju `scheduler` i argumenti za funkciju `swtch` koji se u prvim linijama funkcije kopiraju u registre `eax` i `edx`.

Nakon pruzimanja argumenata sa steka, izvršavanjem instrukcija do linije ①, na steku se snimaju svi prezervirani registri procesora, što rezultira novim stanjem koje je prikazano na slici 9-6.

Izvršavanjem linije ②, adresa vrha aktuelnog steka snima se u polju `context` strukture `cpu` trenutnog jezgra. Nakon ovog, u liniji ③ vrši se promjena steka. Stek niti koja tretira prekid u odabranom



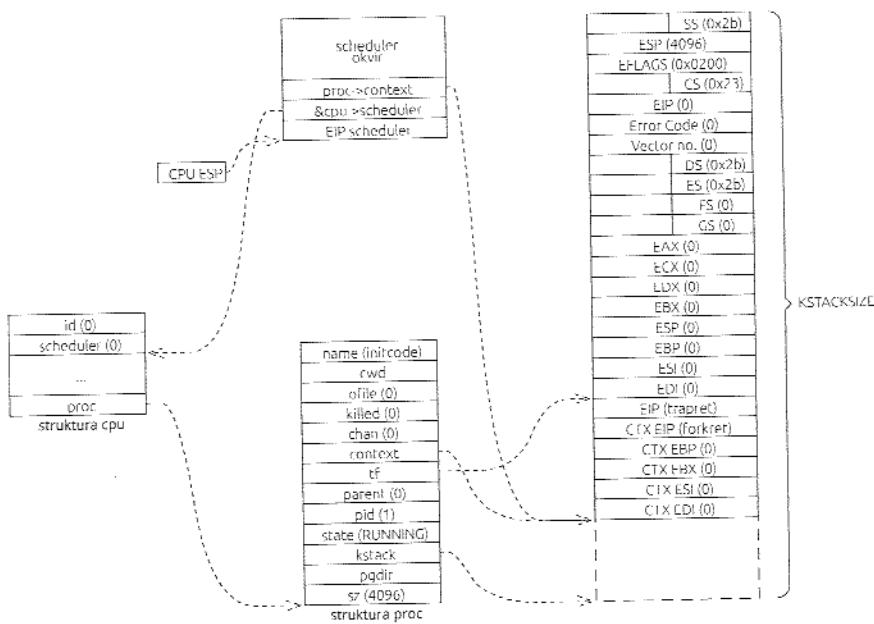
Slika 9-5. Stanje



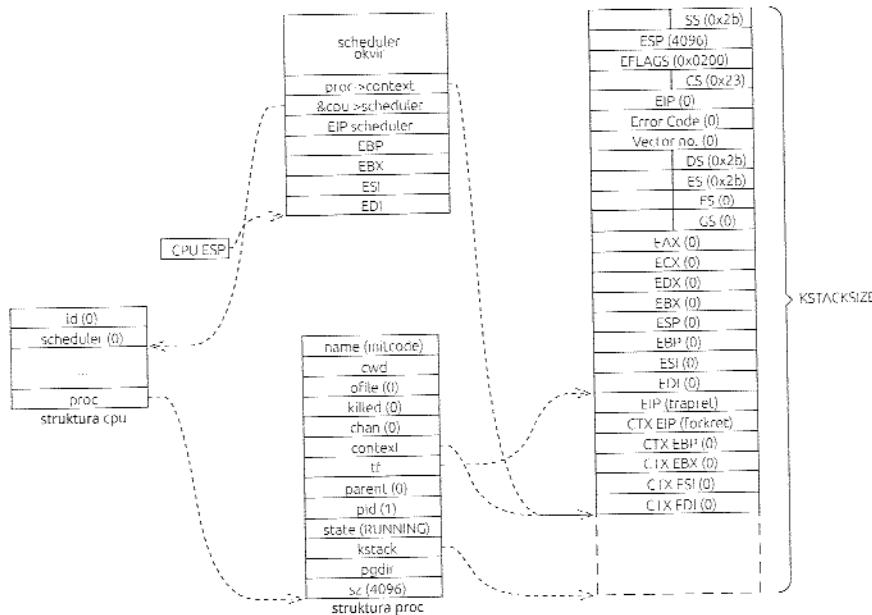
Slika 9-6. Stanje

era. Ovim  
čavo preki-

ka zaslužuje  
switch.S im-

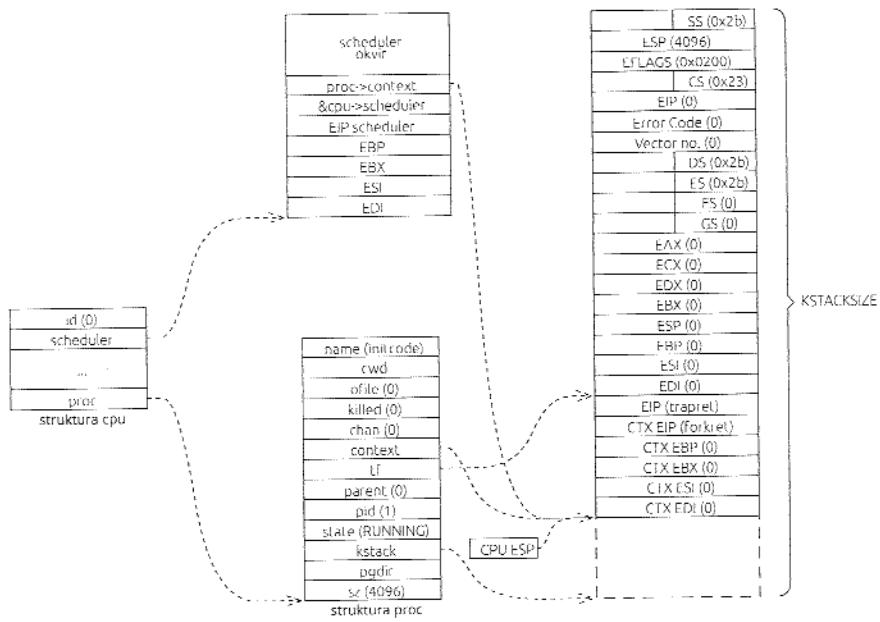


Slika 9-5. Stanje po pozivu funkcije `swtch`



Slika 9-6. Stanje nakon izvršenja linije ① u funkciji `swtch`

procesu postaje aktuelni stek, a registar ESP jezgra pozicionira se na lokaciju na steku određenu poljem context iz strukture proc aktuelnog procesa. Novo stanje, nakon linije ③, prikazano je na slici 9-7.

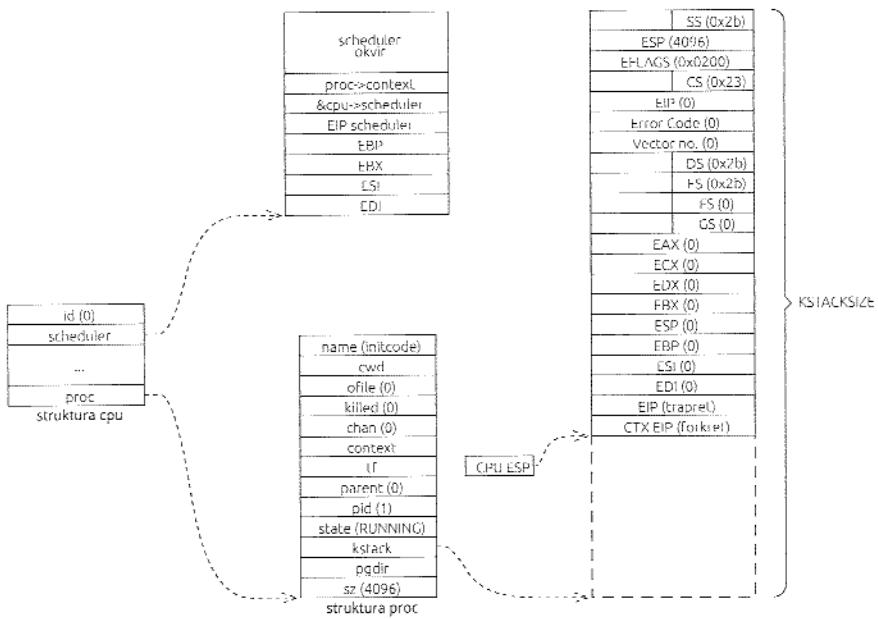
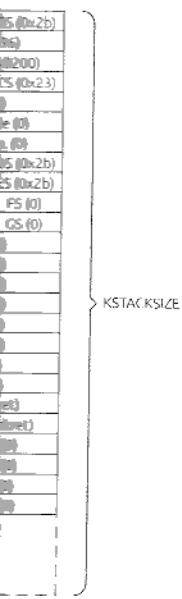


Slika 9-7. Stanje nakon izvršenja linija ② i ③ u funkciji swtch

Sa novog steka do linije ④ vrši se čitanje novih vrijednosti za pre-zervirane registre, čime nastaje stanje prikazano na slici 9-8. Na vrhu steku u tom trenutku ostaje adresa prve instrukcije u funkciji forkret. Izvršavanjem intrukcije u liniji ⑤, ova vrijednost kopira se u registar EIP na aktuelnoj jezgri, čime počinje izvršavanje funkcije forkret.

Funkcija forkret otpušta bravu ptable.lock i, ukoliko se izvršava prvi put, poziva funkcije bitne za pravilnu inicijalizaciju fajl sistema. Pozivom instrukcije return, funkcija preusmjerava jezgro da počne izvršavati kod spram vrijednosti za programski brojač koja je pročitana sa vrha aktuelnog steka, a to je adresa prve instrukcije u funkciji trapret.

ozicionira se na  
ure proc aktuel-  
je na slici 9-7.



Slika 9-8. Stanje nakon izvršenja linije ④ u funkciji swtch

Funkcija `trapret` putem `pop` i `popa` sa steka kopira vrijedosti za odgovarajuće registre, a posljednji registri učitavaju se na jezgro putem `iret`, čime prestaje izvršavanje niti kernela. Od ovog trenutka jezgro izvršava kod od odabranog procesa, a novo aktiviranje kernel niti procesa dogodit će se u trenutku kada nastane bilo kakav prekid u procesu koji se upravo počeo izvršavati.

### **Uspavljivanje i buđenje procesa**

Proces u toku izvršavanja može doći u situacije u kojim ne može nastaviti svoje izvršavanje dok se ne ispunи određeni uslov. Na primjer, pretpostavimo da se u nekom procesu izvršava program koji za određeni proračun zahtijeva podatke koji se nalaze na disku. Od trenutka kada od kernela zatraži čitanje podataka sa diska, program nema drugu opciju nego da čeka da se podaci sa diska učitaju u memoriju. Ukoliko ostane u RUNNABLE stanju, proces će trošiti procesorske resurse u beskonačnoj petlji sve dok se ne ispunii uslov za

nastavak njegovog izvršenja. Obzirom da je čitanje podataka sa diska iznimno spora operacija, pogodnija varijanta bi bila da se datom procesu onemogući šansa za izvršavanje na bilo kojoj jezgri, sve dok se u potpunosti na završi tražena disk operacija. Ovo je moguće učiniti postavljanjem procesa u stanje SLEEPING.

Proizvoljan broj procesa ptable niza u određenom trenutku mogu biti u stanju SLEEPING. Ovi procesi privremeno su uspavani, jer čekaju na ispunjenje određenog uslova da bi mogli nastaviti sa dalnjim izvršavanjem. Razlog uspavljinjanja nekog procesa modelira se putem kanala na kojem proces spava. Kanal je obično adresa neke kernel strukture i zapisuje se u polju chan strukture procesa. Procesi koji imaju isti kanal za spavanje biće probudeni istovremeno, jer čekaju na ispunjavanje istog uslova za nastavak izvršavanja. Buđenje obavlja kernel nit drugog procesa nakon što se konstatiše da je uslov za buđenje na nekom kanalu ispunjen.

Proces koji želi da bude uspavan treba da pozove funkciju sleep, koja ima slijedeću implementaciju:

```
// preuzeto iz fajla proc.c

void sleep(void *chan, struct spinlock *lk)
{
    if(proc == 0)
        panic("sleep");

    if(lk == 0)
        panic("sleep without lk");

    if(lk != &ptable.lock){
        acquire(&ptable.lock); ①
        release(lk); ②
    }

    proc->chan = chan; ③
    proc->state = SLEEPING; ④
    sched(); ⑤

    proc->chan = 0; ⑥

    if(lk != &ptable.lock){
        release(&ptable.lock); ⑦
        acquire(lk); ⑧
    }
}
```

}

① C  
② O  
③ P  
④ V  
⑤ O  
⑥ f  
⑦ F  
⑧ D

Pro  
uslov  
lova z  
brave  
brava  
zastoј  
ključa  
negoј

```
}
```

- ❶ Obzirom da mijenja stanje elementa iz `ptable` strukture, funkcija zaključava bravu `ptable.lock`.
- ❷ Otključava se brava za provjeru uslova.
- ❸ Postavlja se kanal na kojem proces spava.
- ❹ Vrši se promjena stanja procea.

Obavlja se promjena konteksta u funkciji `sched`, i to pozivom funkcije `swtch` kojom prestaje izvršavanje kernel niti trenutnog procesa, a počinje izvršavanje `scheduler` niti. Proces ostaje u uspavanom stanju sve dok se ne pozove `wakeup` na kanalu na kojem proces spava. Brava `ptable.lock` otpušta se nakon promjene konteksta u `scheduler` niti.

- ❺ Kada se u nekom trenutku pozove `wakup` na adekvatnom broju kanala, probuđeni proces, nakon što bude odabran za izvršavanje od strane `scheduler` niti nekog jezgra, nastavlja svoje izvođenje od ove linije koda.
- ❻ Otpušta se brava `ptable.lock` koja je ovaj put zaključana prije promjene konteksta u `scheduler` niti.
- ❼ Ponovo se zaključava brava za provjeru uslova.

Proces prije nego što pozove funkciju `sleep` vrši provjeru nekog uslova za uspavljivanje. Da bi se izbjegla stanja utrke, provjera uslova za uspavljivanje vrši se u kritičnoj sekciji pod kontrolom neke brave. Ukoliko se dozvoli da proces ode na spavanje sa zaključanom bravom, u sistemu bi se pojavila izgledna mogućnost za nastanak zastoja. Zbog toga, funkcija `sleep` uzima kao drugi parametar zaključanu bravu za provjeru uslova, koja se otključava u funkciji prije nego što se proces uspava.

Za buđenje svih procesa koji spavaju na određenom kanalu, neki proces treba da pozove funkciju `wakeup`. Funkcija imaju slijedeću implementaciju:

```
// preuzeto iz fajla proc.c

void wakeup(void *chan)
{
    acquire(&ptable.lock); ①
    wakeup1(chan);
    release(&ptable.lock);
}

static void wakeup1(void *chan)
{
    struct proc *p;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) ②
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE; ③
}
```

- ① Obzirom da mijenja stanje elementa iz `ptable` strukture, funkcija zaključava bravu `ptable.lock`.
- ② Skeniraju se svi elementi u tabeli.

Proces koji je u stanju `SLEEPING` na broju kanala koji je kao parametar proslijeden funkciji `wakeup`, stavlja se u stanje `RUNNABLE`, čime dobija šansu da se ponovo izvršava od lokacije u kodu na kojoj je uspavan.

Kao primjer upotrebe ovih funkcija razmotrit ćemo sistemski poziv `sys_sleep`. Svaki program može koristiti ovaj poziv da bi se njegov proces uspavao na izvjesno vrijeme, koje se računa na osnovu jednog parametra ovom pozivu. Parametar predstavlja broj intervala tajmera koji proces treba provesti u uspavanom stanju. Globalna varijabla `ticks` predstavlja ukupni broj generiranih prekida od tajmera na jezgru koje ima APIC identifikator 0. Pri svakom prekidu od tajmera na tom jezgru, `ticks` se inkrementira u funkciji `traps`, i to unutar kritične sekcije koja je pod kontrolom brave `tickslock`.

analu, neki  
slijedeću

strukture,

je kao  
stanje  
d loka-

temski  
a bi se  
osno-  
ja in-  
. Glo-  
ida od  
treki-  
traps,  
lock.  
odesi

Dodatno, pozivom funkcije `wakeup` unutar funkcije `trap`, vrši se buđenje svih procesa koji su uspavani na kanalu koji je određen adrešom varijable `ticks`.

Sistemski poziv `sys_sleep` ima slijedeću implementaciju:

```
// preuzeto iz fajla sysproc.c
int sys_sleep(void)
{
    int n;
    uint ticks0;

    if(argint(0, &n) < 0) ①
        return -1;
    acquire(&tickslock); ②
    ticks0 = ticks; ③
    while(ticks - ticks0 < n){ ④
        if(proc->killed){ ⑤
            release(&tickslock);
            return -1;
        }
        sleep(&ticks, &tickslock); ⑥
    }
    release(&tickslock);
    return 0; ⑦
}
```

- ① Sa steka procesa koji je izvršio sistemski poziv u varijablu `n`, čita se broj perioda tajmera u kojim proces treba biti uspavan.
- ② Zaključava se brava `tickslock`, i to radi izbjegavanja stanja utrke sa kernel niti koja inkrementira `ticks`.
- ③ U lokalnu varijablu `ticks0`, bilježi se dosadašnji broj prekida tajmera.
- Provjerava se ispunjenje uslova za nastavak izvršavanja procesa, i to u petlji, obzirom da će proces biti probudjen u svakom budućem prekidu tajmera.
- Ukoliko je od zadnjeg buđenja neki proces putem sistemskog poziva `sys_kill` zatražio terminiranje uspavanog procesa, proces odmah izlazi iz funkcije `sys_sleep` te se vraća u funk-

ciju `trap`, i to da bi pozvao funkciju `exit`, gdje se u potpunosti terminira.

Obzirom da uslov za nastavak izvršavanja procesa nije ispunjen, vrši se uspavljivanje procesa pozivom funkcije `sleep`,  
⑥ a paralelno se otključava i brava `tickslock`. Prilikom novog buđenja, proces ponovo provjerava uslov iz linije ②.

Obzirom da je proteklo tačno  $n$  intervala tajmera u kojim je  
⑦ proces bio uspavan, vrši se povrat iz sistemskog poziva u korisnički proces.

## Fajl sistemi

Bitnu ulogu u funkcionisanju računara imaju uređaji za permanentnu pohranu podataka koji se jednim imenom označavaju kao diskovi. Za razliku od memorije, podaci snimljeni na ove uređaje ne gube se nakon nestanka napajanja. Dodatno, diskovi obično imaju značajno veći kapacitet u odnosu na RAM memoriju. Sa druge strane, čitanje i pisanje podataka na ovim uređajima je izuzetno spor proces, što predstavlja njihov glavni nedostatak. Uzimajući u obzir navedene karakteristike, operativni sistemi u posebnoj komponenti označenoj kao *fajl sistem* implementiraju sve funkcije za transfer podataka između memorije i diskova. Dodatno, ova komponenta osigurava i slijedeće funkcije:

- organizacija i pretraga podataka,
- konsistentnost podataka,
- optimizacija procesa čitanja i pisanja.

Da bi fajl sistem mogao da obavlja gore navedene funkcije, podaci na disku moraju biti spremljeni u određenom formatu. Različiti fajl sistemi organiziraju podatke na mediju na različite načine. Dok većina modernih operativnih sistema podržava rad sa različitim fajl sistemima, XV6 implementira sopstveni fajl sistem, koji će biti predmet analize u ovom poglavlju.

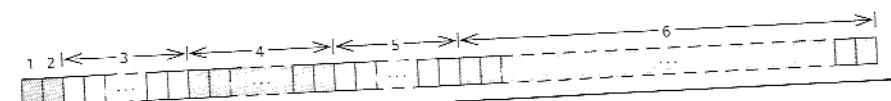
### Organizacija podataka na disku

Spram različitih tehnologija izrade, na današnjem tržištu postoje različiti tipovi diskova, npr. magnenti, ssd, fleš itd. Zajednička karakteristika svih diskova je da čitaju i pišu podatke u blokovima koji

se nazivaju *sektori*. Standardna veličina sektora je 512 bajta, a u posljednje vrijeme i 4KB. Sektori na disku se adresiraju od nulte adrese, slično kao što se pojedinačni bajti adresiraju u RAM memoriji. Različiti diskovi imaju različite brzine pristupa i transfera podataka iz pojedinačnih sektora, ali i najbrži diskovi su za nekoliko magnituda sporiji od RAM memorije.

Svi sektori diska mogu se ravnopravno koristiti za snimanje korisničkih podataka. Međutim, podaci snimljeni na disk nisu od preterane koristi ukoliko ne postoji metodologija za njihov pronalazak i čitanje. Pamćenje adresa blokova u kojim su snimljeni podaci nije adekvatno rješenje. Nadalje, postoji potreba za snimanje podataka u komadima koji zbog svoje veličine zahtijevaju upotrebu većeg broja sektora. Zbog toga, fajl sistemi određene sektore upotrebljavaju za snimanje informacija kojim se opisuje organizacija sadržaja snimljenog na disku, kao i status zauzetosti pojedinačnih sektora diska.

XV6 fajl sistem organizuje sektore na disku u šest različitih sekcija u skladu sa slikom 10-1.



Slika 10-1. Organizacija XV6 fajl sistema

Svaka sekcija fajl sistema ima specifičnu organizaciju i namjenu:

1. Sekcija sadrži samo jedan sektor, tzv. *bootblock*, koji je rezerviran za program za učitavanje operativnog sistema, kao i za informacije o particijama u sistemu.
2. Sekcija sadrži samo jedan sektor, tzv. *superblock*, u kome se nalazi jedna struktura čija polja nose informacije o ostalim sekcijama.
3. Sekcija se označava kao *log sekcija* i sadrži više sektora spram postavki u *superblock* strukturi. Sektori ove sekcije služe za obezbijedivanje konsistentnosti fajl sistema. XV6 organizuje

pisanje sadržaja na disk u transakcije. Sadržaj sektora koji se mijenjaju u okviru jedne transakcije prvo se snima u ovu sekciju, a po okončanju transakcije vrši se kopiranje sadržaja iz log sekcije u odgovarajuće odredišne sektore na disku.

4. Sekcija se označava kao dinode sekcija i sadrži više sektora spram postavki u superblock strukturi. Sadržaj sekcije je kontinuirani niz struktura tipa dinode. Ova struktura reprezentira osnovnu jedinicu za pohranjivanje sadržaja na disku. Ukupan broj dinode elemenata u ovoj sekciji određuje maksimalan broj fajlova i direktorija koji mogu biti kreirani u fajl sistemu. Svaki dinode element sadrži adrese sektora na disku u kojim se nalaze podaci koji pripadaju tom elementu.
5. Sekcija se označava kao bitmap sekacija i sadrži više sektora spram postavki u superblock strukturi. Svaki bit u ovoj sekciji korespondira jednom sektoru na disku, pri čemu stanje bita određuje da li je njemu asocirani sektor slobodan ili u upotrebi. Jedan sektor u ovoj sekciji nosi informacije o 4096 sektora na disku.
6. Sekcija se označava kao data sekcija i sadrži više sektora spram postavki u superblock strukturi. Sektori ove sekcije koriste se za snimanje sadržaja koji pripadaju dinode elementima definiranim u sekciji 4.

Struktura koja se nalazi u subperblock sekciji ima sljedeću definiciju:

```
// preuzeto iz fajla fs.h

struct superblock {
    uint size;      ①
    uint nblocks;   ②
    uint ninodes;   ③
    uint nlog;      ④
    uint logstart;  ⑤
    uint inodestart; ⑥
    uint bmapstart; ⑦
};
```

- ❶ Veličina diska izražena u broju sektora.
- ❷ Broj sektora u sekciji 6.
- ❸ Broj `dinode` elemenata u sekciji 4.
- ❹ Broj sektora u sekciji 3.
- ❺ Adresa prvog sektora sekcije 3.
- ❻ Adresa prvog sektora sekcije 4.
- ❼ Adresa prvog sektora sekcije 5.

Ova struktura uvijek se nalazi na početku sektora koji ima adresu 1, a sadržava informacije koje omogućavaju efikasan pristup i manipulaciju sadržaja ostalih sekcija na disku. Na primjer, za pronalaženje prvog slobodnog sektora u fajl sistemu potrebno je sprovesti sljedeće korake:

1. Čitanje sektora pod rednim brojem 1, čime se u memoriju učitava sadržaj `superblock` strukture.
2. Skeniranje pojedinačnih bita u `bitmap` sekciji.
  - Sadržaj sekcije u memoriju se učitava od sektora na disku određenog poljem `bmapstart` iz linije ❼.
  - Redni broj prvog bita u `bitmap` sekciji koji ima vrijednost 0 predstavlja adresu prvog sektora na disku koji je slobodan.

Sektori koji pripadaju sekciji 6 sadrže podatke od elemenata kreiranih u sekciji 4. Svaki element sekcije 4 modeliran je putem strukture `dinode` koja ima sljedeću definiciju:

```
// preuzeto iz fajla fs.h

struct dinode {
    short type; ❶
    short major; ❷
    short minor; ❸
    short nlink; ❹
```

```
    uint size;      ⑤  
    uint addrs[NDIRECT+1]; ⑥  
};
```

- ① Tip dinode elementa.
- ② Koristi se samo ukoliko je element uređaj, a predstavlja primarni broj uređaja.
- ③ Koristi se samo ukoliko je element uređaj, a predstavlja sekundarni broj uređaja.
- ④ Broj referenci na dati element u fajl sistemu.
- ⑤ Trenutna količina sadržaja izražena u bajtima.
- ⑥ Niz u kojem se nalaze adrese sektora na disku u kojima je snimljen sadržaj tog elementa.

XV6 u fajl sistemu dozvoljava četiri tipa dinode elemenata koji se mogu odabratи spram polja ①:

- T\_DIR → 1, predstavlja direktorij,
- T\_FILE → 2, predstavlja fajl,
- T\_DEV → 3, predstavlja uređaj.
- 0, element je slobodan, tj. nije u upotrebi.

Sadržaj dinode elemenata snima se na disku u sektorima data sekcije, a adrese konkretnih sektora korištenih za pohranjivanje sadržaja čuvaju se u nizu addrs koji je definiran u liniji ⑥. Svaki dinode element za snimanje svog sadržaja u osnovnoj konfiguraciji može koristiti maskimalno 140 sektora na disku. Adrese prvih NDIRECT (12) korištenih sektora bilježe se direktno u nizu addrs. Ukoliko dinode element spram količine asociranih podataka zahtijeva više od NDIRECT sektora, aktivira se još jedan niz koji omogućava bilježenje dodatnih 128 adresa sektora. Ovaj niz alocira se u bilo kojem neko-

rištenom sektoru data sekcije, a adresa odabranog sektora se snima u zadnjem elementu niza `addr`.

Elementi tipa `T_DEV` na disku nemaju nikakav asociran sadržaj, obzirom da reprezentiraju uređaje sa kojim je moguće komunicirati upotreboom funkcija koje implementira fajl sistem,

Sadržaj snimljen na disku koji je asociran sa elementima tipa `T_FILE` sa stanovišta operativnog sistema nije struktuiran i ostavljen je na interpretaciju aplikacijama koje ga čitaju ili pišu. Sa druge strane, sadržaj elemenata tipa `T_DIR` struktuiran je i upravljan od strane kernela, i to da bi se svi elementi fajl sistema organizirali u jednu podatkovnu strukturu tipa stablo. Svaki `dinode` element tipa `T_DIR` predstavlja jedan direktorij u fajl sistemu čiji je sadržaj na disku snimljen kao niz elemenata tipa `dirent`. Struktura `dirent` ima slijedeću definiciju:

```
// preuzeto iz fajla fs.h

#define DIRSIZ 14

struct dirent {
    ushort inum; ❶
    char name[DIRSIZ]; ❷
};
```

- ❶ Redni broj nekog `dinode` elementa iz sekcije 4 fajl sistema.
- ❷ Ime za odabrani element iz linije ❶.

Jedna `dirent` struktura asocira neki element iz fajl sistema odabran putem polja u liniji ❶, sa određenim imenom koje je postavljeno u polju `name` iz linije ❷. Za direktorij sa kaže da sadrži sve elemente fajl sistema koji su asocirani u njegovom nizu `dirent` struktura. Svi elementi fajl sistema koji su sadržani unutar istog direktorija moraju biti imenovani različito.

Opisana šema ima slijedeće implikacije:

1. Element dinode može biti referenciran više puta unutar fajl sistema, pri čemu:
  - ime može biti isto ili različito ukoliko se referenciranje vrši u različitim direktorijima,
  - ime mora biti različito ukoliko se isti element referencira više puta unutar istog direktorija,
2. Direktorij može da sadrži: druge direktorije, fajlove i uređaje, i to u ovisnosti od tipa elementa koji se referenciraju u njegovom nizu dirent struktura.

Kada se neki element tipa T\_FILE ili T\_DEV referencira u nekom direktoriju, XV6 inkrementira polje nlink u dinode strukturi tog elementa. Sa druge strane, ukoliko je element tipa T\_DIR, XV6 inkrementira njegovo polje nlink samo kada se u sadržaju tog direktorija doda novi direktorij. Ukoliko uslijed operacija u fajl sistemu polje nlink nekog dinode elementa dostigne vrijednost 0, XV6 uklanja sav sadržaj na disku asociran sa datim elementom, nakon čega se taj element unutar sekcije 4 označava kao slobodan. Dakle, svaki element da bi se nalazio u fajl sistemu mora biti referenciran u manje jednom direktoriju.

Svaki direktorij fajl sistema uvijek sadrži barem dva elementa u svom nizu struktura dirent, i to:

1. strukturu dirent čije polje name ima vrijednost “.”, putem koje dati direktorij uvijek ima referencu na svoj dinode element u fajl sistemu.
2. strukturu dirent čije polje name ima vrijednost “..”, putem koje dati direktorij uvijek ima referencu na dinode element od direktorija u kojem se trenutno nalazi.

Element pod rednim brojem 1 u dinode sekciji fajl sistema ima specijalnu namjenu i slijedeće karakteristike:

- uvijek je prisutan u fajl sistemu,
- uvijek je tipa T\_DIR, a njegovi elementi “.” i “..” referenciraju dinode element pod rednim brojem 1, tj. ovo je jedini direktorij koji nije sadržan u nekom drugom direktoriju fajl sistema.
- označava se kao root direktorij jer predstavlja vrh stabla fajl sistema.

Elementi tipa T\_DIR služe za logičku organizaciju sadržaja na disku, tako da se svi elementi u fajl sistemu mogu pronaći spram zadate staze koja u formi stringa identificira unikatni dinode element. Staza ima specifičan format, između dva sucesivna karaktera “/” unutar staze se nalazi ime nekog direktorija, a prvi karakter “/” u stazi predstavlja root dinode element.

Neka je kao primjer data slijedeća staza:

/opt/foo

Staza unikatno identificira dinode element koji je pod imenom foo referenciran kao sadržaj direktorija opt, a direktorij opt je referenciran kao sadržaj root dinode elementa. Dakle, pretraživanjem sadržaja direktorija u fajl sistem stablu, počevši od root elementa, moguće je pronaći bilo koji dinode element u skladu sa zadanim stazom.

## Fajl sistem organizacija u memoriji

Organizacija podataka na disku predstavlja vrlo bitan aspekt fajl sistema. Ništa manje bitne su kernel funkcije i podatkovne strukture kojim se vrši manipulacija sadržaja na disku.

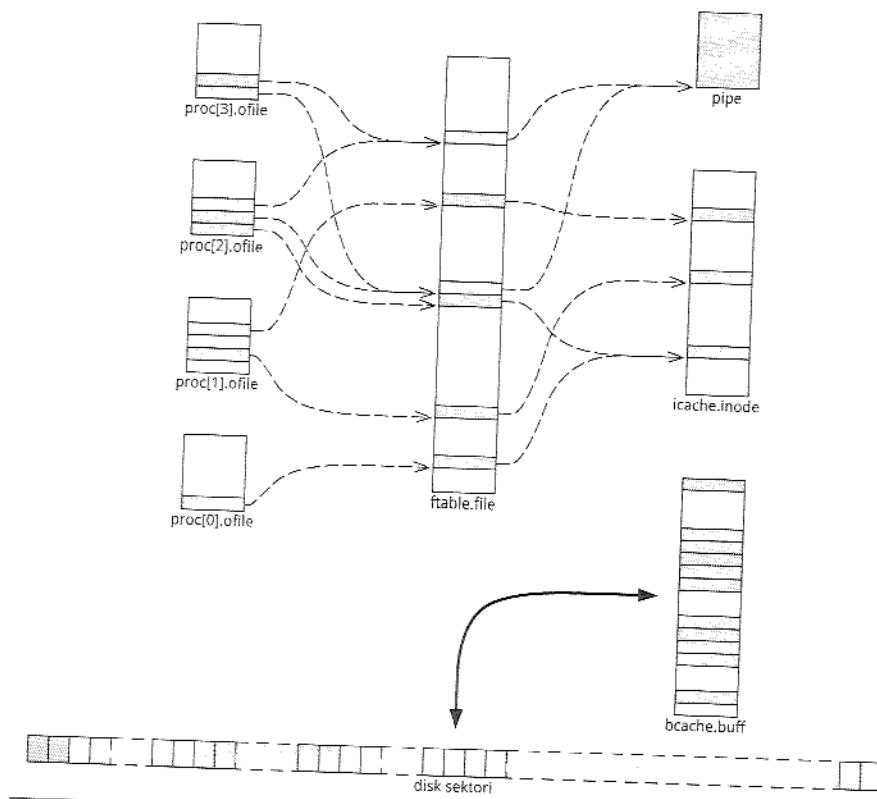


naima  
nciraju  
rekto-  
tema.  
bla fajl  
na di-  
m za-  
ement.  
era "/"  
r "/" u  
  
enom  
je re-  
anjem  
henta,  
n sta-

il sis-  
ture  
  
istem

Infrastruktura koju kernel u memoriji posvećuje fajl sistemu ima zadatok da obezbijedi:

- konkurentni pristup elementima fajl sistema iz više korisničkih procesa,
- sinhronizaciju pristupa radi izbjegavanja stanja utrke za pojedinačne sektore,
- optimizirano čitanje i pisanje podataka keširanjem često korištenih sektora,
- konsistentnost organizacije fajl sistema na disku u slučaju iznenadnog gubitka napajanja.



Slika 10-2. XV6 memorijske strukture za fajl sistem operacije

Za implementaciju navedenih funkcija, kernel koristi više memorijskih struktura čije međusobne relacije su prikazane na slici

10-2, pri čemu ključnu ulogu imaju tri globalne strukture: bcache, icache i ftable.

### BCACHE

Varijabla bcache predstavlja globalnu strukturu čije polje buf je dvostruko uvezana lista elemenata tipa bafer koji predstavljaju memoriske reprezentacije često korištenih sektora sa diska. Glavni zadatak ove liste je keširanje korištenih sektora sa različitih diskova. Pojedinačni sektor u listi je reprezentiran putem slijedeće bafer strukture:

```
// preuzeto iz fajla buf.h

struct buf {
    int flags; ①
    uint dev; ②
    uint blockno; ③
    struct buf *prev; ④
    struct buf *next; ⑤
    struct buf *qnext; ⑥
    uchar data[BSIZE]; ⑦
};
```

- ① Trenutno stanje bafera.
- ② Disk na kojem se nalazi sektor sa kojim je bafer asociran.
- ③ Broj sektora sa kojim je bafer asociran.
- ④ Prethodni bafer u listi bcache.
- ⑤ Slijedeći bafer u listi bcache .
- ⑥ Slijedeći bafer u jednostruko povezanoj listi bafera koju koristi disk drajver.
- ⑦ Keširana kopija sadržaja asociranog sektora sa diska.

Slijedeće funkcije skupa sa funkcijama disk drajvera operiraju na elementima bcache liste:

- `binit` — inicijalizira listu bafer elemenata.
- `bget` — asocira konkretni sektor na disku sa nekim baferom u listi, a putem zastavice `B_BUSY` postavlja bafer u zauzeto stanje. Proces koji pokuša `bget` operaciju na sektoru čiji je asocirani bafer u tom trenutku zauzet, mora da čeka na njegovo oslobođanje. Funkcija, po potrebi, sa kraja liste reciklira korištene baferne koji trenutno nisu u zauzetom stanju.
- `bread` — Počinje operaciju kopiranja kompletног sadržaja sektora na disku u asocirani bafer. Kada se završi kopiranje, disk drajver putem zastavice `B_VALID` označava bafer validnim.
- `bwrite` — Počinje operaciju kopiranja kompletног sadržaja nekog validnog bafera u njemu asocirani sektor na disku. Bafer se označava sa zastavicom `B_DIRTY`, koju naknadno uklanja disk drajver kada se u potpunosti završi operacija kopiranja.
- `brelse` — Za proslijedeni bafer uklanja zastavicu `B_BUSY` i budi sve procese koji su čekali na oslobođanje tog bafera.

Obzirom da više procesa u isto vrijeme može pozivati operacije `bread` ili `bwrite` na različitim baferima, disk drajver organizuje zadatke čitanja ili pisanja bafera u jednostruko povezanu listu koja se procesira po fifo principu. Tokom procesiranja bafera u ovoj listi, disk drajver drži uspavane sve procese čiji se zahtjevi za čitanje ili pisanje diska nalaze u listi.

Da bi se u slučaju gubitka napajanja zadržala konsistentnost stanja na disku, fajl sistem funkcije višeg nivoa ne pozivaju funkciju `bwrite` direktno, već koriste posebnu log komponentu fajl sistema koja pisanja na disk organizira u transakcije. Izmijenjeni sadržaji bafera iz bcache liste unutar jedne transakcije prvo se snimaju u sektore fajl sistema koji čine log sekciјu na disku. Tokom transakcije bilježe se odredišni sektori za sadržaj koji se snima u log sekciјi. Ka-

da se okonča snimanje sadržaja zadnjeg izmijenjenog bafera u okviru jedne transakcije, u zaglavljaju log sekcijskog fajla sistema upisuje se vrijednost kojom se signalizira da je kompletan sadržaj transakcije snimljen u log sekcijskoj. Nakon ovog, počinje kopiranje sadržaja log sekcijskog fajla u odgovarajuće odredišne sektore na disk. Prvobitno postavljeni signal u zaglavljaju log sekcijskog fajla uklanja se tek po okončanju kopiranja kompletног sadržaja snimljenog u log sekcijskoj. U slučaju gubitka napajanja prije nego što se u zaglavljaju log sekcijskog fajla postavi odgovarajući signal, sadržaj izmijenjenih sektora u okviru transakcije u potpunosti je izgubljen, ali fajl sistem ostaje u prijašnjem konsistentnom stanju. Međutim, ako prekid u napajanju nastane u bilo kojem trenutku nakon što je signal zapisan u zaglavljaju log sekcijskog fajla, pri sljedećem pokretanju operativnog sistema, a neposredno pred startanje procesa `init`, nanovo započinje kopiranje kompletног sadržaja koji je zapisan u log sekcijskoj prije neuspjeli transakcije. Po okončanju ovog kopiranja, fajl sistem na disku je u novom konsistentnom stanju.

Transakcija započinje pozivom funkcije `begin_op`, a završava sa `end_op`. Pozivanjem ovih funkcija u sekvenci, proizvoljan broj procesa može učestvovati u istoj transakciji, pri čemu kernel vodi računa da se sadržaj bafera izmijenjenih u okviru transakcije počne snimati u log sekcijsku diskova tek kada zadnji proces pozove `end_op`. Isto tako, unutar funkcije `begin_op` procesi mogu biti uspavani do okončanja već pokrenute transakcije, ili da bi se izbjegle situacije u kojima bi se u okviru jedne transakcije prepunila log sekcijska diskova. Izmijenjeni bafer se dodaje u transakciju putem funkcije `log_write`.

Slijedeći segment koda predstavlja tipičan scenario upotrebe opisanih funkcija:

```
begin_op(); ①
struct buf *b = bread(disk, sektor); ②
b->data[128]=4; ③
log_write(b); ④
brelse(b); ⑤
end_op(); ⑥
```

① P  
A  
ž  
i  
t  
③ L  
④ L  
⑤ D  
⑥ C

ICACH

Varija  
preze  
sekciј  
na slij// p  
strv  
ut  
ut  
in  
in  
sk  
sk  
sk  
sk  
ut  
ut  
};;①  
②

Fajl

① Početak transakcije.

Asocira sektor sa nekim raspoloživim baferom i kopira sadržaj sektora sa diska u bafer.

- ② Eventualni istovremeni pristup istom sektoru za više procesa sinhronizira se unutar funkcije `bread`.

③ Mijenja sadržaj bafera.

④ Dodaje bafer u transakciju.

Otpušta bafer, kako bi asociranom sektoru na disku mogao

- ⑤ pristupiti neki drugi proces.

⑥ Okončava transakciju sa stanovišta datog procesa.

## ICACHE

Varijabla `icache` predstavlja globalnu strukturu čije polje `inode` reprezentira niz u kojem se keširaju pojedinačni `dinode` elemenati iz sekcije 4 fajl sistema na disku. Niz sadrži elemente koji su definirani na slijedeći način:

```
// preuzeto iz fajla file.h

struct inode {
    uint dev;      ①
    uint inum;     ②
    int ref;       ③
    int flags;     ④

    short type;   ⑤
    short major;  ⑤
    short minor;  ⑤
    short nlink;  ⑤
    uint size;    ⑤
    uint addrs[NDIRECT+1]; ⑤
};
```

- ① Broj diska na kojem se nalazi keširani `dinode` element.

- ② Redni broj `dinode` elementa koji se kešira.

#### *• ~~Trasumirati sve vrijednosti resursa u nizu icache~~*

- ④ Stanje elementa.
- ⑤ Polja koja se kopiraju sa diska iz keširanog **dinode** elementa.

Struktura **inode** predstavlja memorijsku reprezentaciju određenog fajla, direktorija ili uređaja, koji se na disku nalaze u formi **dinode** elementa u sekciji 4 fajl sistema. Struktura iz niza **icache** nalazi se u validnom stanju ukoliko je asocirana sa nekim **dinode** elementom sa diska, te ukoliko su polja ⑤ od asociranog **dinode** elementa pročitana sa diska. Proizvoljan broj procesa može u isto vrijeme koristiti keširani **dinode** element, pri čemu se broj procesa koji trenutno imaju referencu na element prati putem polja ③. Ukoliko neki proces koji ima referencu na strukturu **inode** vrši manipulaciju sa stanjem keširanog **dinode** elementa, struktura se stavlja u zauzeto stanje sve dok traje manipulacija keširanim **dinode** elementom. Polje ④ bilježi trenutno stanje strukture.

Slijedeće funkcije operiraju na elementima **icache** niza:

- **iget** — Asocira **dinode** element sa diska sa nekim **inode** elementom u **icache** nizu. Ne vrši učitavanje polja ⑤ sa diska i ne postavlja **inode** u validno stanje, ali inkrementira brojač referenci ③.
- **ilock** — Stavlja **inode** u zauzeto stanje. Ukoliko element nije validan, učitava polja ⑤ sa diska te ga stavlja u validno stanje.
- **iunlock** — Označava da **inode** više nije zauzet i budi procese koji su čekali na ovu promjenu.
- **iput** — Dekrementira brojač referenci ③. Ukoliko se konstatuje da polja **ref** i **nlink** imaju vrijednost nula, resursi keširanog **dinode** elementa potpuno se oslobođaju na disku.
- **iupdate** — Kopira stanje iz **inode** elementa u asocirani **dinode** element na disku.

Za pristup sadržaju na disku koji je asociran sa određenim **dinode** elementom keširanim u **icache** nizu, mogu se koristiti funkcije **readi** i **writei**. Funkcije vrše transfer određene količine podataka između neke adrese u memoriji i lokacije unutar sadržaja **dinode** elementa na disku, pri čemu funkcija **readi** čita podatke sa diska, a **writei** piše podatke na disk.

Funkcije **writei**, **iput** i **iupdate** prepostavljaju da se pozivaju unutar neke transakcije.

Slijedeći segment koda predstavlja tipičan scenario upotrebe opisanih funkcija:

```
struct inode* ip = iget(uređaj, dinode) ①
ilock(ip) ②
// čitanje ili promjena stanja
// keširanog dinode elementa ili
// pristup njegovom sadržaju na disku
iunlock(ip) ③
iput(ip) ④
```

- ① Preuzima referencu na određeni **dinode** element.
- ② Označava da se keširani element od ovog trenutka koristi. Po potrebi, učitava sadržaj svih polja **dinode** elementa sa diska.
- ③ Nakon obavljene promjene stanja, **inode** element se označava slobodnim kako bi ga mogao koristiti neki drugi proces.
- ④ Efektivno označava da proces više neće manipulirati stanjem **inode** elementa, tj. da ga zatvara.

Putem funkcija **namei**, **nameiparent** i **namex**, XV6 omogućava pronalaženje i keširanje **dinode** elemenata na osnovu zadane staze u fajl sistemu. Funkcije prolaze kroz sekvencu čitanja različitih sektora fajl sistema, sve dok ne pronađu i keširaju traženi **dinode** element. Na primjer, neka je data slijedeća linija koda:

```
struct inode* ip = namei("/opt/foo")
```

Da bi se prvo pronašao, a zatim u memoriju i učitao traženi `dinode` element, funkcija mora pročitati, a potom i analizirati sadržaj sljedećih sektora diska:

1. `superblock` sektor, da bi se dobio broj sektora od kojeg počinje `dinode` sekcija fajl sistema.
2. sektor koji sadrži `root` element, tj. `dinode` element pod rednim brojem 1.
3. sektor u kojem je snimljen sadržaj `root` elementa, kako bi se kroz analizu `dirent` struktura iz pročitanog sadržaja `root` direktorija mogao pronaći redni broj `dinode` elementa koji je asociran sa imenom "opt".
4. sektor u kojem je snimljen `dinode` element čiji je redni broj do- biven u prethodnom koraku.
5. sektor u kojem je snimljen sadržaj direktorija opt, čime se do- lazi do rednog broja `dinode` elementa koji je asociran sa imenom foo.
6. sektor u kojem se nalazi `dinode` element čiji je redni broj dobi- ven u prethodnom koraku.

Čak i ovako elementarna operacija fajl sistema zahtijeva čitanje velikog broja sektora sa diska. Zbog uloge koju imaju, određeni sektori diska koristit će se češće nego ostali, pa će se zbog toga njihov sadržaj duže keširati u nekom od bafera iz strukture `bcache`.

## FTABLE

Varijabla `ftable` predstavlja globalnu strukturu čije polje `file` reprezentira tabelu svih otvorenih fajlova u sistemu. Otvoreni fajlovi modelirani su putem strukture koja ima slijedeću definiciju:

```
// preuzeto iz fajla file.h
struct file {
```

```

enum { FD_NONE, FD_PIPE, FD_INODE } type; ①
int ref; ②
char readable; ③
char writable; ④
struct pipe *pipe; ⑤
struct inode *ip; ⑥
uint off; ⑦
};

```

- ① Tip otvorenog fajla.
- ② Broj memorijskih referenci.
- ③ Indikator da je dozvoljeno čitanje.
- ④ Indikator da je dozvoljeno pisanje.
- ⑤ Pointer na asocirani `pipe`, pod uslovom da je tip `FD_PIPE`.
- ⑥ Pointer na asocirani `inode`, pod uslovom da je tip `FD_INODE`.
- ⑦ Trenutni pomak unutar fajla od kojeg se vrši slijedeće čitanje ili pisanje.

Otvoreni fajl može reprezentirati `dinode` element na disku ili `pipe`, poseban kernel objekat koji služi za komunikaciju između procesa. Kada je fajl putem pointera ⑤ asociran sa `dinode` elementom keširanim u `icache` nizu, svako čitanje ili pisanje sadržaja asociranog elementa inkrementira polje ⑦ spram količine bajta koja se prebacuje u toku operacije. Polje ② predstavlja broj procesa koji istovremeno koriste isti fajl.

Slijedeće funkcije operiraju na elementima `ftable` niza:

- `filealloc` — Pronalazi prvi element u `ftable` nizu koji se u tom trenutku ne koristi, te ga stavlja u zauzeto stanje postavljanjem polja `ref` na vrijednost 1.
- `filedup` — Inkrementira polje `ref` proslijedenog fajla.
- `fileread` — Spram trenutnog pomaka `off`, kopira `n` bajta iz sadržaja asociranog `dinode` elementa na zadatu lokaciju u memoriji.

riju. Alternativno, ukoliko je fajl tipa FD\_PIPE, čitanje se vrši iz asociranog pipe elementa.

- `filewrite` — Kopira n bajta pročitanih sa određene adrese u memoriji, u sadržaj asociranog dinode elementa spram trenutnog pomaka `off`. Alternativno, ukoliko je fajl tipa FD\_PIPE, pisanje se vrši u asocirani pipe element.
- `fileclose` — Zatvara asocirani pipe ili dinode element, mijenja tip fajla u FD\_NONE, a polje `ref` stavlja na vrijednost 0.

Ukoliko više procesa koji imaju referencu na isti fajl pokušaju operaciju `filewrite` ili `fileread`, sinhronizacija ovih operacija vrši se na nivou asociranog inode ili pipe elementa. Ovo garantuje da će svi procesi obaviti operacije, ali ne definira nikakav redoslijed njihovog izvršavanja. Dodatno, svi procesi koji dijele isti fajl čitaju ili pišu sadržaj spram istog trenutnog pomaka definiranog putem polja `7`. Sa druge strane, procesi koji otvore isti dionde element, ali u različitim fajlovima, koriste različite trenutne pomake prilikom čitanja ili pisanja.

## Fajl deskriptori

Programi nemaju mogućnost direktnog pristupa elementima globalne fajl tabele. Međutim, kernel za svaki proces u njegovoј proc strukturi kreira polje `ofile` koje predstavlja niz pointera na strukture tipa `file`. Programi putem posebnih sistemskih poziva mogu tražiti od kernela da u njihovo ime otvori određeni element sa diska, a zatim kreira fajl u globalnoj tabeli. Kernel potom postavlja pointer na otvoreni fajl na najnižu nekorištenu poziciju u `ofile` nizu tog procesa. Indeksi u `ofile` tabeli procesa označavaju se kao *fajl deskriptori*. Kada fajl deskriptor u tabeli `ftable` ima asociran fajl, proces putem sistemskih poziva može čitati ili mijenjati sadržaj fajla.

Za kreiranje elemenata fajl sistema na disku, XV6 obezbijeduje slijedeće sistemske pozive:

- `open` — Za kreiranje dinode elemenata tip `T_FILE`, i to u skladu sa zadatom stazom.
- `mkdir` — Za kreiranje dinode elemenata tip `T_DIR`, i to u skladu sa zadatom stazom.
- `mknod` — Za kreiranje dinode elemenata tip `T_DEV`, i to u skladu sa zadatom stazom i tipom uređaja na osnovu dva cijela broja (major i minor).

Svi navedeni pozivi rezultiraju i kreiranjem odgovarajućih elemenata u kernel nizovima `ftable` i `icache`. Poziv `open` koristi se i za otvaranje već postojećih dinode elemenata na disku, a uvijek vraća fajl deskriptor koji je asociran sa otvorenim elementom. Pri pozivu `open` mogu se koristiti dozvole: samo za čitanje, samo za pisanje ili istovremeno za čitanje i pisanje, pri čemu direktoriji podržavaju pristup samo za čitanje.

Za kreiranje dodatnih imena u skladu sa novom i već postojećom stazom za kreirane fajlove ili uređaje, koristi se sistemski poziv `link`. Sa druge strane, za uklanjanje imena sa određene staze fajl sistema koristi se sistemski poziv `unlink`. Ukoliko dinode element asociran sa imenom koje se uklanja, nema više referenci u fajl sistemu, `unlink` rezultira uklanjanjem njegovog sadržaja sa diska.

Prilikom operacije `fork`, svi deskriptori iz trenutnog procesa kopiraju se u njegovo dijete. Obzirom da se na vrhu hijerarhije procesa nalazi `init`, svaki proces dobija tri unaprijed definirana deskriptora koji su u procesu `init` asocirani sa uređajem `/console`. Njihovi redni brojevi su:

- 0 → Standardni ulaz.
- 1 → Standardni izlaz.
- 2 → Izlaz za greške.

Za kreiranje fajlova koji su asocirani sa objektima tipa `pipe`, koristi se sistemski poziv `pipe` u slijedećem obliku.

```
int fd[2];
pipe(fd); ❶
```

Poziv funkcije u liniji ❶ na kernel strani alocira jedan `pipe` objekat, kao i dva fajla u nizu `ftable` koji su asocirani sa kreiranim `pipe` objektom. Prvi fajl je kreiran za čitanje iz objekta, a drugi za pisanje u objekat. Dodatno, u procesu koji je pozvao `pipe`, kernel kreira i dva fajl deskriptora koji su asocirani sa kreiranim `pipe` fajlovima, a njihova vrijednost zapisuje se u proslijedjeni niz `fd`. Objekat `pipe` koristi se za komunikaciju dva procesa koji su u relaciji roditelj — dijete i to na način da jedan proces piše a drugi čita iz deskriptora koji su inicijalizirani pri kreiranju `pipe` elementa. Pri pisanju i čitanju `pipe` objekta, kernel vrši neophodnu sinhronizaciju.

Kada na raspolaganju ima bilo koji fajl deskriptor, proces putem sistemskih poziva `write` i `read` može pisati i čitati podatke iz elementa koji je asociran sa fajl deskriptorom. Dakle, ne postoji nikakva razlika u čitanju ili pisanju podataka iz fajla na disku, uređaja ili `pipe` objekta. Po okončanju svih operacija, fajl asociran sa deskriptorom potrebno je zatvoriti upotrebom sistemskog poziva `close`, čime se oslobođa prethodno zauzeti fajl deskriptor. Na primjer, neka je dat slijedeći segment koda nekog korisničkog programa:

```
char buf[] = "Pozdrav";
int fd = open("/foo.txt", O_WRONLY | O_CREATE); ❶
write(fd, buf, 8); ❷
close(fd); ❸
```

- Na disku se kreira fajl na stazi `/foo.txt`. Kreirani fajl se otvara sa dozvolama za pisanje, a ujedno se vraća i asocirani fajl deskriptor `fd`.
- ❶ Putem deskriptora `fd` u fajl se upisuje osam bajta pročitanih iz memorije sa lokacije `buf`.

- pe, ko-
- 3 Po okončanju operacija otvoreni fajl se zatvara.

