Nastavno minimalističko jezgro operativnog sistema

Samir Ribić, Adnan Salihbegović
Elektrotehnički fakultet u Sarajevu
Univerzitet u Sarajevu
Sarajevo, Bosna i Hercegovina
samir.ribic@etf.unsa.ba, adnan.salihbegovic@etf.unsa.ba

Sadržaj—**Kursevi operativnih sistema su često organizovani prema tri osnovna pristupa, kao korisnički orijentisani, teoretski kroz objašnjenje principa rada ili kroz analizu i modifikovanje izvornog koda jezgra. Dok je treći pristup najilustrativniji, on zahtijeva i najviše vremena na uštrb širine tematike analize implementacije različitih algoritama operativnih sistema. Stoga se često pristupa razvoju malih jezgara, umjesto analize onih koja su u široj upotrebi. U ovom radu će se predstaviti jedno jednostavno jezgro, sa minimalnom količinom izvornog koda, da se u cjelosti može opisati za nastavnih 90 minuta, ali ipak dovoljno da se samostalno pokrene na jednom tipičnom PC računaru i da uključuje osnovne funkcije jezgra: rad s ekranom, tastaturom, diskom, datotečnim sistemom, memorijom, procesima i nitima.**

Ključne riječi-operativni sistemi; edukacija; minimalizam;

I. UVOD

Većina fakulteta na kojima se obrazuje kadar iz oblasti računarskih nauka i informacionih sistema u svom planu i programu ima i kurs operativnih sistema. U kursevima koji pokrivaju ovu fundamentalno značajnu temu iskristalisala su se tri osnovna pristupa.

Na studijima orijentisanim prema informacionim sistemima i informacionim tehnologijama, česti su korisnički orijentisani kursevi, više okrenuti praktičnim problemima instalacije, konfiguracije i administracije operativnih sistema.

Teoretska analiza algoritama koji se javljaju u implementaciji operativnih sistema je pristup koji je dominantan u kursevima operativnih sistema na fakultetima orijentisanim prema računarskim naukama. Prednost pristupa je u široj slici koju studenti imaju o strukturi operativnih sistema, a mana što predstava operativnih sistema i dalje ostaje apstraktna.

Metoda da se principi rada operativnih sistema uvode kroz objašnjenje ili razvoj izvornog koda jezgra nekog operativnog sistema se primjenjuje ukoliko studenti imaju dovoljnog predznanja. Na ovaj način se bolje shvata suština operativnog sistema ali se smanjuje manevarski prostor za upoznavanje s drugim pristupima.

II. NEKI NASTAVNI OPERATIVNI SISTEMI

Veličina izvornog koda je pojam koji ima sasvim različito značenje u produkciji i edukaciji. Na primjer, jedan megabajt predstavlja oko 20000 linija izvornog koda. I dok je to na disku zanemarljiva količina, odštampano na papir to predstavlja oko 500 strana, dakle količina gradiva za čije je objašnjavanje potreban cijeli semestar. To je, naravno, neprihvatljivo jer je opisivanje isključivo izvornog koda dosta monoton način predavanja. Operativni sistemi koji se koriste u realnom svijetu (Windows NT jezgro i Linux), su daleko veći, na primjer Windows 7 ima 2 miliona, a Linux 3.10 ima 17 miliona linija koda, samo za jezgro. Stoga njihov izvorni kod ne može biti nastavno sredstvo za proučavanje principa operativnih sistema.

Ako se želi da izvorni kod jezgra operativnog sistema bude dodatak teoretskom razmatranju principa operativnih sistema kao ilustracija rada na realnoj mašini, praktičniji su primjeri manjih dimenzija. Često se koriste starije verzije uspješnih operativnih sistema ili se razvijaju vlastiti mali operativni sistemi.

Najčešći pristup je razvoj jezgra koje podsjeća na Unix jezgro. To nije iznenađenje s obzirom na popularnost ovog operativnog sistema i njegove korijene na univerzitetu.

Minix [1] je najpoznatiji nastavni operativni sistem. Djelomično je Unix kompatibilan i objašnjen u knjizi, uz dodatak teorije operativnih sistema koja se opisuje uz sam izvorni kod podsistema koji se opisuje. Dužine je oko 16000 linija, pa se cijeli kurs treba organizovati oko njega. Upotrebljiv je i za visoko-pouzdane sisteme, a arhitektura je sa mikro-kernelom.

XINU [2] je također korišten kao jezgro oko koga je napisana popularna knjiga iz oblasti operativnih sistema. Prva verzija je namijenjena za PDP-11, a zatim su se pojavile verzije za IBM PC [3], Motorola 68000 sisteme, Mips, Spark i nedavno ARM. Pristup je dosta razumljiv, ima oko 8000 linija koda, ali ne ulazi u temu datotečnih sistema, nego se na PC računarima oslanja na MS DOS pozive za rad s datotekama.

Nachos [4] je pisan u objektno-orijentisanim jezicima, postoje C++ i Java verzija, uz dodatak C i MIPS asemblerskog koda. Zauzima oko 5000 linija koda, a izvršava se u virtualnom okruženju.

Stare verzije Unix i Linux jezgra se također koriste u edukativne svrhe. Na MIT univerzitetu za nastavu dizajna operativnih sistema dugo je korištena stara verzija Unix V6, sa oko 8300 linija koda. Nakon što je studentima bila nejasna svrha učenja zastarjelog sistema na zastarjeloj platformi napisanog u prastaroj verziji jezika C, pod imenom xv6 prebačena je na 80386 platformu [5]. U slučaju Linux-a, verzija 0.11 je bila prva verzija sposobna za samostalno

izvršavanje, a imala je 14000 linija koda. Komentarisana verzija ovog jezgra korištena je u [6], a skraćivanjem na 5000 linija nastao je Qutenix [7].

Windows Research Kernel ima oko 830000 linija koda [8] i dostupan je edukacijskim ustanovama, uz posebne ugovore sa Microsoftom. Uprkos tolikoj veličini on ne može samostalno da se izvršava.

Ako se jezgro oslobodi dizajnerskog zahtjeva da bude slično Unix-u, moguća su i manja jezgra. Geekos [9], sa izvornim kodom od oko 3000 linija je primjer jezgra koje je dizajnirano da bude malo i stoga lako shvatljivo, a da pri tome nije opterećeno zahtjevom za sličnost s Unix-om ili Windows-om. Međutim, ovaj operaticni sistem nema datotečnog sistema. U [10] je prikazan nastavni koristi floppy baziran CP/M sličan operativni sistem, sa priličnim ograničenjima (datoteke do 13K i procesi do 64K), ali je izvorni kod svega 1200 linija, što ga čini najmanjim nastavnim jezgrom do koga smo došli u toku istraživanja.

III. STRUKTURA KERNELA

A. *Principi dizajna jezgra*

U svrhu boljeg razumijevanja principa operativnih sistema a da se ipak ne potroši previše vremena u analizi izvornog koda, razvijeno je vlastito minimalističko jezgro operativnog sistema. Principi na kojima je ono dizajnirano su sljedeći.

Jezgro mora biti sposobno da se samostalno izvršava na realnom hardveru, a ne na virtualnoj mašini. Ono ne treba biti ovisno o postojećem operativnom sistemu. Zbog široke raspoloživosti, ovo jezgro je namijenjeno za 386 kompatibilne PC računare u trideset-dva bitnom režimu rada, sa IDE diskom, tastaturom i tekstualnim ekranom. Namijenjeno je za pokretanje pod emulatorom qemu, ali se može izvršavati i na realnoj mašini. U cilju razjašnjavanja nekih tehničkih specifičnosti korištene su informacije s web stranice http://wiki.osdev.org/Main_Page.

Potrebno je uključiti u jezgro funkcionalnosti ulaza/izlaza, organizacije memorije, diska, datotečnog sistema i raspoređivanja procesa. No, pri tome se mogu koristiti što jednostavniji algoritme u realizaciji podsistema (kao što su first fit kod memorije, kontinualna alokacija kod datotečnog sistema, ili kooperativni round robin za raspoređivanje procesa), uz mogućnost da studenti kroz zadatke realizuju i naprednije algoritme, ili se sa njima teoretski upoznaju.

Ovo jezgro je napisano u jeziku C, izuzev boot sektora, dijela koda za spašavanje registara i vrlo kratkih rutina za ulaz/izlaz i strukturu steka interapta koji su u asemblerskom jeziku.

Jezgro je organizovano slojevito iz metodičkih razloga. Svaka nova datoteka izvornog koda može da koristi prethodne, ali uz dodatak main funkcije sa njima može predstavljati i zaokruženu cjelinu. U tabeli su prikazane datoteke izvornog koda jezgra s njihovom dužinom. Tabela predstavlja i redoslijed kojima se one obrađuju. Testne datoteke se pozivaju nakon objašnjenja rada pojedinih podprograma i sadrže main program jezgra.

TABELA I. DATOTEKE IZVORNOG KODA

Datoteka	Uloga	Lin.
bootsect.asm	Učita jezgro i pređe u zašt. režim	85
string.c	Rad s nizovima znakova	41
ports.c	U/I mapirane instrukcije	25
video.c	Prikaz teksta na ekranu	58
hello.c	Pozdravna poruka	10
testvideo.c	Test video podsistema	
keyboard.c	Drajver tastature	147
testkbd.c	Test tastature	
interrupt.c	Implementacija prekida	145
testinter.c	Test prekida	
diskio.c	Direktni pristup IDE HD	47
testdisk.c	Test diska	
files.c	Datotečni sistem	278
testfiles.c	Test datoteka	
memory.c	Linearna i stranična memorija	104
testmemory.c	Test memorije	
sched.c	Raspoređivač	138
syscall.c	Sistemski pozivi	90
testproc.c	Test raspoređivača	
Ukupno linija koda jezgra		1168

Uz prosječno objašnjavanje koda od 15 linija u minuti, uz faze kompajliranja međuverzija i prikaz različitih demonstracija, standardnih 90 minuta laboratorijskih vježbi je dovoljno za cjelovit prikaz rada jezgra. U daljnjem dijelu rada reći će se nešto o realizaciji ovih podsistema uz isječke koda za kraće rutine.

B. *Boot sektor*

Ovo je jedini podprogram koji je napisan u asemblerskom jeziku. Boot sektor mora biti velik 512 bajta i završiti se signaturom. Sastoji se iz dva osnovna dijela, 16 bitnog i 32 bitnog. Šesnaest-bitni dio koda, koji se izvršava na adresi 000:7c00 najprije dva puta poziva BIOS interapt 13h. Ovom interapt pozivu se prosljeđuju parametri u registrima. Pri prvom pozivu se kontroler diska resetuje, a pri drugom pozivu se prebace sektori sa diska u RAM. To učita datoteku s jezgrom u memoriju. U drugoj polovini šesnaest-bitnog dijela se učita GDTR registar da pokazuje na tabelu koja jednostavno mapira segmente u linearno adresiranje. Nakon promjene vrijednosti CR0 registra, pređe se u zaštićeni režim rada.

Trideset-dva bitni dio boot sektora sada postavi segmentne registre, stek pointer i skače u jezgro. U boot sektoru se nalazi i tablica deskriptora, potrebna da se transformiše Intelova segmentna arhitektura u linearnu. Ona se sastoji od samo tri deskriptora, jednog praznog, jednog za kod i jednog za podatke.

C. *String rutine*

Operativni sistem se izvršava na praznoj mašini. Stoga, programeru nije na raspolaganju standardna C biblioteka. Nekoliko neophodnih funkcija za rad sa stringovima treba realizovati. Ove funkcije se mogu realizovati čisto u C jeziku. Implementirane funkcije strncmp, strncpy i itoa

D. Mapiranje I/O portova

Standardni jezik C nema ugrađen pristup U/I mapiranim periferijskim uređajima. Stoga je potrebno koristiti asemblerske rutine, no njih je lako enkapsulirati u odgovarajuće podprograme koristeći inline asembler ugrađen u GCC. Tako na primjer, za upis osmobitnog podatka koristi se sljedeća funkcija.

```
void out(unsigned short _port, unsigned char _data) {
  __asm__ ("out %%al, %%dx" : :"a" (_data), "d" (_port));
}
```

E. Video drajver

Kako bi prvi test u razvoju jezgra operativnog sistema bio ispis pozdravne poruke, sljedeći korak bi bio prikaz na tektualnom ekranu. Prvi dio jednostavnog video drajvera se bavi pozicioniranjem kursora. Pozicija kursora se čuva u varijabli cursorpos, koja sadrži 80*red+kolona. Njena vrijednost se očitava i mijenja jednostavnim funkcijama

```
void SetCursor(short row, short col) {
  cursorpos=((row*80) + col); hardwarecursor();
}
void GetCursor(short * row, short * col) {
 *row = cursorpos/80; *col=cursorpos % 80;
}
```

U rutini hardwarecursor koriste se registri na portovima 0x3D4 i 0x3D5, kojima se uključuje tekstualni kursor.

Tekstualna video memorija na PC računarima se nalazi na poznatoj apsolutnoj adresi 0xB8000. Pristup tekstualnoj memoriji je u formi matrice od 80 kolona, u formatu ASCII znak, atribut za svaki znak na ekranu. Ekran se briše ubacivanjem znaka s kodom 0 ili 32 na svaku poziciju video memorije. Ispis teksta na ekran se obavlja rutinom Print sa petljom koja ubacuje ASCII kodove poruke, preskačući atribute u video memoriji, dok rutina PrintNum realizuje ispis cijelog broja pozivajući atoi funkciju prije Print. Za vježbu se ostavlja realizacija scroll funkcija, specijalnih znakova kao što je CR/LF itd.

```
void Print(const char * msg)
{
 unsigned long i;
 unsigned char *vmem;
 vmem= (unsigned char *)0xB8000;
 vmem += cursorpos<<1; i = 0;
 while (msg[i] != 0) {
  *vmem = msg[i++]; vmem += 2;
 }
cursorpos += i;
 hardwarecursor();
}
```

F. Drajver tastature

Relativno jednostavan dio drajvera za tastaturu radi direktno sa sken kodovima. Kada je pritisnut taster (port 0x64h vraća na nultom bitu vrijednost 1), može se očitati vrijednost tastera sa porta 0x60h.

```
unsigned char GetScanCode(){
  static char code;
  if(in(0x64) & (0x01))  code = in(0x60);
  else  code = 0;
  return code;
}
```

Znatno veći dio ovog drajvera predstavlja rutina koja konvertuje sken kod u ASCII, u stanjima sa SHIFT, CTRL i običnom pritisku tastera. Koriste se tri tablice kbdnorm, kbdshift i kbdctl koje omogućavaju konverziju sken kodova u ASCII kodove.

```
char * kbdnorm = "\ 0\0331234567890-=\b\tqwertyuiop[]\n
,\0asdfghjkl;\"`\0\\zxcvbnm,./"\0\0\0 ";
```

Rutina ScanToAscii gleda da li je pritisnuti taster jedan od Shift, Caps Lock, Ctrl i Alt tastera itd, te na bazi toga uključuje ili isključuje pojedine flegove. Prema kombinaciji flegova, obavi se izbor prave tabele konverzije. Očitavanje tastera se u ovoj fazi obavlja zaposlenim čekanjem, rutinom WaitKey, a demo je program koji prikazuje na ekranu pritisnute tastere.

```
char WaitKey(){
  int keycode = 0;
  while(!keycode)  keycode = GetScanCode();
  return ScanToAscii(keycode);
}
```

Za vježbu se može dodati implementacija drugih rasporeda tastature, podrška funkcijskim tasterima i numeričkoj tastaturi.

G. Interapt drajver

Naredni zadatak je izbjeći zaposleno čekanje. U tu svrhu se koriste interapti. U zaštićenom režimu rada registar IDTR pokazuje na tabelu interapt vektora. Svaki element te tabele se sastoji od osam bajta.

```
struct IDTDescr{
  unsigned short offset_low;
  unsigned short seg;
  unsigned short flags;
  unsigned short offset_high;
};
```

Segmentni dio deskriptora je uvijek postavljen na vrijednost 8, jer je ranije memorija linearizovana, a i flegovi su postavljeni na istu vrijednost za sve interapt deskriptore. Stoga se postavlja samo vrijednost nižih i viših 16 bita ofseta.

Postavljanje početka te tabele se obavlja naredbom mašinskom instrukcijom LIDT koja je upakovana u C podprogram. Na sličan način, upakovane su i rutine za uključivanje i isključivanje interapta, na primjer

```
void enable_interrupts () {  __asm__ ("sti");  }
```

Rani interapt kontroleri na PC su postavljali vektore za obradu hardverskih zahtjeva počevši od IRQ0 na vektore interapta od 8 nadalje. To je ostala podrazumijevana konfiguracija. No od procesora 80386 ti vektori se koriste za druge svrhe, na tim adresama se generišu procesorski izuzeci. Stoga je potrebno premapirati interapt vektore, da se hardverski

zahtjevi obrađuju od vektora 32 nadalje, što je implementirano u rutini PIC_remap koja se uglavnom sastoji od out instrukcija.

Format C podprograma nije jednak formatu interapt rutine. Zato rutine koje se izvršavaju u interaptima imaju određene asemblerske instrukcije na početku i kraju kako bi imale pravilni format steka. Prva od njih je vezana za očitavanje tastature. Primljeni sken kod tastature se smiješta u kružni bafer. Rutina za očitavanje tastature preuzima znak iz tog bafera. One su implementirane kao u sljedećem dijelu koda.

```
char kbdbuffer[256];
unsigned int readchar,storedchar;
void kbdirq() {
  static unsigned char c;
 __asm__ __volatile__("pushal; ");
   c=GetScanCode();
   if (c != 0) {
    storedchar=(storedchar+1) % 256;
    kbdbuffer[storedchar]=c;
   }
  out(0x20,0x20);
 __asm__ __volatile__(" popal; leave ; iret");
}
unsigned char readintscancode() {
  if (readchar!=storedchar) {
    readchar=(readchar+1) % 256;
        return kbdbuffer[readchar];
  }
  else  return 0;
}
```

Drugi implementirani interapt je tajmer interapt. Njegov zadatak je jednostavan, uvećava brojač za jedan svakih 10 ms.

```
void Clock() {
 __asm__ __volatile__("pushal");
   elapsedtime++;
   out(0x20,0x20);
 __asm__ __volatile__(" popal ; leave ; iret");
}
```

Inicijalizacija interapt sistema postavlja interapt tabelu i aktivira vektore ova dva interapta, dok demo program prikazuje kucanje tastature dok vrijeme prolazi. Za vježbu se može dodati kritična sekcija za kontrolu pristupa baferu tastature, obrada interapta koji generiše serijski port, ili implementacija nekih procesorskih izuzetaka.

H. Drajver diska

Iako je njihova upotreba u novijim PC računarima sve rjeđa, kao uređaj spoljne memorije korišten je IDE hard disk, jer je to najjednostavnija vrsta diska za pristup na niskom nivou. Kod ove vrste diskova koriste se portovi 0x1f1 do 0x1f7 na koje se može poslati LBA ili CHS adresa prvog sektora u grupi sektora koji se želi prebaciti, te koliko se sektora prebacuje. Slanjem vrijednosti na port 0x1f7 se aktivira komanda kojom započinje prijenos u jednom ili drugom pravcu. Nakon toga se mogu očitavati vrijednosti ili upisivati. Rutine koje pišu i čitaju blokove s diska su vrlo slične, pri čemu rutina za čitanje s diska izgleda ovako.

```
int ReadHDBlock( void* buf, int first, int num ){
    int i;
    unsigned short* dat = (unsigned short*) buf;
    while(num > 0 ){
     while( in(0x1f7) & 0x80 ){}  //wait for ready
     out(0x3f6,2);      //no interrupts
     out(0x1f2,1);      //sector count
     out( 0x1f3,first & 0xff    );     //sector #
     out( 0x1f4,(first>>8) & 0xff   );
     out( 0x1f5,(first>>16) & 0xff  );
     out( 0x1f6,((first>>24) & 0x0f)|0xe0  );
     out(0x1f7,0x20);                  //start read
     while( in(0x1f7) & 0x80 ) {}  //wait for ready
     while( (in(0x1f7) & 0x08) == 0 ) {}  //wait for data
     for(i=0;i<256;++i, dat +=1)
        *dat = inw(0x1f0);;
     --num; ++first;
    }
    return 1; }
```

Demo primjer ovog podsistema prikazuje na ekranu bajtove na disku gdje je smješteno jezgro operativnog sistema.

I. Datotečni sistem

Datotečni sistem je dizajniran da bude što jednostavniji. Stoga je implementirana direktorijska struktura na jednom nivou, sa kontinualnom alokacijom datoteka na disku. Ipak, strukture podataka koje opisuju elemente direktorija pored podataka o imenu i poziciji datoteke na disku, posjeduju i polje directory koje omogućuje proširenje na direktorije s dva ili više nivoa. Struktura direntry sadrži podatke o svim datotekama na disku, a openentry o otvorenim datotekama. Ove strukture su poredane u tabele dirtable i opentable.

Izmjene sadržaja datoteke rezultuju upisom elementa direktorija na disk, što radi procedura SaveDirEntry koja iz rednog broja datoteke u direktorijskoj strukturi izračuna poziciju odgovarajućeg elementa direktorijske strukture na disku i upiše blok.

```
int SaveDirEntry(unsigned int num) {
   if (num<0 || num>=maxfiles)   return -EINVALID;
   int dirsperblock=512/sizeof(struct direntry);
   int relposondisk=num/dirsperblock;
   int direntrytowrite= relposondisk *dirsperblock;
   WriteHDBlock( &(dirtable[direntrytowrite]),
    dirtableondisk+ relposondisk,1 );
   return OK;
}
```

Otvaranje datoteke kopira element s proslijeđenim rednim brojem u tabeli direktorija u slobodno mjesto u tabeli otvorenih datoteka uz punjenje elemenata tabele otvorenih datoteka koji ne postoje u tabeli direktorija. Druga rutina OpenFile traži datoteku s navedenim imenom i poziva OpenFileByNumber.

Direktorijska struktura poznaje kao vrste datoteke slobodno područje, alociranu regularnu datoteku i datoteku koja je tip direktorija. Brisanje datoteke je prosto pretvaranje njenog sadržaja direktorija u slobodnu zonu, kao što se vidi u dijelu koda.

```
int DeleteFile(char * name) {
 int i;
 struct direntry * currentdirentry;
 currentdirentry=dirtable;
 for (i=0;i<maxfiles;i++,currentdirentry++) {
  if(currentdirentry->entrytype==REGULARFILE) {
   if(!strncmp( currentdirentry->name,name, NAMELEN) {
      currentdirentry->entrytype=FREEZONE;
      return SaveDirEntry(i);
    }
   }
  }
 return -EFILENOTFOUND;
}
```

Rutina za kreiranje datoteke, zbog kontinualne alokacije kreiranje datoteke treba u parametru da ima i njenu maksimalnu veličinu. Nakon toga se pronalazi algoritmom najboljeg uklapanja (best fit) slobodni blok na disku među ranije alociranim datotekama. Zatvaranje datoteke prosto promijeni statusno polje u tabeli otvorenih datoteka. Pozicioniranje je takođe jednostavno.

```
int CloseFile(int fd) {
  if (fd<0 || fd>=maxopenfiles) return -EINVALID;
  opentable[fd].status=CLOSED;
  return OK;
}
```

Pisanje u datoteku započinje, nalaženjem pozicije njenih blokova na disku, a zatim se prvi od njih učita u bafer. Nakon toga se podaci prebacuju iz korisničkog bafera u bafer disk bloka i kada pokazivač dođe na kraj bafera, blok se upiše na disk. Ovdje se mora paziti još da snimljeni blokovi ne pređu limit predviđen za tu datoteku. Ako je upisano više od trenutne dužine, ažurira se odgovarajući element direktorijske strukture na novu dužinu. Čitanje datoteke takođe izračuna iz pozicije navedene u otvorenoj datoteci poziciju bloka na disku. Nakon toga se poziva funkcija za čitanje grupe sektora sa diska u memoriju.

Inicijalizacija postavlja početne vrijednosti potrebnih varijable i učitava u memoriju strukturu direktorija, dok demo uključuje upise, brisanja, dodavanja i prikaze nekoliko kreiranih datoteka. Za vježbu se mogu dodati rutine za pretraživanje direktorija i modifikovati počeci particija.

J. Upravljanje memorijom

U ovom operativnom sistemu, memorija između 0 i 640 K je rezervisana za jezgro i njegove tablice, dok se za korisničke aplikacije alocira prostor iznad 1M. U modulu za upravljanje memorijom implementirane su dvije strategije, za fizičku memoriju i virtualnu memoriju. Fizički blok memorije je predstavljen elementom memtable. Ovdje je u pitanju niz jednostavnih struktura koje sadrže početnu adresu i bit polja koja sadrže dužinu i zauzetost. Inicijalizacija memorijskog podsistema postavlja tablicu elemenata fizičke memorije na jedan blok od 128 megabajta, počevši od adrese 0x100000.

AllocMemory će onda dodijeliti prvi slobodan blok date Fizička memorija se dodjeljuje kontinualnom alokacijom, first fit algoritam, neiskorišteni dio šupljine postaje nova šupljina.

```
void * AllocMemory(int size) {
 int i;
 if (maxused>=TABLELEN)
   return (void *) EOUTMEMORY;
 for (i=0; i<=maxused; i++) {
  if ((memtable[i].length>=size) &&
     (memtable[i].busy==0)) {
       memtable[i].busy=1;
       if (memtable[i].length>size) {
       maxused++;   memtable[maxused].busy=0;
       memtable[maxused].startaddress=
         memtable[i].startaddress+size;
       memtable[maxused].length=
          memtable[i].length-size;
       memtable[i].length=size;
    }
    return (void *) memtable[i].startaddress;
  }
 }
  return (void *) EOUTMEMORY;
}
```

Rutina za oslobađanje zauzetog bloka fizičke memorije brine o ukrupnjavanju susjednih šupljina, gledajući sva četiri slučaja položaja postojećih i nove šupljine.

Procesi mogu da vide memoriju drugačije, te se takođe ilustruje mapiranje virtualne memorije. Rutine ActivatePaging i LinearMode mijenjaju registre CR0 i CR3 i time mijenjaju režim straničenja na uključen ili isključen. Koristi se straničenje na dva nivoa. Adresni prostor svakog procesa od 4 megabajta se vidi kao da je između adresa 0 i 4M, a premapira se na odgovarajuću fizičku adresu iznad 1M. Kako Intelovi procesori koriste straničenje na 2 nivoa, potrebne su dvije tablice, jedna koja predstavlja imenik stranica (i ima samo jedan iskorišteni element) i druga koja predstavlja tablicu stranica (sa 1024 iskorištena elementa). Prvi megabajt u toj tablici se mapira svim procesima jednako, dok se preostala tri megabajta virtualnog prostora mapira u memorijsko područje odvojeno za svaki proces.

```
void MapVirtualSpace(void * tableaddr,void * useraddr) {
  unsigned int *pagedir, *first_page_table, i,address;
  page_directory=tableaddr;  address=0;
  for(i = 0; i < 1024; i++)  pagedir[i] = 2;
  first_page_table = pagedir + 1024;
  for(i = 0; i < 1024; i++) {
    first_page_table[i] = address | 3;     address += 4096;
    if (i==255) address=(unsigned int) useraddr;
  }
  pagedir[0] = (unsigned int)first_page_table ] |= 3;
```

K. Procesi

Radi jednostavnosti, raspoređivač je kooperativnog tipa, što znači da aplikacije ili niti u jezgru moraju povremeno zvati sistemske pozive koji obavljaju raspoređivanje. Za čuvanje ranijeg konteksta procesa ili sistemske niti koristi se jmp_buf struktura, u koju se sačuva stanje registara i pripremi skok. Naredbom setjmp se pripremi povratna točka a naredbom longjmp se skače na neku od postavljenih povratnih tačaka.

U strukturi task_t se nalazi kontrolni blok procesa. On se sastoji od statusa procesa, pokazivača na njegovu memoriju, stranice sadržaja registara iz setjmp. Kontrolni blokovi procesa

su poredani u niz. Raspoređivač radi po Round robin principu u kooperativnoj verziji. Kada se pozove schedule funkcija sa setjmp se sačuva stanje trenutno izvršavanog procesa, promijeni se tablica memorijskih strania i zatim se s longjmp prelazi na naredni proces.

```
void schedule(void) {
        unsigned prev;tatic int cr0;
        prev = current;
        do {
          current++; // round-robin switch to next task
          if(current >= NUM_TASKS)current = 0;
        } while (g_tasks[current].state!=READY);
        idlecpu=(prev==current==0);
        if(setjmp(g_tasks[prev].regcontext) != 0) return;
        ActivatePaging ( g_tasks[current].pagetable);
        longjmp(g_tasks[current].regcontext, 1);
}
```

Pokretanje novog procesa alocira memoriju od 3M, postavi registre, premapira memoriju i ako nije sistemska nit učita datoteku s procesom, i doda vrijednosti u tablicu procesa. Završetak procesa oslobađa njegovu memoriju, i poziva raspoređivač.

```
void Terminate(int pid) {
  if (pid >=0 && pid < NUM_TASKS ) {
         setprocstate(pid,DELETED);
         LinearMode();
         FreeMemory(g_tasks[pid].memoryblock);
         schedule();
        }
}
```

L. Sistemski pozivi

Pošto je raspoređivač kooperativnog tipa, u sistemske pozive se pored osnovne funkcije koju izvršava doda i poziv raspoređivača, kao u sljedećem primjeru.

```
int _OpenFile(char * name) {int i=OpenFile(name); schedule(); return i;}
```

IV. PROMJENE U NASTAVNOM PROCESU

U školskoj 2014/2015 godini na ETF Sarajevo, je uvedena vježba praćenja i kompajliranja minimalističkog kernela. Nakon objašnjavanja svake grupe slojeva i pisanja različitih verzija main programa gcc kompajlerom i nasm asemblerom se prevedu do tada obrađeni slojevi jezgra i izvrši pod QEMU, a zatim koristeći disketu prebaci na jedan izdvojeni PC sa IDE diskom na izvršavanje na realnom hardveru. Ranijih godina na kursu operativnih sistema kao primjer izvornog koda su prikazivani dijelovi starijih ili novijih verzija sistema Linux ili Minix, međutim vrlo mali broj studenata je bio sposoban da prati izvorni kod i dijelova ovih sistema, dok se vremena za kompletan opis jednostavno nije imalo. Studenti su u istovremeno prošli i jedan kurs asemblerskog programiranja i arhitekture PC računara. Eksperiment je pokazao da se ovo novo jezgro može u potpunosti opisati za jedan blok laboratorijskih vježbi od 90 minuta, koji se izvodi po završetku teoretskog dijela kursa operativnih sistema, a da je pri tome zadržana pažnja i relativna razumljivost teme. Ipak, ako se želi inkorporirati više samostalnog rada studenata (npr. pisanje vlastitih izmjena u jezgru), treba se raditi više sedmica.

V. ZAKLJUČAK

Minimalistički operativni sistemi ne pretenduju na upotrebu u realnom svijetu, ali kao obrazovni alat za prikaz principa operativnih sistema imaju veliki značaj. Primjenom najjednostavnijih algoritama postignuto je jezgro dovoljno kratko da izvorni kod operativnog sistema postigne svoju svrhu: da se pokaže način rada operativnih sistema i sa praktične strane, a da se ne potroši mnogo vremena na uštrb teoretskog pristupa.

LITERATURA

[1] A. S.Tanenbaum, and A. S. Woodhull. "The Minix Book–Operating Systems." (2006).

[2] D. Comer, "Operating System Design: The XINU Approach, 1984." Bell Telephone Laboratories, Incorporated

[3] T. Fossum, "PC-XINU features and installation." ACM SIGOPS Operating Systems Review 21.3 (1987)

[4] C. Wayne., S. J. Procter, and T. E. Anderson. "The Nachos instructional operating system." Proceedings of the USENIX Winter 1993 Conference Proceedings.

[5] R. Cox, M. F. Kaashoek, and R. Morris. "Xv6, a simple Unix-like teaching operating system." 2013-09-05]. http://pdos. csail. mit. edu/6.828/2012/xv6. html (2011).

[6] Z. Jiong, and T. Jarvi. "Adopting and Commenting the Old Kernel Source Code for Education." Linux Symposium. 2005.

[7] Qu, Bo, and Z. Wu. "Design and Implementation of Tiny Educational OS." Recent Advances in Computer Science and Information Engineering. Springer Berlin Heidelberg, 2012. 437-442.

[8] S. Diomidis. "A tale of four kernels." Proceedings of the 30th international conference on Software engineering. ACM, 2008.

[9] H. David, J..K. Hollingsworth, and B. Bhattacharjee. "Running on the bare metal with GeekOS." ACM SIGCSE Bulletin. Vol. 36. No. 1. ACM, 2004.

[10] M. Black, "Build an Operating System from Scratch: A Project for an Introductory Operating Systems Course,", Proceedings of the 40th Technical Symposium on Computer Science Education, SIGCSE 2009

ABSTRACT (HEADING 5)

A Operating systems courses are often organized according to three basic approaches: user-oriented courses, theoretical OS algorithm courses or by analyzing and modifying the kernel source code. While the third approach is the most illustrative, it requires big ammount of time at the expense of the width of topics analysis of different operating systems algorithms. One solution for this problem are small kernels, rather than analyzing those that are in wide use. In this paper we will present a simple kernel, with a minimal amount of source code that can be fully described during the teaching 90 minutes, but still big enough to boot on a typical PC, and that includes the basic functions of the core: work with the screen, keyboard, disk, file system, memory, processes and threads..

Education Minimalistic Operating System Kernel

Samir Ribić, Adnan Salihbegović