SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1781

UPRAVLJANJE PROCESIMA U JEDNOSTAVNIM OPERACIJSKIM SUSTAVIMA

Antonio Arbutina

Zagreb, veljača 2009.

SADRŽAJ

1. UVOD 1

2. OSNOVE ARHITEKTURE I386 2

2.1 ZAŠTITA MEMORIJSKOG PROSTORA 2
2.1.1 Zaštita preko zaštitnih prstena 2
2.1.2 Zaštita preko segmentacije 3
2.1.3 Zaštita preko straničenja 4
2.2 REGISTRI SUSTAVA 6
2.3 TABLICE SUSTAVA 8
2.3.1 Opća tablica opisnika - GDT 9
2.3.2 Tablica opisnika prekida - IDT 10
2.3.2.1 Podjela iznimaka arhitekture i386 11
2.3.2.2 Izgled stoga i format koda pogreške pri pojavi iznimke 11
2.3.2.3 Upravljanje prekidima 12
2.3.2.4 Ostvarenje poziva jezgre pomoću prekida 12
2.3.3 Lokalna tablica opisnika - LDT 12
2.4 SEGMENT STANJA ZADATKA - TSS 13
2.5 BROJILO SUSTAVA - PIT 14
2.6 PROGRAMSKI NAČINI RASPOREĐIVANJA 15

3. MYOS: STRUKTURA IZVORNOG KODA 16

3.1 SHEMA STRUKTURE DIREKTORIJA I DATOTEKA IZVORNOG KODA 16
3.2 ULAZNA DATOTEKA START.ASM 17
3.3 POČETAK KODA JEZGRE OPERACIJSKOG SUSTAVA 23
3.3.1 Pokretanje ljuske 23
3.3.2 Postavljanje operacijskog sustava 24
3.3.3 GDT tablica 25
3.3.4 Prebacivanje u korisnički način rada 26
3.3.5 Zadatak čekanja sustava 27
3.4 PRINCIP RADA LJUSKE OPERACIJSKOG SUSTAVA 27
3.4.1 Datoteka keyb.c 27
3.4.2 Naredbe i parsiranje naredbi 28
3.4.3 Povezanost ljuske sa raspoređivačem poslova 37
3.5 PRINCIP RADA KORIŠTENIH TIPOVA RASPOREĐIVAČA POSLOVA 38
3.5.1 Upravljanje po jednakim vremenskim intervalima enja zadataka 38
3.5.2 Upravljanje po redu prispijeća 38
3.5.3 Ostali načini korištenjem ljuske 38
3.6 MEMORIJSKA MAPA MYOS-A – PROSTOR ZA STOG, INSTRUKCIJE I PODATKE 38

4. RASPOREĐIVAČ POSLOVA 40

4.1 KOMUNIKACIJA MEĐU ZADACIMA I JEZGROM 40
4.1.1 Memorijski prostor zadataka 40
4.1.2 Izgled samog programskog koda zadataka MyOS-a 42
4.1.3 TASK_STRUCT struktura raspoređivača poslova i TASK0 zadatak 45
4.1.4 Stanja zadataka i način komunikacije sa jezgrom 47
4.2 OSTVARENJE ALGORITAMA U MYOS-U 48
4.2.1 Pomoćne funkcije jezgre operacijskog sustava korištene za raspoređivanje zadataka 48
4.2.1.1 New_task() funkcija 48
4.2.1.2 Run_task() funkcija 50
4.2.1.3 Finish_task() funkcija 51
4.2.2 Raspoređivač poslova prema jednakim intervalima izvođenja zadataka 52
4.2.3 Raspoređivač poslova po redu prispijeća zadataka 54
4.2.4 Mogućnost ostvarenja ostalih algoritama raspoređivača poslova u MyOS-u 56
4.3 POBOLJŠANJE OSTVARENJA ALGORITAMA RASPOREĐIVAČA POSLOVA 58

5. PRIMJERI KORIŠTENJA LJUSKE I RASPOREĐIVAČA POSLOVA 59
6. ZAKLJUČAK 60
7. POPIS KORIŠTENE LITERATURE I POMOĆI 61
8. SAŽETAK 62

1. Uvod

Kada bi vam netko rekao da napravite operacijski sustav tek toliko da znate „kako se to radi“ i za akademske svrhe, što bi značilo, u svrhu istraživanja, koju biste arhitekturu odabrali? A kada bi to trebali napraviti na i386, da li biste to smatrali zanimljivim ili teškim? Bilo kako bilo, ovaj rad je ostvario osnovne dijelove operacijskog sustava koristeći tu arhitekturu, koja je dostupna većini korisnika kućnih računala diljem svijeta.

Ostvarenje ovog diplomskog zadatka uvelike se oslanja na raniji seminarski rad *Alati za izgradnju operacijskog sustava*[2]. U tom radu može se pronaći objašnjenje pojmova koje ovdje nećemo objašnjavati. Cilj ovog diplomskog rada je izvedba konkretnog raspoređivača poslova, a uporabom i386 arhitekture računala. Prije toga je svakako potrebno upoznati se sa svim detaljima same arhitekture koji su potrebni za osnovan rad operacijskog sustava. Kao što će biti dalje objašnjeno, cilj je ostvariti tri osnovne zadaće operacijskog sustava:

- *upravljanje procesima,*
- *upravljanje prekidima,*
- *pozivi jezgre.*

Kako bi mogli jasnije shvatiti samo ostvarenje u programskom jeziku C, nužno je razumjeti međusobnu povezanost raznih dijelova arhitekture, povezanost dijelova jezgre operacijskog sustava te kako je ostvaren prijelaz između stanja raspoređivača poslova unutar same jezgre. Pojam procesa i pojam zadatka se u ovom radu koriste kao sinonimi. Većinom će umjesto upravitelja procesa biti korišten pojam raspoređivač poslova, odnosno zadataka.

Stoga je potrebno pozabaviti se samom arhitekturom i386, odnosno kako je zaštićen memorijski prostor u sklopu te arhitekture, koje specifične razine zaštite posjeduje te kako sa stajališta programera ostvariti prije navedene zadaće, a to će biti objašnjeno u nastavku ovog rada. Važno je napomenuti kako ćemo samo objasniti osnove arhitekture nužne za ostvarenje cilja našeg projekta, odnosno jednostavnog operacijskog sustava. U samom kodu C jezika koristi se asemblerski kod unutar C koda (engl. *inline assembly)*, a osnove su objašnjene u *GCC inline assembly HOW TO[7].*

Pojmovi koji se koriste te sve slike i sheme koje prikazuju arhitekturu preuzete su iz Intelovog priručnika za i386 arhitekturu, dio 3A[1].

2. Osnove arhitekture i386

2.1 Zaštita memorijskog prostora

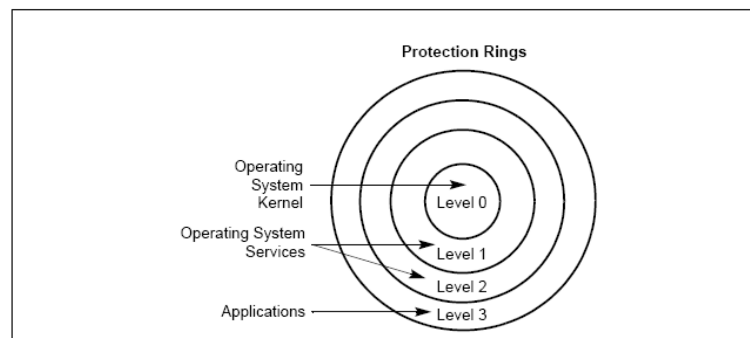
U ovom dijelu objasnit ćemo na koje načine je moguće zaštiti memorijski prostor i386. Obratit ćemo posebnu pažnju registrima i instrukcijama koje su potrebne kako bi se postavile tablice sustava. Također će biti objašnjeni zaštitni prstenovi koje i386 koristi (engl. *protection rings*).

2.1.1 Zaštita preko zaštitnih prstena

Kao zaštitu i386 koristi tri dvo-bitna polja koja sadrže informacije o trenutnom prstenu u kojem se nalazimo, prstenu u koji želimo preći na neki način, ili prstenu u kojem se određeni segment memorije nalazi.

Dakle, radi se o trenutnoj razini privilegija – CPL (engl. *current privilege level)*, razini zaštite trenutnog segmenta memorije u kojoj se instrukcije izvode odnosno segmentu koda ili instrukcija (engl. *code segment)*. Razina privilegije opisnika – DPL (engl. *descriptor privilege level)*, koristi se kako bi se u opću tablicu opisnika - GDT (engl. *global descriptor table*) ili lokalnoj tablici opisnika - LDT (engl. *local descriptor table*), svakome segmentu pridijelila razina zaštite odnosno odredilo kojem zaštitnom prstenu pripada dotični segment memorije. Zahtijevana razina privilegija - RPL (engl. *requested privilege level),* koristi se u izbornicima segmenta (engl. *segment selectors*), odnosno također u registrima koji sadrže izbornike određenih segmenata memorije, a mehanizam tih izbornika bit će objašnjen kasnije kad će biti govora o tablicama sustava, GDT i LDT, ali i u nastavku gdje se objašnjava segmentacija (engl. *segmentation*) memorije.

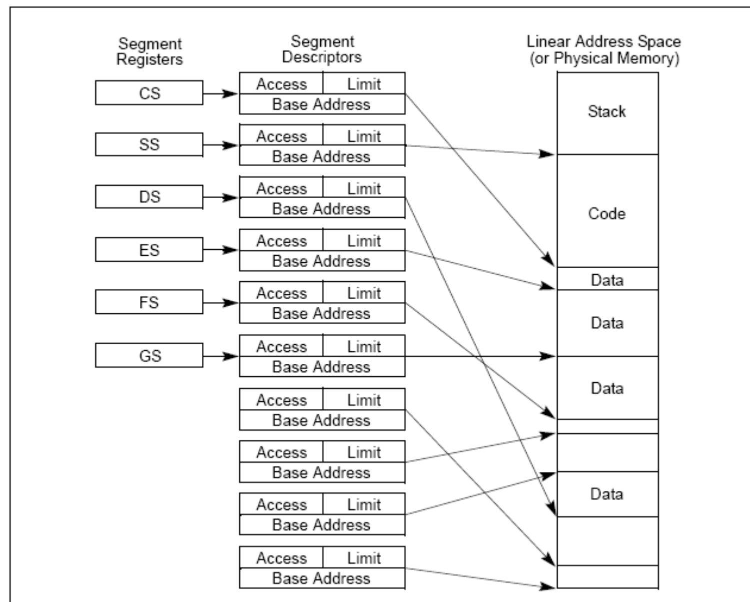
Slika 2.1-1 prikazuje kako izgledaju zaštitni prstenovi i čemu služe. Obično se u današnjoj upotrebi ne koriste prstenovi 1 i 2, nego samo 0 i 3, odnosno prsten 0 koristi se za jezgrin način rada (engl. *kernel mode)*, gdje jezgra operacijskog sustava jedina ima pristup, a prsten 3 se koristi za korisničke procese, aplikacije itd.



Slika 2.1-1 Zaštitni prstenovi

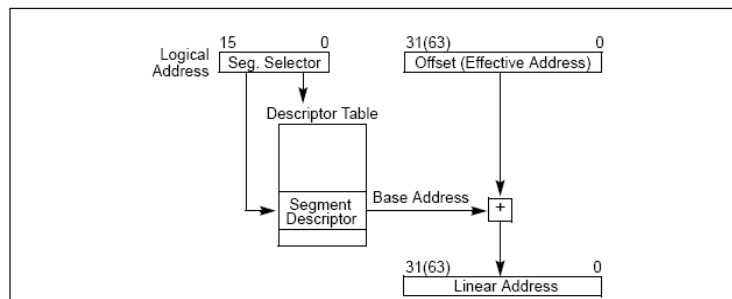
U nastavku ćemo objasniti način rada mehanizma za segmentaciju, a kasnije i straničenje, što predstavlja dva moguća načina zaštite memorije u arhitekturi i386, a budući da ova arhitektura posjeduje dva načina zaštite memorije u tom je pogledu jedinstvena.

2.1.2 Zaštita preko segmentacije



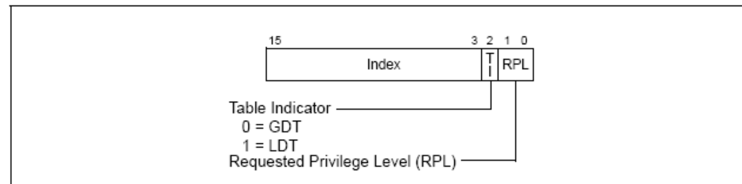
Slika 2.1-2 Segmentacija

Slika 2.1-2 prikazuje kako registri segmenata, sadrže indeks koji jednoznačno određuje opisnik segmenta koji opet jednoznačno određuje segment memorije. U ovoj i svim narednim slikama adrese memorijskih lokacija rastu odozdo prema gore. Registar segmenta, sadrži izbornik segmenta, čiji oblik će biti objašnjen nešto kasnije. Za sada je dovoljno reći da registar `CS` određuje trenutni segment instrukcija, `SS` određuje segment stoga, `DS`, `ES`, `FS`, `GS` određuju podatkovni segment, odnosno dijelove memorije određene za instrukcije, stog i podatke. No, sadržaj ovih registara se može mijenjati tijekom izvođenja instrukcija ili promjene zaštitnog prstena. To također znači da to nije jedini način kako se pristupa segmentima memorije, što znači da nije nužno uvijek da ovi registri sadrže izbornik segmenta, već se izbornike segmenata može koristiti i izravno. U poglavlju 2.4 će biti objašnjena konkretna primjena opisanog načina pristupa segmentima.



Slika 2.1-3 Prijevod logičke adrese u linearnu adresu

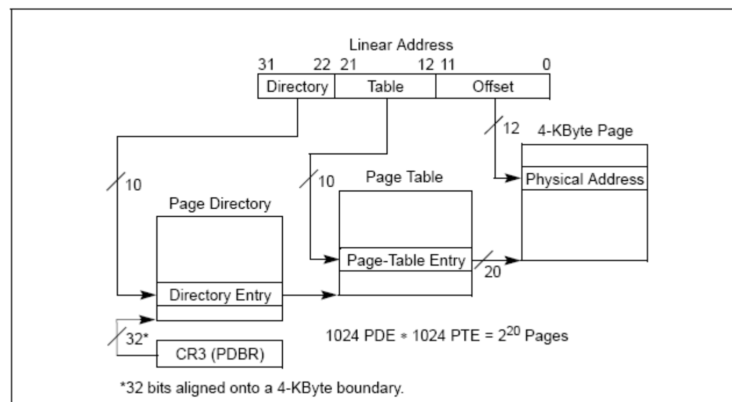
Slika 2.1-3 objašnjava kako pomoću 16-bitovnog izbornika segmenta odabiremo određenu tablicu opisnika, GDT ili IDT. U njoj se pomoću indeksa koji se nalazi u opisniku segmenta određuje početna adresa (engl. *base address*) segmenta memorije, koja zajedno sa efektivnom adresom memorije koju trenutno koristimo pretvara u linearnu adresu memorije, odnosno i fizičku adresu, ako se ne koristi straničenje (engl. *paging*).



Slika 2.1-4 Izbornik segmenta

Slika 2.1-4 prikazuje kako konkretno izgleda izbornik segmenta. Sastoji se od indeksa izbornika koji se nalazi u tablici izbornika, a bit 2 određuje TI (engl. *table indicator)* polje, koje nam govori gdje se nalazi opisnik segmenta, ako se nalazi u GDT tablici, bit je obrisan, ako se radi o trenutnoj LDT tablici, bit je postavljen. Tražena razina privilegija(RPL) je dvobitovno polje koje sadrži vrijednosti od 0-3, a opisuje kojem zaštitnom prstenu pripada dotični prsten koji izbornik određuje. Primijetimo da je polje za indeks duljine 13 bita - što daje mogućnost za pohranu 2^{13} opisnika segmenata, odnosno 8192 moguća opisnika u tablici bilo GDT i/ili trenutnoj LDT tablici.

2.1.3 Zaštita preko straničenja



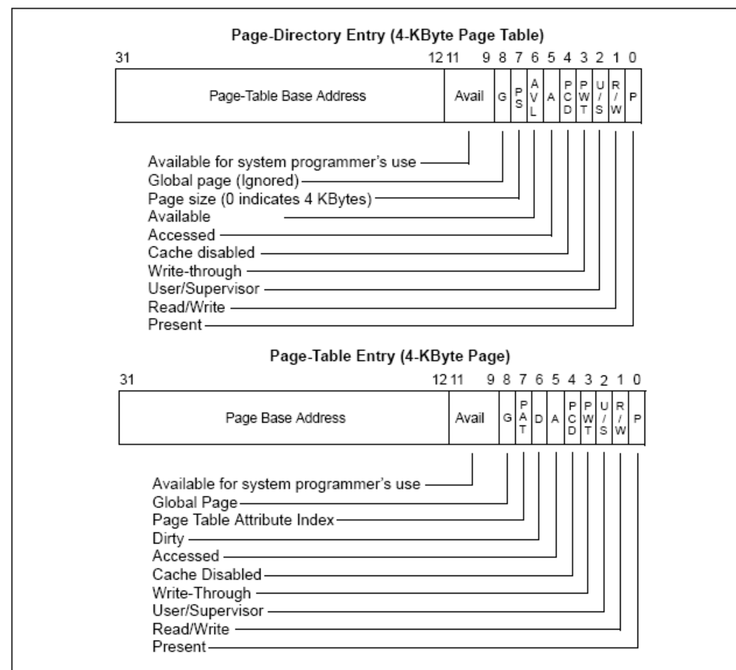
Slika 2.1-5 Prijevod linearne u fizičku adresu

Slika 2.1-5 prikazuje kako funkcionira mehanizam straničenja (engl. *paging*) kada je u uporabi straničenje veličine stranice 4kB. Arhitektura omogućava straničenje stranicama veličine 4kB, 2MB i 4MB. U nastavku je prikazan najsloženiji način i najčešće korišteni način, a to je straničenje stranicama 4kB.

Logička adresa generirana u programu prvo se prevede u linearnu adresu, prema slici 2.1-7. Linearna 32-bitna adresa dijeli se na tri dijela:

- direktorij stranica (engl. *page directory*), bitovi 22-31,
- tablicu stranica (engl. *page table*), bitovi 12-21,
- odmak unutar stranice *(engl. offset)*, bitovi od 0-11.

Veličina stranice je 2^{12} B, odnosno 4kB za jednu stranicu. Najveći broj stranica koji zadatak u opisanoj 32-bitovnoj arhitekturi može imati je 2^{20}, $2^{10} * 2^{10}$ (mogući broj zapisa u direktoriju puta mogući broj zapisa u tablicama stranica), odnosno 2 MB stranica. Ukupno se može koristiti $2^{20}*2^{12} = 2^{32} = 4$ GB memorije.



Slika 2.1-6 Format direktorija stranica i tablica stranica

Slika 2.1-6 prikazuje izgled pojedinih stavki koje unosimo u direktorij stranica, ali i stavki tablice stranica koje sadrže popis pojedinih stranica. Primijetimo da se na razini svake stranice ili direktorija može postaviti ograničenje za pisanje, čitanje, te tko ima pravo pristupiti pojedinoj stranici, odnosno direktoriju, tj. tko pristupa stranici ovisno o jezgrinom načinu rada ili korisničkom načinu rada.

Kad se upotrebljava straničenje, sustav za upravljanje memorijom (engl. *memory management system*) ima više mogućnosti za korištenje straničenja. Na primjer, može koristiti jedan direktorij za straničenje za sve programe i procese, jedan direktorij za svaki proces, ili kombinaciju tih dvaju načina.

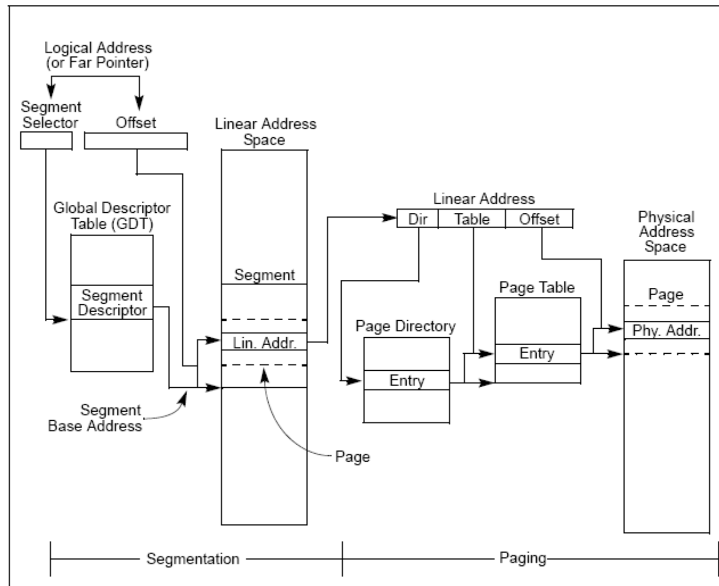
Ako procesor generira grešku u straničenju (engl. *page fault exception*), operacijski sustav općenito mora izvesti sljedeće operacije:

1. Kopirati stranicu pohranjenu na disku u fizičku memoriju.
2. Učitati adresu stranice u tablicu stranica ili direktorija postaviti zastavicu koja označava da je stranica prisutna (engl. *present*) u fizičkoj memoriji. Druge zastavice, kao što su prljava stranica - D (engl. *dirty*) i stranica kojoj je pristupljeno - A (engl. *accessed*), isto tako mogu biti postavljene u ovom koraku.
3. Prekinuti valjanost trenutne vrijednosti tablice stranica.
4. Vratiti se iz rukovatelja iznimke straničenja (engl. *page fault handler*), kako bi se vratili u normalno izvođenje prekinutog programa ili procesa, koji zatim nastavlja raditi sa stranicom koja je sada prisutna u memoriji.

Kada je straničenje omogućeno, procesor prvo provjerava zaštitu segmentacijom, a tek onda provjerava zaštitu straničenjem. Ako procesor utvrdi povredu na bilo kojem od ovih zaštita, pristup memoriji je zapriječen i procesor dojavljuje iznimku.

Ako se iznimka dogodila na segmentacijskoj zaštiti, tada se ne dojavljuje iznimka straničenja. Zaštita na razini straničenja ne može se upotrijebiti kako bi se onesposobilo zaštitu na razini segmentacije. Na primjer, segment instrukcija po definiciji je zabranjen za

pisanje. Ako je segment instrukcija u straničenju, tj. podijeljen na stranice, ako postavimo zastavicu za pisanje na stranice tog segmenta tada te stranice ipak neće biti dostupne za pisanje, već će se u slučaju pisanja dogoditi iznimka na razini segmentacije.



Slika 2.1-7 Segmentacija i straničenje

Slika 2.1-7 daje kratak pregled kako funkcionira sustav segmentacije i straničenja u kombinaciji.

Napomena: u ovom diplomskom radu nije ostvarena zaštita memorijskog prostora straničenjem.

2.2 Registri sustava

System Table Registers

	47(79) 16	15 0
GDTR	32(64)-bit Linear Base Address	16-Bit Table Limit
IDTR	32(64)-bit Linear Base Address	16-Bit Table Limit

	System Segment Registers (15 0)	Segment Descriptor Registers (Automatically Loaded)		Attributes
Task Register	Seg. Sel.	32(64)-bit Linear Base Address	Segment Limit	
LDTR	Seg. Sel.	32(64)-bit Linear Base Address	Segment Limit	

Slika 2.2-1 Registri sustava

Arhitektura i386 općenito u zaštićenom načinu rada (engl. *protected mode)* općenito koristi 3 tablice. GDT se koristi za jezgrin način rada (engl. *kernel mode),* gdje jezgra operacijskog sustava jedina ima pristup.

LDT je tablica koja je vezana uz trenutni zadatak/proces koji se izvodi. Tablica opisnika prekida - IDT (engl. *interrupt descriptor table*) je tablica koja se koristi za rukovanje prekidima (engl. *interrupt handling).*

Postoje odgovarajući registri sustava koje moramo napuniti vrijednostima ako želimo da procesor koristi tablice koje smo mi postavili negdje u memoriji. Za tu svrhu su određeni posebni registri sustava (engl. *system registers*) prema slici 2.2-1 .

`GDTR` registar je 48-bitni registar koji sadrži poseban pokazivač (engl. *pseudo selector*), koji se sastoji od 32 bita adrese i 16 bitova opsega GDT tablice u bajtovima.

IDTR registar je 48-bitni koji sadrži poseban pokazivač. 32 bita za adresu, te 16 za opseg IDT tablice.

TR (engl. *task register*) sadrži izbornik segmenta trenutnog zadatka odnosno njegove TSS strukture o kojoj će biti govora u poglavlju 2.4.

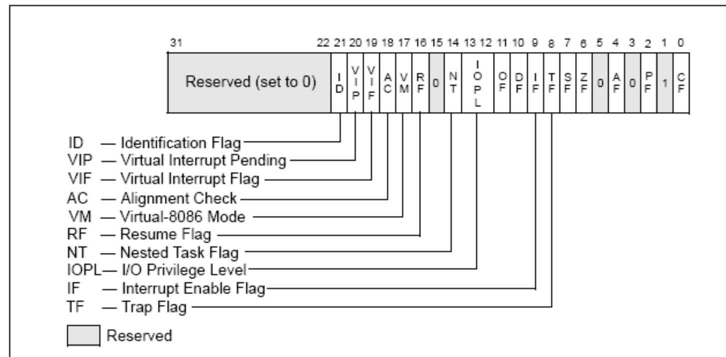
LDTR registar sadrži izbornik segmenta lokalne tablice opisnika trenutnog zadatka čiji se opisnik nalazi u GDT tablici što je objašnjeno u poglavlju 2.3.3.

GDTR i IDTR registri se uobičajeno pune samo jednom vrijednostima, na početku rada operacijskog sustava posebnim instrukcijama:

```
LGDT [poseban pokazivač]
LIDT [poseban pokazivač]
```

Registri TR i LDTR se pune vrijednostima ovisno o tome na koji način koristimo ugrađeni mehanizam višezadaćnog sustava (engl. *multitasking*). Oni se pune samo izbornicima segmenta, a ne posebnim pokazivačima kao kod GDTR i IDTR.

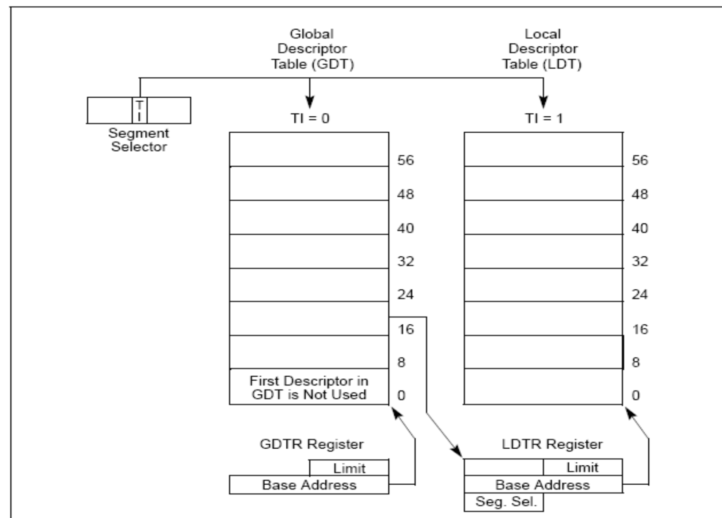
```
LIDT [izbornik segmenta]
LTR [izbornik segmenta]
```



Slika 2.2-2 Registar zastavica EFLAGS

Na slici 2.2-2 nalazi se registar EFLAGS, registar zastavica. Iako nije registar sustava, potreban je u objašnjenju rada tablica sustava u poglavlju 2.3.

2.3 Tablice sustava

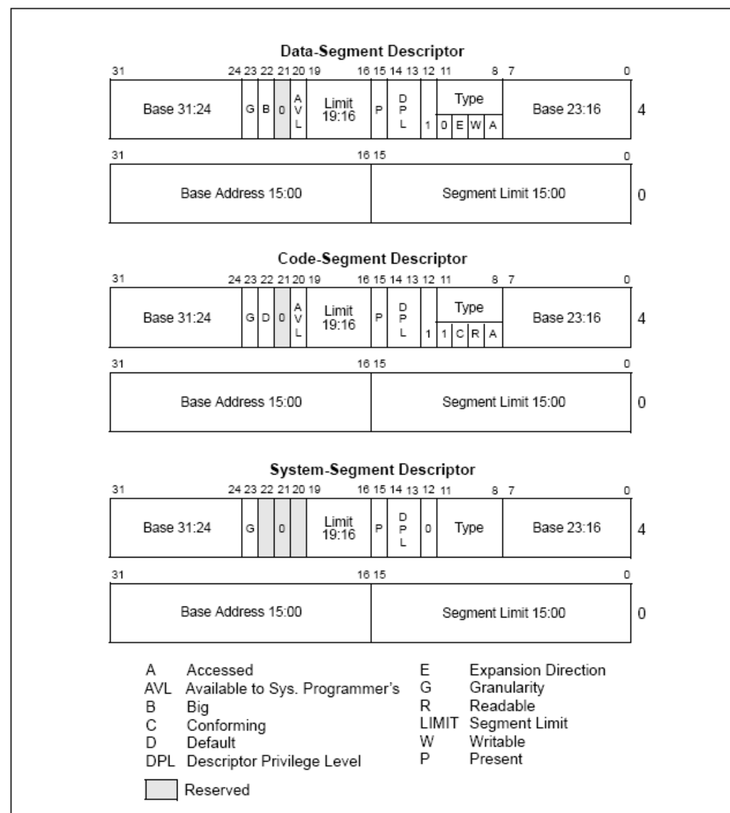


Slika 2.3-1 Tablice sustava

Slika 2.3-1 prikazuje kako se pomoću izbornika segmenta odabire pojedini segment u tablicama sustava (engl. *system tables*) GDT ili LDT trenutnog zadatka, pomoću polja TI. Procesor pomoću registara `GDTR` i `LDTR` zna gdje se nalaze te tablice u memoriji. No, u nastavku ćemo objasniti kako izgledaju opisnici koji se koriste u tim tablicama i sam princip korištenja tablice.

Napomena: nulti opisnik (engl. *null descriptor*) GDT tablice je rezerviran za potrebe procesora i ne koristi se od strane programera.

2.3.1 Opća tablica opisnika - GDT

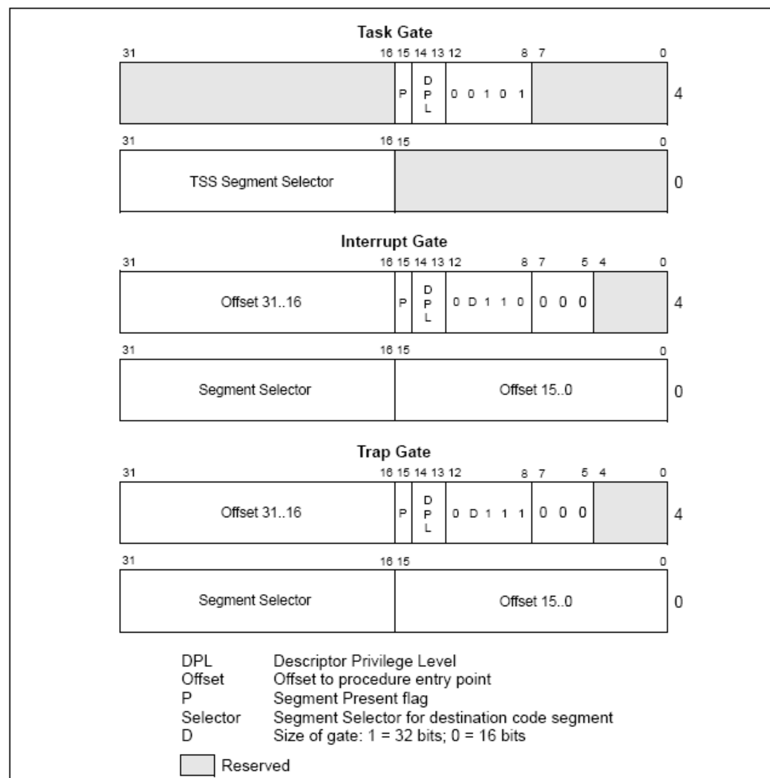


Slika 2.3-2 Tipovi opisnika GDT tablice

Prema slici 2.3-2 GDT ima 3 tipa opisnika (engl. *system types*), odnosno LDT tablica 2 tipa opisnika. To su opisnik segmenta za podatke, opisnik segmenta za instrukcije i opisnik segmenata sustava (engl. *system segment descriptor*).

Opisnik segmenata sustava je na primjer, TSS opisnik o kojem će biti više riječi u poglavlju 2.4. Neka polja zajednička za sve segmente su na primjer, zastavica zrnatosti - G (engl. *granularity*), 1kB ili 4kB. Polje E (engl. *expansion direction*) koje je za segment podataka i instrukcija smjer širenja prema rastućim adresama, a za segment stoga je prema padajućim adresama. Početna adresa (engl. *base address*) podrazumijeva početak segmenta u memoriji, opseg segmenta (engl. *segment limit*) označava veličinu tog prostora u memoriji počevši od početne adrese. P polje (engl. *present*) koristi se u mehanizmu straničenja kada određeni segment ne mora biti dostupan u fizičkoj memoriji nego se nalazi na disku. Razina privilegija opisnika (DPL) označava kojem zaštitnom prstenu pripada dotični segment, zapravo, tko ima pravo pristupiti tom segmentu. Polja W i R kad su postavljena označavaju da li je dozvoljeno pisanje po segmentu memorije ili čitanje. Te se vrijednosti tipično postavljaju ovisno o kojem se segmentu radi. Na primjer, ako je riječ o segmentu instrukcija, on je samo dostupan za čitanje.

2.3.2 Tablica opisnika prekida - IDT



Slika 2.3-3 Tipovi opisnika vrata

Odmak sadrži adresu prekidne procedure, DPL označava zaštitni prsten u kojem se koristi prekid. Izbornik segmenta navodi u kojem segmentu se smije koristiti navedeni prekid.

Slika 2.3-3 jasno prikazuje kako se opisnici vrata (engl. *gate descriptors*), odnosno opisnici IDT tablice dijele na tri tipa:

1. vrata zadataka (engl. *task gate*)
2. vrata prekida (engl. *interrupt gate*)
3. vrata zamke (engl. *trap gate*)

U ovome radu ćemo koristiti samo 2. i 3. tip opisnika. Naime, opisnik vrata zadataka podrazumijeva korištenje prekida kod raspoređivanja zadataka. U takvom načinu prebacivanja izvođenja zadataka, koristi se prekid kako bi se prebacilo izvođenje programskog toka na drugi zadatak. To samo po sebi podrazumijeva korištenje polja pokazivača prethodnog zadatka (engl. *previous task link)* TSS strukture prema slici 2.4-2 . Tada se pozivom oprekida prebacuje izvođenje na drugi zadatak, a pri povratku iz tog zadatka u polju pokazivača prethodnog zadatka je informacija o prethodnom zadatku koji se izvodio, te se izvođenje prebacuje natrag na taj zadatak.

U takvom načinu rada koristi se ugniježđeno raspoređivanje poslova (engl. *nested task scheduling)*, što u nekom ozbiljnijem načinu rada sa raspoređivanjem poslova ne daje prostora ostvarenju nekog drugačijeg algoritma. Pri takvom načinu rada novi zadatak se poziva CALL instrukcijom. Stoga u ovom radu i inače za izradu raspoređivača poslova, a uporabom sklopovskog prebacivanja zadataka (engl. *hardware task swithcing)*, tj. uporabom TSS strukture, koristi se raspoređivanje bez gniježđenja odnosno, to se postiže instrukcijom `JMP`, odnosno dugim skokom (engl. *far jump*, *long jump)*, gdje se ta instrukcija koristi u

kombinaciji sa TSS izbornikom segmenta koji je opisan pripadajućim opisnikom u GDT tablici.

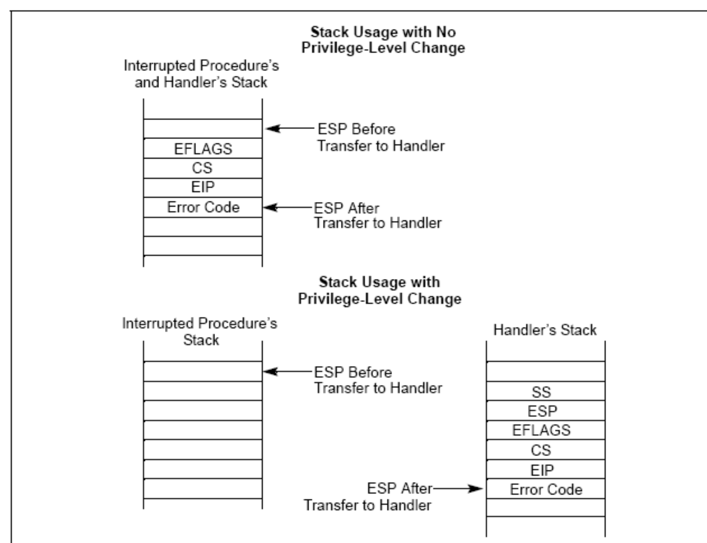
Za sada je dovoljno reći da u ovom radu koristimo isključivo vrata za prekide, te vrata za iznimke. Pod prekidima podrazumijevamo sklopovske prekide koje izazivaju razne izlazno-ulazne jedinice, a iznimke izaziva procesor uslijed nastalih grešaka u izvođenju. Prekidi se dijele na maskirajuće i nemaskirajuće, to jest MI (engl. *maskable interrupts*) i NMI (engl. *non maskable interrupts*). Maskirajući prekidi se mogu potisnuti (engl. *inhibited interrupts*) brisanjem IF zastavice u `EFLAGS` registru zastavica prema slici 2.2-2 , odnosno nemaskirajući prekidi se ne mogu potisnuti na taj način. Postavljenje zastavice IF izvršava se `STI` instrukcijom, dok se brisanje izvršava `CLI` instrukcijom. Ove dvije instrukcije se koriste u kombinaciji kad želimo osigurati neki kritični odsječak koda. Prvo se izvrši instrukcija `CLI`, zatim kritični dio, zatim `STI` čime se ponovno omogućuje prekide.

2.3.2.1 Podjela iznimaka arhitekture i386

Iznimke (engl. *exceptions*) se dijele na 3 vrste:

1. Greške (engl. *faults*) - greška je iznimka koja u općenitom slučaju dozvoljava normalan nastavak rada trenutnog toka izvođenja. Pri povratku iz rukovatelja iznimke procesor izvođenje vraća na instrukciju koja je izazvala grešku, a ne na iduću.
2. Zamke (engl. *traps*) - zamka se izvršava odmah nakon instrukcije koja ju je izazvala. Pri povratku iz rukovatelja iznimke izvođenje se nastavlja sa idućom instrukciju nakon one koja je izazvala iznimku.
3. Izbačaji (engl. *aborts*) - izbačaj je iznimka koja nužno ne prijavljuje točno mjesto gdje se u izvođenju dogodila iznimka. Izbačaj prijavljuje neke ozbiljne sklopovske pogreške ili neispravne vrijednosti u tablicama sustava ili registrima.

2.3.2.2 Izgled stoga i format koda pogreške pri pojavi iznimke

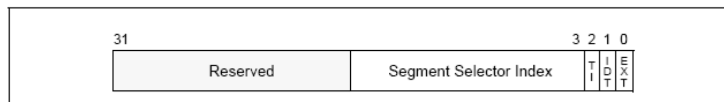


Slika 2.3-4 Okvir stoga kod pojave iznimke

Okvir stoga (engl. *stack frame*) nakon pojave iznimke izgleda kao što je to prikazano na slici 2.3-4 . Ovisno o tome da li se pri pojavi iznimke mijenja razina zaštitnog prstena postoje dva stoga, jedan za prsten 0, jedan za prsten 3. Na slici je vidljivo da se pohranjuje trenutni `SS`

(engl. *stack segment*), ESP (engl. *stack pointer*), EFLAGS(registar zastavica), CS, EIP (engl. *instruction pointer*) i kod pogreške (engl. *error code*), koji se stavlja na stog samo kod grešaka kao podskupine iznimaka.

Kod greške ima format kao što je to na slici 2.3-5 .



Slika 2.3-5 Format koda greške

EXT polje opisuje da li je uzročnik vanjski, na primjer sklopovski prekid. IDT polje govori da li je grešku izazvao prekid iz tablice IDT. Ako polje IDT nije postavljeno, tada ako je uzrok opisnik u GDT ili LDT tablici, TI polje je za LDT postavljeno, a GDT obrisano. Ovisno o uzročniku greške bilo to IDT prekidu ili GDT/LDT opisnicima u kodu greške se nalazi indeks opisnika u pripadnoj tablici koji je izazvao grešku. Na primjer kod 18_{16} (binarno 1100_{2}), označava da je uzročnik u trenutnoj LDT tablici, opisnik pod indeksom 1.

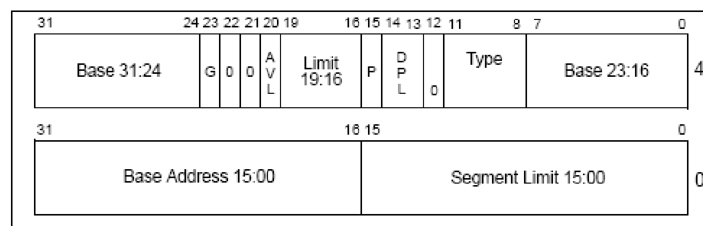
2.3.2.3 Upravljanje prekidima

Upravljanje prekidima (engl. *interrupt handling*) podrazumijeva da se kod pisanja procedura za prekid, ali isto tako i za iznimke, vodi računa o čuvanju konteksta odnosno stanju stoga. Svaka takva procedura mora završavati IRET instrukcijom koja označava povratak iz prekida.

2.3.2.4 Ostvarenje poziva jezgre pomoću prekida

Poziv jezgre ostvaruje se pomoću programskog prekida odnosno poziva se pomoću INT instrukcije i indeksa opisnika u IDT tablici. Ostvarenje procedure poziva jezgre je potpuno isto kao i procedure za sklopovski prekid.

2.3.3 Lokalna tablica opisnika - LDT

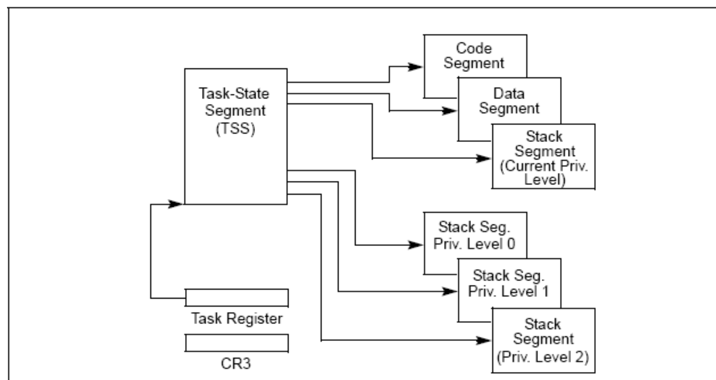


Slika 2.3-6 LDT opisnik

Opisnik LDT tablice je formata kako je prikazano na slici 2.3-6. Format opisnika je sličan opisniku GDT tablice.

Napomena: LDT tablica ne smije sadržavati TSS opisnike ili bilo koju drugu vrstu opisnika sustava.

2.4 Segment stanja zadatka - TSS



Slika 2.4-1 Shema TSS strukture

Prema slici 2.4-1 TSS struktura je upravo to, struktura koja sadrži adrese raznih segmenata, ovisno o tome da li se zadatak nalazi u zaštitnom prstenu 0 ili 3. Ako se pojavi prekid dok je određeni zadatak u prstenu 3, tada je potrebno koristiti dva stoga za jedan te isti zadatak. Jedan za prsten 0, drugi za prsten 3. Također potrebno je čuvati i kazalo stogova za prstene 0 i 3. TSS struktura nije ništa drugo nego način kako se sklopovski čuva kontekst određenog zadatka kako bi se kasnije mogao ponovo pokrenuti. Čuvaju se razni registri, ali kao što smo rekli ne koristimo gniježđenje zadataka, stoga nije potrebno polje za pokazivač na prethodni zadatak. Bitno je obratiti pažnju kako svaka TSS struktura, dakle svaki zadatak, ima pripadni LDT izbornik segmenta. To znači da preko njega određujemo gdje se u memoriji nalazi LDT tablica za svaki pojedini zadatak.

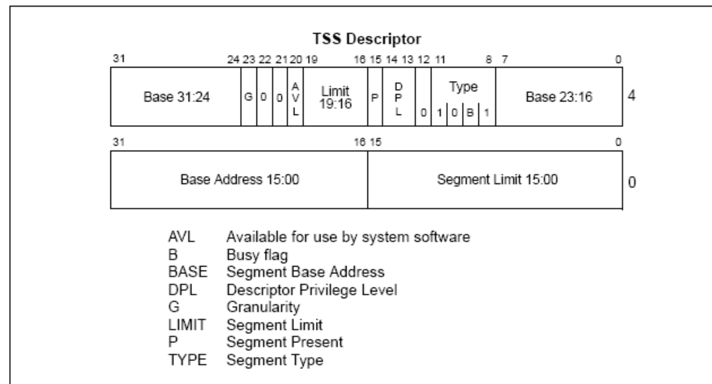
Slika 2.4-2 prikazuje sadržaj 104B TSS strukture za i386.

31	15	0	
I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

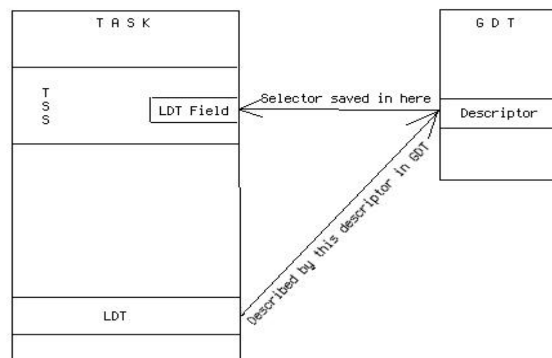
Reserved bits. Set to 0.

Slika 2.4-2 Izgled TSS strukture

TSS opisnik na slici 2.4-3 sadrži adresu TSS strukture i nalazi se zajedno sa pripadnim opisnikom LDT tablice zadatka u GDT tablici. Dakle, svaki zadatak ima svoj TSS opisnik, svoju LDT tablicu koja opisuje njegove segmente za podatke, instrukcije i stog, a ta dva opisnika se nalaze u GDT tablici.



Slika 2.4-3 TSS opisnik



Slika 2.4-4 Povezanost TSS strukture i LDT tablice zadatka

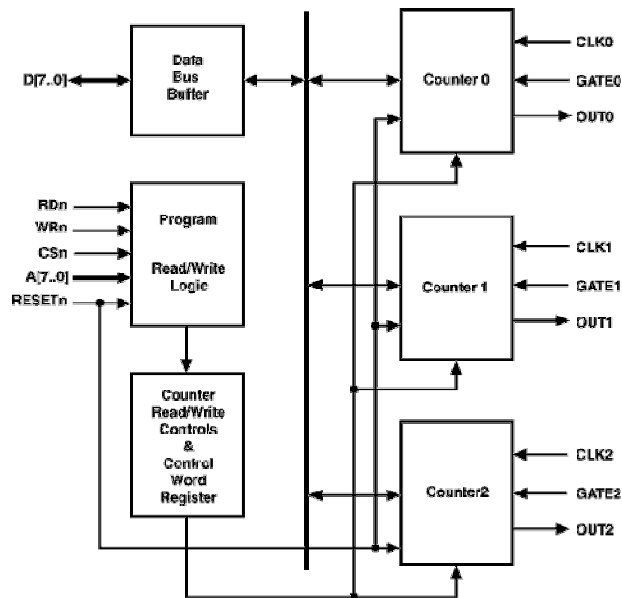
Slika 2.4-4 opisuje kako navedeni mehanizam funkcionira. Sa lijeve strane se nalaze TSS struktura i LDT tablica zadatka koji su smješteni negdje u memoriji, a sa desne strane se nalazi GDT tablica sa pripadnim opisnikom LDT tablice za dotični zadatak.

2.5 Brojilo sustava - PIT

Pogonitelj (engl. *driver*) tipkovnice je objašnjen u prethodnom seminarskom radu [2]. Ali ovdje će ukratko biti objašnjen princip rada PIT brojila (engl. *timera)* prema *OSdev* internet stranicama[6]. Naime, svaka arhitektura koja ima ikakve ozbiljne namjere na sebi pokretati operacijski sustav mora imati brojilo sustava. Razlog tome je što bez njega ne možemo koristiti bilo kakav raspoređivač poslova okidan prema vremenskim intervalima (engl. *time-slice).*

PIT (engl. *programmable interval timer*) je također poznat pod nazivom 8253 ili kao kasniji 8254 što je oznaka tog mikrokontrolera.

Sama frekvencija njegovog oscilatora je 1.193182 MHz, a razlog tome je što ta frekvencija proizlazi iz frekvencije korištene u elektroničkim krugovima televizijskih monitora u vrijeme kasnih 70-tih. Prikaz PIT mikrokontrolera je na slici 2.5-1 , a detaljan opis se može pronaći na već spomenutim stranicama.



Slika 2.5-1 Prikaz 8253/8254 brojila

2.6 Programski načini raspoređivanja

Većina modernih operacijskih sustava mora udovoljiti potrebama sve zahtjevnijih uređaja koji su spojeni na računalo kao što je to jedinica za prikaz na ekranu - GPU (engl. *graphics processing unit*). Takve jedinice sadrže vlastite registre koje je potrebno sačuvati u kontekstu trenutnog zadatka koji se izvodi.

Drugi problem je što TSS struktura sadrži mnogo podataka koje možda ne trebamo u kontekstu zadatka u našem operacijskom sustavu što generira mnogo nepotrebnog trošenja resursa sustava (engl. *overhead*).

Kao što ćemo vidjeti za jedan raspoređivač poslova nije dovoljna TSS struktura. Nju treba proširiti i napraviti strukturu zadatka, kojom raspolaže raspoređivač poslova. Na primjer u njoj se može nalaziti stanje zadatka, prioritet, signali itd, ovisno o dizajnu raspoređivača poslova i samog operacijskog sustava.

Umjesto cijele TSS strukture na slici 2.4-2 , dovoljno je sačuvati sadržaje registara `EBP, ESP, EIP, EFLAGS,` za jednostavnu uporabu prebacivanja zadataka, te ih napuniti ručno vrijednostima novog zadatka. Pri tom se ne koristi dugi skok, nego obična `JMP` instrukcija sa adresom novog zadatka.

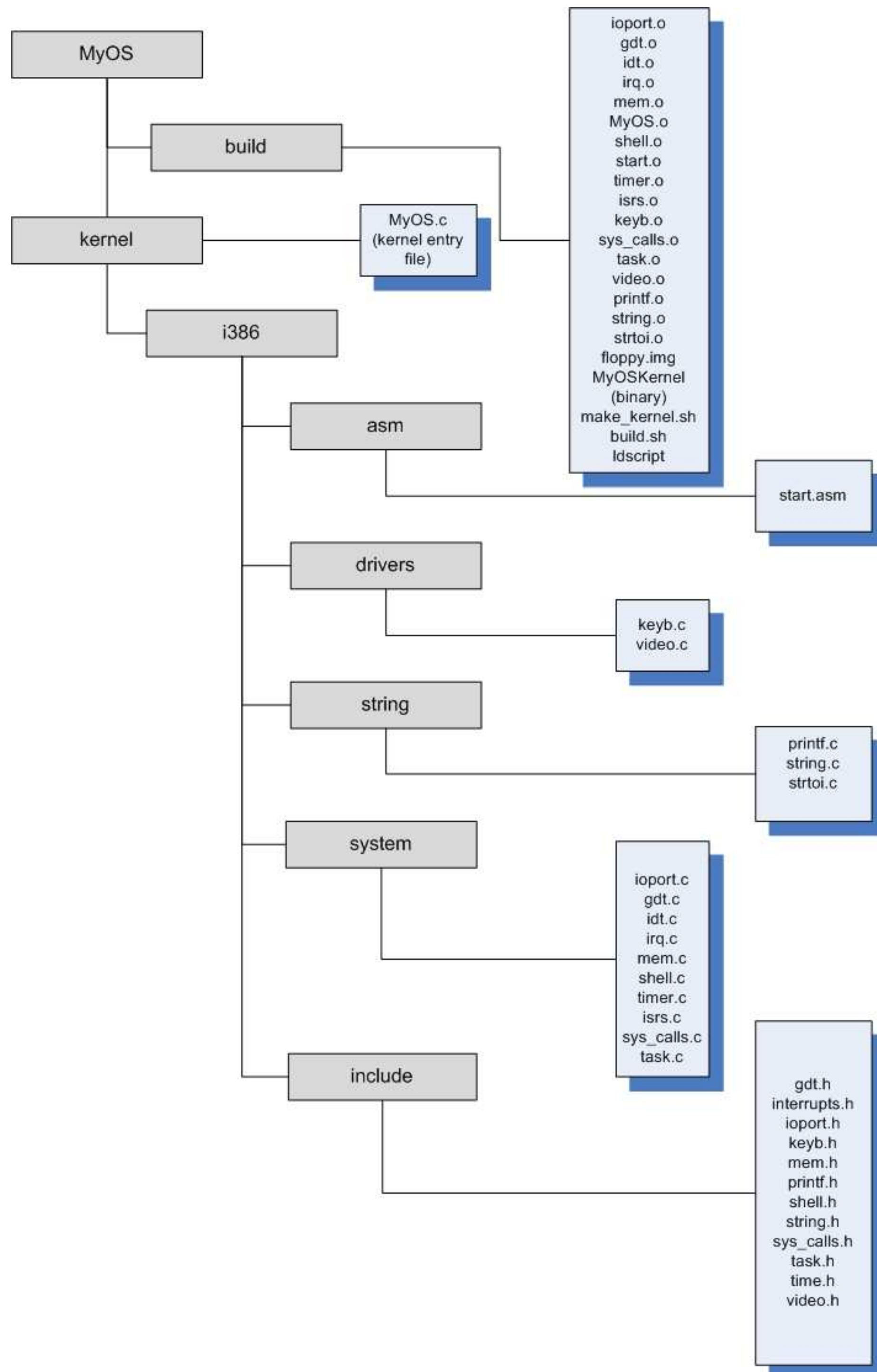
Programski način prebacivanja izvođenja danas je u najširoj upotrebi jer sklopovski način ne zadovoljava specifične potrebe konkretnog operacijskog sustava, bilo što se tiče performansi ili općenito kvantitete podataka, koji ne uključuje cijeli potreban kontekst.

3. MyOS: struktura izvornog koda

Ime operacijskog sustava nije posebno odabrano iz nekog razloga, nego jednostavno MyOS, zbog jednostavnosti. Možda je ime već korišteno za neki drugi operacijski sustav, ali budući da se ovdje radi tek o primjeru za akademske svrhe, to ne bi trebao biti problem.

U nastavku će detaljno biti objašnjen izvorni kod ovog operacijskog sustava.

3.1 Shema strukture direktorija i datoteka izvornog koda



Slika 3.1-1 Shema izvornog koda MyOS-a

Slika 3.1-1 prikazuje shemu svih direktorija i datoteka sa izvornim kodom MyOS-a. U daljnjem tekstu nećemo objašnjavati gdje se konkretno koja datoteka nalazi, već je to jasno vidljivo na ovoj slici.

3.2 Ulazna datoteka start.asm

U ovoj datoteci je početak koda jer je potrebno upravljati načinom kako se i gdje učitavaju pojedini dijelovi operacijskog sustava, a to je moguće napraviti preko asemblerskog koda i skripte povezivanja (engl. *linker script*), `ldscript` koja će kasnije biti objašnjena. Naime, kada gcc kompajler kompajlira može se dogoditi da na razna mjesta u memoriji stavi dijelove operacijskog sustava, a zbog toga što koristimo format izvršne datoteke jezgre (engl. *binary image)* za više tipova izvršnih datoteka operacijskog sustava(engl. *multiboot*) za Grub podizač jezgre (engl. *bootloader*), izvršna datoteka jezgre se mora učitati iznad 1MB memorije. Ostali detalji su objašnjeni u prijašnjem seminarskom radu[2].

U nastavku će nakon svakog dijela asemblerskog koda biti objašnjeno što predstavlja i kako se nadovezuje na kod koji je napisan na razini C jezika. Radi jednostavnosti, svi potprogrami i rukovatelji prekida, bit će smješteni u jednu datoteku. Dakako, kod se moglo podijeliti na više asemblerskih datoteka, ali MyOS ima samo jednu datoteku `start.asm`.

```
[BITS 32]
global start

start:
      mov   esp, sys_stack    ; This points the stack to our new stack area
      jmp   stublet
```

Ovdje se koristi globalna labela `start` kako bi bila vidljiva u C datatotekama ako se pokaže potreba za tim. `sys_stack` je labela koja se nalazi na kraju ove datoteke i označava prostor memorije zauzet za stog. Ovdje instrukcijom `MOV` punimo registar `ESP` upravo tim mjestom u memoriji kako bi naše kazalo stoga pokazivalo na tu mjesto.

```
; This part MUST be 4byte aligned, so we solve that issue using 'ALIGN 4'
ALIGN 4
mboot:
      ;Multiboot macros to make a few lines later more readable
      MULTIBOOT_PAGE_ALIGN    equ 1<<0
      MULTIBOOT_MEMORY_INFO   equ 1<<1
      MULTIBOOT_AOUT_KLUDGE   equ 1<<16
      MULTIBOOT_HEADER_MAGIC  equ 0x1BADB002
      MULTIBOOT_HEADER_FLAGS  equ MULTIBOOT_PAGE_ALIGN |
      MULTIBOOT_MEMORY_INFO | MULTIBOOT_AOUT_KLUDGE
      MULTIBOOT_CHECKSUM      equ -(MULTIBOOT_HEADER_MAGIC +
                              MULTIBOOT_HEADER_FLAGS)
      EXTERN code, bss, end
      ; This is the GRUB Multiboot header. A boot signature
      dd    MULTIBOOT_HEADER_MAGIC
      dd    MULTIBOOT_HEADER_FLAGS
      dd    MULTIBOOT_CHECKSUM
      ; be physical addresses. Make a note of these:
      ; The linker script fills in the data for these ones!
      dd    mboot
      dd    code
      dd    bss
      dd    end
      dd    start
```

Ovaj dio koda mora biti poravnat na granici od 4 bajta kako bi se zaglavlje za izvršnu datoteku jezgre moglo jasno učitati od strane Gruba. Ostalo su potrebna polja za takvo zaglavlje. Sva daljnja pojašnjenja se mogu naći u seminarskom radu[2].

Labele `code, bss, end` su vanjske jer su postavljene u `ldscriptu`. Tamo ćemo vidjeti kako se ostvaruje učitavanje izvršne datoteke iznad 1MB memorije.

```
; This is an endless loop here. Make a note of this: Later on, we
; will insert an 'extern _main', followed by 'call main', right
; before the 'jmp $'.
stublet:
      extern      cmain
      call  cmain
      jmp   $
```

Ovaj dio je zapravo pravi početak programskog koda naše jezgre. Ovdje se postavlja vanjska labela, odnosno memorijska lokacija `cmain()`, koja se nalazi u datoteci `MyOS.c`. Navedena C funkcija se poziva, ali se za svaki slučaj, ako u C datoteci ne postoji beskonačna petlja, ovdje procesor ipak zaustavi, prekine sa radom sa `jmp $`.

```
; This will set up our new segment registers. We need to do
; something special in order to set CS. We do what is called a
; far jump. A jump that includes a segment as well as an offset.
; This is declared in C as 'extern void gdt_flush();'
global gdt_flush
global gp
extern gp
gdt_flush:
      lgdt  [gp]
      mov   ax, 0x10
      mov   ds, ax
      mov   es, ax
      mov   fs, ax
      mov   gs, ax
      mov   ss, ax

      jmp   0x08:flush2
flush2:
      ret
```

`gdt_flush()` se koristi u `gdt.c` kako bi se napunile vrijednosti `GDTR` registra posebnim pokazivačem. Registri `DS, ES, FS, GS` koji su svi redom registri podataka, pune se vrijednošću `0x10` što označava izbornik segmenta našeg podatkovnog segmenta jezgre koji je postavljen u GDT tablici, a ona se nalazi u `gdt.c` datoteci.

Dugi skok osigurava da se prije izvođenja daljnjih instrukcija nalazimo u segmentu instrukcija operacijskog sustava. Zatim se izvođenje vraća u C kod u datoteci `gdt.c` instrukcijom `RET`.

```
; Loads the IDT defined in 'idtp' into the processor.
; This is declared in C as 'extern void idt_load();'
global idt_load
extern      idtp
idt_load:
      lidt  [idtp]
      ret
```

Ovaj dio se koristi u `idt.c` kako bi se napunio registar `IDTR` posebnim pokazivačem, a osigurava da procesor zna gdje se nalazi naša IDT tablica. Nakon toga se izvođenje vraća u C datoteku `idt.c` sa instrukcijom `RET`.

```
; Service Routines (ISRs) right here!
global      isr0
      .
      .
      .
global      isr31

; 0: Divide By Zero Exception
isr0:
      cli
      push  byte 0
      push  byte 0
      jmp   isr_common_stub
      .
      .
      .
; 31: Reserved
isr31:
      cli
      push  byte 0
      push  byte 31
      jmp   isr_common_stub
```

Ovdje su postavljeni rukovatelji iznimaka (engl. *interrupt service routines, isrs*). Postoje razne iznimke po svojim vrstama, ali ne vraćaju sve kod greške, nego samo greška kao podskupina iznimki. Za ove ostale stavljamo 0 bajt na stog kako bi osigurali da se pri povratku iz rukovatelja, vrati prvotno stanje stoga, te osigura istovjetna procedura za sve iznimke. U nastavku se izvođenje prebacuje na memorijsko mjesto `isr_common_stub`.

```
; We call a C function in here. We need to let the assembler know
; that 'fault_handler' exists in another file

extern fault_handler

; This is our common ISR stub. It saves the processor state, sets
; up for kernel mode segments, calls the C-level fault handler,
; and finally restores the stack frame.

isr_common_stub:
      pusha
      push  ds
      push  es
      push  fs
      push  gs
      mov   ax, 0x10
      mov   ds, ax
      mov   es, ax
      mov   fs, ax
      mov   gs, ax
      mov   eax, esp ; save our ESP
```

Ovdje se stavljaju na stog svi registri za podatkovni segment te kazalo stoga.

```
      push  eax
      mov   eax, fault_handler
      call  eax
      pop   eax
      pop   gs
      pop   fs
      pop   es
      pop   ds
```

```
        popa

        add   esp, 8      ;Cleans up the pushed error code and pushed ISR
                        ;number
        iret            ;pops 5 things at once: CS, EIP, EFLAGS, SS, and ESP!
```

U ovom dijelu pozivamo `fault_handler()` koji se zapravo nalazi u datoteci `isrs.c`. `fault_handler()` ne radi nešto posebno nego jednostavno ispiše broj i tip pogreške jednom na ekran. Kod naprednijih rukovatelja iznimki moguće je izmijeniti kod koji uzrokuje iznimku kako se ne bi javljala iznimka (engl. *debugging*). Ovdje je to učinjeno na način koji osigurava da kazalo stoga ostane sačuvano, jer smo ga sačuvali u `eax` registru, kojeg nakon toga isto stavljamo na stog te onda adresu `fault_handlera()` stavljamo u taj registar nakon čega ga pozivamo sa instrukcijom `CALL`. Prije instrukcije `IRET`, odnosno prije povratka iz procedure iznimke, uvećavamo kazalo stoga sustava, upravo zato jer smo kod ulaska u proceduru stavljali 2 bajta jedan za kod greške kod grešaka, a kod ostalih 0 bajt, te sam broj iznimke. Sada da bi se vratili na pravo mjesto na stogu, uvećavamo ga za 8.

```
global irq0
        .
        .
        .
global irq15

; 32: IRQ0
irq0:
        cli
        push  byte 0
        push  byte 32
        jmp   irq_common_stub
        .
        .
        .
; 47: IRQ15
irq15:
        cli
        push  byte 0
        push  byte 47
        jmp   irq_common_stub

extern      irq_handler
```

Procedure za vanjske prekide se nalaze ovdje tj. za MI prekide. Potrebno je naglasiti kako je potrebno napraviti ponovno postavljanje (engl. *remap*) PIC-ova, kako bi se prekidi nalazi redom od 0 do 15 na mjestima u IDT tablici od 32-47. PIC-evi su mikrokontroleri koji obavljaju posao dojave prekida ili iznimke procesoru. To je potrebno jer se na primjer kod pojave iznimke dijeljenja s nulom (engl. *division by zero*) i IRQ0 prekida, tj. prekida brojila, javlja greška, a ne prekid. `irq_remap()` i ostale funkcije za postavljanje prekida nalaze se u `irq.c` datoteci. `irq_handler()` se nalazi u datoteci `irq.c`.

```
irq_common_stub:
        pusha
        push  ds
        push  es
        push  fs
        push  gs
        ;ensure that we use system stack, that is system DATA SEGMENT
        mov   ax, 0x10
        mov   ds, ax
```

```
      mov   es, ax
      mov   fs, ax
      mov   gs, ax
      mov   eax, esp

      push eax
      mov   eax, irq_handler
      call  eax

      pop   eax
      pop   gs
      pop   fs
      pop   es
      pop   ds
      popa
      add   esp, 8 ; this is because here are the new CS and EIP of the ISR
                 ;routine
      iret
```

Ovaj dio koda je sličan kodu procedura za iznimke. Osiguravamo da kazalo stoga pokazuje na ispravno mjesto na stogu. Prije toga se poziva `irq_handler()`,glavni rukovatelj prekidima koji se nalazi u `irq.c` datoteci.

```
;tasks for demonstration
global      task1_run
global      task2_run
global      task3_run
global      task4_run
global      task5_run
global      task6_run
global      task7_run
global      task8_run
global      task9_run

extern      do_task1
extern      do_task2
extern      do_task3
extern      do_task4
extern      do_task5
extern      do_task6
extern      do_task7
extern      do_task8
extern      do_task9

task1_run:
      call  do_task1
      jmp   task_finished

task2_run:
      call  do_task2
      jmp   task_finished

task3_run:
      call  do_task3
      jmp   task_finished


task4_run:
      call  do_task4
      jmp   task_finished
```

```
task5_run:
      call  do_task5
      jmp   task_finished

task6_run:
      call  do_task6
      jmp   task_finished

task7_run:
      call  do_task7
      jmp   task_finished

task8_run:
      call  do_task8
      jmp   task_finished

task9_run:
      call  do_task9
      jmp   task_finished

task_finished:
      jmp   task_finished

;end of tasks for demonstration
```

Ovaj dio se odnosi na zadatke kojima je pravi početak ovdje, a glavni posao se odvija u datoteci `task.c` gdje se nalaze zadaci koji se pozivaju `CALL` instrukcijom. Svaki zadatak koji je završio, ostaje u petlji `task_finished`. Zadaci su po brojevima od 1-9, a zadatak čekanja sustava `TASK0` postavljen je u `task.c`.

```
;system calls are implemented here

global sys_calls_entry
global schedule
extern schedule

;push context on stack, eip, cs, old_esp, old_ss is pushed
;automatically during call instruction
sys_calls_entry:

      push  ds
      push  es
      push  fs
      ;in ax is our parameter for system cal
      mov   edx,10h
      mov   ds,dx
      mov   es,dx
      mov   edx,0fh
      mov   fs,dx
      cmp   eax,0
      je    sys_finished
      cmp   eax,1
      je    sys_schedule
      jmp   restore_context

sys_finished:
      jmp   restore_context
```

```
sys_schedule:
      call  schedule
      jmp   restore_context

;restore context
restore_context:
      pop   fs
      pop   es
      pop   ds
      iret
```

Ovaj dio je prava procedura za pozive jezgre, ali samo za pokazne svrhe. Ostvarena su 2 poziva jezgre, jedan za postavljanje stanja `TS_FINISHED`, nad trenutnim zadatkom, a drugi za pozivanje raspoređivača poslova. Parametar poziva, određuju se prema registru `eax`, koji se koristi nakon što se instrukcijom `INT 0x64` negdje u našem kodu pozove poziv jezgre.

```
; Here is the definition of our BSS section. Right now, we'll use
; it just to store the stack. Remember that a stack actually grows
; downwards, so we declare the size of the data before declaring
; the identifier 'sys_stack'
SECTION .bss
      resb    512000         ; This reserves 500kBytes of memory here
sys_stack:
```

Ovaj dio je rezerviran je za stog sustava, 500kB memorije, a raste prema padajućim adresama. Na početku `start.asm` postavlja se kazalo stoga na ovo mjesto u memoriji.

3.3 Početak koda jezgre operacijskog sustava

Bit će objašnjen kod na razini C jezika, koji se poziva iz `start.asm`, gdje se poziva funkcija `cmain()`.

3.3.1 Pokretanje ljuske

Korisnik šalje zahtjeve preko sučelja prema jezgri operacijskog sustava kako bi se mogao promatrati učinak raspoređivača poslova na sve dijelove sustava, zato se u funkciji `cmain()` pokreće ljuska (engl. *shell*) sustava.

```
//kernel entry point cmain() with parameters which are passed from
//grub
int cmain (unsigned long magic, unsigned long addr)
{
      //install everything
      setup_system();
```

Ovdje se postavlja sve od prekida do tablica sustava.

```
      //moves to ring 3, that is user mode
      move_to_user_mode();
```

Prebacujemo se u prsten 3, korisnički način rada operacijskog sustava.

```
      //start the shell inside kernel
      cprintf("\nTo start shell press ESC, if you made quit from shell");
      shell_running = 1;
      shell_starter();
```

Pokrećemo ljusku, sa postavljanjem zastavice `shell_running` te pozivom funkcije `shell_starter()` koja je postavljena u `shell.c` datoteci.

```
        //halt the machine, that is display this wheel :) sort of system idle
        //task
        for (;;) wheel();
}
```

Nakon što se prebacimo u korisnički način rada, sve tablice, prekidi, te dosad izvedene instrukcije, postaju zadatak pod nazivom `TASK0` koji je definiran u `task.c` sa cijelom svojom pripadnom `TSS` i `TASK_STRUCT` strukturom koju koristi raspoređivač poslova. Taj zadatak se naziva i zadatak čekanja sustava (engl. *system idle task, wait task*), koji se izvodi u slučaju da nije pokrenut ni jedan drugi zadatak. Taj zadatak stoji u beskonačnoj petlji, a prikazuje „kotačić“ na ekranu.

3.3.2 Postavljanje operacijskog sustava

U `cmain()` funkciji pozivamo funkciju `setup_system()` koja postavlja sve uređaje, tablice itd. na operacijskom sustavu.

```
//setup system, that is system drivers, IRQs, GDT and keyboard, video
//drivers
void setup_system()
{
        /* Clear the screen. */
        clear_task_print_area();
        cls();
```

`clear_task_print_area()` briše dio ekrana rezerviran za provjeru stanja zadataka odnosno pokazuje u kojem su koraku izvršavanja (engl. *task print area*), a to su prve dvije linije ekrana. `cls()` briše dio koji je rezerviran za ispis ljuske sustava, te isto tako zadataka ako koriste funkciju `cprintf()` postavljenu u `printf.c` datoteci.

```
        //install everything on a system
        gdt_install();
        idt_install();
```

Ovaj dio postavlja GDT i IDT tablice. `gdt_install()` se nalazi u `gdt.c`, a `idt_install()` se nalazi u `idt.c`. GDT tablica će biti objašnjena pomoću funkcije `gdt_install()` u poglavlju 3.3.3.

```
        isrs_install();
        irq_install();
        timer_install();
        keyboard_install();
        sys_calls_install();

        load_tasks();
}
```

Na ovom mjestu se postavljaju iznimke, prekidi, te se izvršava postavljanje prekida tipkovnice i brojila. Postavlja se prekid za pozive jezgre pod brojem `0x64`, a `sys_calls_install()` se nalazi u datoteci `sys_calls.c`. `irq_install()` se nalazi u `irq.c`, `isrs_install()` u `isrs.c`, `timer_install()` u `timer.c`, `keyboard_install()` u `keyb.c`, te `load_tasks()` u task.c. `load_tasks()` puni stogove i `TASK_STRUCT` strukture zadataka od 1- 9.

3.3.3 GDT tablica

Ovdje se nalazi funkcija `gdt_install()` koja postavlja poseban pokazivač, postavlja pojedine opisnike sa `gdt_set_gate()`, te na kraju poziva `gdt_flush()`iz `start.asm`, koja učitava poseban pokazivač u registar sustava `GDTR`,. U `gdt.c` korištene su već gotove funkcije iz kratkog uvoda u razvoj operacijskog sustava *Bran's kernel development tutorial [4].*

```
//* Should be called by main. This will setup the special GDT
//*  pointer, set up the first 3 entries in our GDT, and then
//*  finally call gdt_flush() in our assembler file in order
//*  to tell the processor where the new GDT is and update the
//*  new segment registers */
void gdt_install()
{
   /* Setup the GDT pointer and limit */
   gp.limit = (5*sizeof (struct gdt_entry)) - 1 ;
   gp.base = &gdt;
```

Postavljamo veličinu tablice i početnu adresu, a ona je postavljena adresom `gdt` polja struktura gdje svaka struktura u polju predstavlja jedan opisnik GDT tablice. `gp` je poseban pokazivač koji se koristi u `gdt_flush()`.

```
   /* Our NULL descriptor */
   gdt_set_gate(0, 0, 0, 0, 0);

   /* The second entry is our Code Segment. The base address
   *  is 0, the limit is 4GBytes, it uses 4KByte granularity,
   *  uses 32-bit opcodes, and is a Code Segment descriptor.
   *  Please check the table above in the tutorial in order
   *  to see exactly what each value means */
   gdt_set_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF);

   /* The third entry is our Data Segment. It's EXACTLY the
   *  same as our code segment, but the descriptor type in
   *  this entry's access byte says it's a Data Segment */
   gdt_set_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF);

   gdt_set_gate(3, 0, 0, 0, 0);
   gdt_set_gate(4, 0, 0, 0, 0);

   /* Flush out the old GDT and install the new changes! */
   gdt_flush();
}
```

Komentari koda opisuju kako izgledaju naši opisnici i koja prava su postavljena za svaki pojedini segment memorije. MyOS koristi pet segmenata koji su opisani indeksima od 0-4, odnosno koristimo 5 segmenata. Opisnici 3 i 4 koriste se za sve zadatke. Nulti opisnik je rezerviran za potrebe procesora. U `gdt_install()` postavljamo ih na 0, budući da njihove vrijednosti postavljamo tek prije prebacivanja u korisnički način rada, prema poglavlju 3.3.4 . GDT tablica je prikazana u tabeli 3.3-1.

Tabela 3.3-1 MyOS GDT opisnici

0 – Nulti opisnik
1 – Opisnik jezgrinog segmenta instrukcija
2 – Opisnik jezgrinog segmenta podataka
3 – TSS opisnik za sve zadatke
4 – LDT opisnik za sve zadatke

3.3.4 Prebacivanje u korisnički način rada

Arhitektura i386 ne posjeduje neku posebnu instrukciju za prebacivanje u korisnički način rada, već se za to služimo jednim trikom koji uključuje instrukciju `IRET`, dakle kao kod povratka iz prekida ili iznimke. `IRET` funkcija uzima sa stoga `EIP`, `SS`, `EFLAGS` i ostale registre prema slici 2.3-4 . `move_to_user_mode()` se poziva iz `cmain()`.

```
//moves to user mode by setting the proper entries in GDT -> TSS and
//LDT descriptor
void move_to_user_mode()
{
   //use template TASK0 structure to load TSS, LDT
   TASK0.tss_entry = set_tss((unsigned long long)
                        ((unsigned int)&TASK0.tss));
```

Opisnik pod indeksom 3 u GDT tablici zauzet je za TSS opisnik, a koriste ga svi zadaci. Kod prebacivanja izvođenja na drugi zadatak, raspoređivač poslova jednostavno mijenja dio gdje se u TSS strukturi nalazi `EIP` zadatka i postavlja novi pomoću `set_tss()` koji se nalazi u `task.c`. Postavlja se na trenutni zadatak, na `TASK0` koji predstavlja zadatak jezgre.

```
   TASK0.ldt_entry = set_ldt((unsigned long long)
                        ((unsigned int)&TASK0.ldt));
```

Opisnik pod indeksom 4 je zauzet za LDT opisnik, a tablicu koju on opisuje koriste svi zadaci, dakle ne postoji memorijska zaštita, svi zadaci djele isti memorijski prostor, što uključuje samu jezgru i njen zadatak `TASK0`. Kao i za TSS, LDT se postavlja tako da se upiše novi LDT opisnik za zadatak na koji se prebacuje izvođenje uporabom funkcije `set_ldt()` iz `task.c`.

```
   __asm__ ("ltrw %%ax\n\t"::"a"(TSS_SEL));
```

Rekli smo da instrukcija `LTR` postavlja `TR` registar, odnosno puni ga vrijednošću TSS izbornik segmenta (`0x18` prema `task.h`), to radimo samo jednom, jer MyOS koristi samo jedan TSS opisnik u GDT tablici za sve zadatke. Razlog tome je što izbornici segmenta imaju ograničenje na broj indeksa koji mogu upisati (8192 zapisa), što podrazumijeva ograničen broj zadataka, ali budući da ovdje koristimo, ranije opisanu metodu, mjenjamo samo `EIP` pojedinog zadatka i pokrećemo ga, što omogućava neograničen broj zadataka, naravno, ne doslovno jer to ovisi o veličini memorije.

```
   __asm__ ("lldt %%ax\n\t"::"a"(LDT_SEL));
```

Ovdje postavljamo `LDTR` registar instrukcijom `LLDT` i izbornikom segmenta LDT opisnika (`0x20` prema `task.h`) koji koristimo za sve zadatke.

```
      __asm__ __volatile__ ("cli");
```

Kada ulazimo u kritičan odsječak koda, onemogućujemo prekide.

```
      //this moves to user mode.
      __asm__ ("movl %%esp,%%eax\n\t"
            "pushl $0xf\n\t"
            "pushl %%eax\n\t"
            "movl  $0x3202,%%eax\n\t" //enables interrupts
                                       //after switch to user mode
            "pushl %%eax\n\t"
            "pushl $0x07\n\t"
            "pushl $1f\n\t"
            "iret\n"
            "1:\tmovl $0xf,%%eax\n\t"
            "movw %%ax,%%ds\n\t"
```

```
            "movw %%ax,%%es\n\t"
            "movw %%ax,%%fs\n\t"
            "movw %%ax,%%gs"
            :::"ax");
      //***end of move to user mode
}
```

Izvodimo prebacivanje u korisnički način rada. Zapazimo da ovdje postavljamo vrijednosti pojedinih registara, na način da ih stavimo na stog, a instrukcija `IRET` ih tada makne sa stoga i napuni registre vrijednostima. `0xf` označava segment podataka i segment stoga zadatka, a `0x07` označava segment instrukcija zadatka. Dio koda „`pushl $1f\n\t`" označava da se na stog stavlja memorijska lokacija iduće labele „`1:`", a ona se nalazi odmah iza `IRET` instrukcije. Dakle nakon što se izvrši `IRET`, izvođenje se prebacuje na „`1:`" te punimo registre segmenta podataka kako bi osiguralo da sadrže ispravne vrijednosti inače procesor dojavljuje grešku povrede memorijskog prostora - GPF (engl. *general protection fault*).

3.3.5 Zadatak čekanja sustava

Nakon prebacivanja u korisnički način rada, `TASK0` je pokrenut te zapravo ostaje u spomenutoj beskonačnoj petlji koja ispisuje „kotačić" na ekran u prostoru ekrana za zadatke, na 0,0 lokaciji ekrana. `wheel()` funkcija se nalazi u `timer.c`.

```
//halt the machine, that is display this wheel :) sort of system idle
//task
for (;;) wheel();
```

3.4 Princip rada ljuske operacijskog sustava

U ovom poglavlju objasnit ćemo kako je ostvarena ljuska sustava MyOS-a. Na ostalim poznatijim operacijskim sustavima ljuska je ostvarena kao jedan zaseban zadatak kojem operacijski sustav šalje signale i prekide sa tipkovnice. Budući da je ovo jednostavan sustav, ostvarenje je takvo da na prekid sa tipkovnice se pokreće ljuska sustava koja u skladu sa odgovarajućom tipkom prevodi naredbe i izvršava ih. Sve funkcije koje se koriste u ljuski pozivaju se izravno iz memorijskog prostora jezgre, jer MyOS nema odvojen memorijski prostor za jezgru i zadatke. Zato nema potrebe za ostvarenje poziva jezgre, jer svi zadaci mogu pristupiti jezgrinom memorijskom prostoru. Ne postoje ni knjižnice koda koje bi služile kao omotač (engl. `wrapper`) poziva jezgre kako se to uobičajeno radi.

U nastavku slijedi objašnjenje rukovatelja prekida tipkovnice.

3.4.1 Datoteka keyb.c

U ovoj datoteci nalazi se rukovatelj prekida tipkovnice.

```
//keyboard handler routine, also "trigger" of kernel shell
void keyboard_handler(struct regs *r)
{
   __asm__("cli");
```

Onemogućujemo prekide, jer ulazimo u obradu prekida, tj. u kritični odsječak.

```
   //key has been pressed
   char a;
   outb_p(0x21,inb_p(0x21)&0xfd);
   a=inb_p(0x61);
   outb_p(0x61,a|0x80);
   outb(a,0x61);
```

Kada je tipka pritisnuta, oglasi se PC speaker.

```
key_captured = 1;
key_scancode = kbd_data_read();
```

`key_captured` je zastavica koja označava da je tipka pritisnuta, a vidljiva je svim dijelovima koda koji uključe `keyb.h` datoteku. `kbd_data_read()` učitava kod tipke (engl. *scancode*), a varijabla `key_scancode` sadrži kod tipke, a vidljiva je svim datotekama koje uključe navedenu datoteku zaglavlja (engl. `header file`).

```
if (shell_running)
{
   shell(key_scancode);
}
```

Ako je pokrenuta ljuska pristupa se proslijeđivanju koda tipke funkciji `shell()` koji se nalazi u `shell.c`.

```
else if (key_scancode == KBD_CODE_ESC)
{
   shell_running = 1;
   shell_starter();
}
```

Ako nije pokrenuta ljuska, pritiskom tipke ESC pokrećemo ljusku, postavljanjem zastavice `shell_running` i izvodi se `shell_starter()`. `shell_starter()` je potrebna kako bi se svaki puta ispravno ispisala poruka da je pokrenuta ljuska, o čemu će biti više riječi kada će se govoriti o datoteci `shell.c`.

```
/* key stroke has been handled */
int com;
outb(0x61, (com=inb(0x61))|0x80 );
outb(0x61, com&0x7f);
```

Tipka je otpuštena i to dojavljujemo na PC speaker.

```
   __asm__("sti");
}
```

Napuštamo kritični odsječak i omogućujemo prekide.

3.4.2 Naredbe i parsiranje naredbi

U ovom poglavlju objasnit ćemo kako funkcije u `shell.c` zajedno čine ljusku sustava, kako se naredbe prevode i parsiraju, te kako se izvršavaju nakon parsiranja. Za početak objasnit ćemo `shell()` funkciju koja se poziva iz `keyb.c`.

```
void shell(char scancode)
{
   char keyboard_command;
```

Ovu varijablu koristimo kako bi postavili naredbu koju ćemo slati tipkovnici za postavljanje LED dioda, koje se pale na tipke 1,2,3 ili su sve upaljene za neku drugu tipku.

```
   //when key released, on MSb of the scancode, is zero.
   if (!(scancode & 0x80))
   {
      //OxED playing with LEDs, nothing really useful :D
      kbd_data_write(KBD_CCMD_SETLEDS);
```

```
        switch(scancode_to_ascii[scancode])
        {
              case '1':keyboard_command=0x02;break;
              case '2':keyboard_command=0x04;break;
              case '3':keyboard_command=0x01;break;
              default:keyboard_command=0x07;
        }
        kbd_data_write(keyboard_command);
```

Provjeravamo da li je tipka otpuštena, tj. najviši bit koda tipke je nula. Prvo se provjeri koja je tipka pritisnuta kako bi se namjestile diode na već opisani način.

```
    //manage scancodes through this function
    manage_chars(scancode);
    }
}
```

Zatim se kod tipke proslijeđuje funkciji `manage_chars()`koja provjerava kod tipke i grupira skupine znakova u naredbe ljuske, kako će biti opisano u nastavku.

Za grupiranje znakova u naredbe koristi se sljedeća funkcija:

```
//this function manages scancodes passed from shell() function
void manage_chars(char scancode)
{
      //well here we override the switch below and work directly with
      //scancodes, not ascii's
      if (scancode == KBD_CODE_BACKSPACE && backspace_enabled)
      {
            //we can only use backspace if we are writing a command after
            //prompt, and nothing else, so we use backspace_enabled to
            //check can we erase chars?
            if(strcmp("",COMMAND_BUFFER) && xpos > strlen(PROMPT))
            {
                  //rearrange command_buffer to erase one character from
                  //buffer
                  *command_buffer--;
                  *command_buffer == '\0';
                  xpos--;
                  //erase character before
                  cprintf("%c",' ');
                  //return to previous position, because cprintf increases
                  //xpos
                  xpos--;
                  }
      }
```

Ukoliko se radi o tipki za brisanje (engl. *backspace*) te je omogućeno korištenje tipke preko zastavice `backspace_enabled`, uspoređuje se da li je u tom trenutku na naredbenom upitu (engl. *prompt*) prazan niz znakova, da li je trenutno mjesto na ekranu desno od mjesta našeg naredbenog upita odnosno niza znakova za naredbeni redak. Ako su zadovoljeni ovi uvjeti, pokazivač `command_buffer` umanjuje se za jedan, što znači izbacujemo trenutnu tipku i znak iz grupe znakova naredbe, postavljamo na to mjesto znak za kraj niza znakova (engl. *null character*), umanjujemo trenutno mjesto ispisa na ekran `xpos`, ispisujemo prazan znak sa `cprintf()`, te opet umanjujemo trenutno mjesto na ekranu kako bi se vratili na ispravno mjesto na ekranu jer `cprintf()` također uvećava mjesto na ekranu za jedno mjesto.

```
      //else just check chars
      else switch(scancode_to_ascii[scancode])
      {
```

Ovaj dio koda se pokreće tek nakon što se izravnom metodom uspoređivanja sa kodom tipke ne utvrdi koja je tipka, te se tada gleda u našem popisu tipki (engl. *keymap*) o kojoj je tipki riječ, a popis se nalazi u `keyb.c` u polju `scancode_to_ascii[]`.

```
            //after Enter key we interpret command
            case '\n':
            case '\r':
                  *command_buffer='\0'; //end of string -> null character
                  backspace_enabled = 0;
                  command_interpret(COMMAND_BUFFER);
```

Pritiskom tipke za idući red stavljamo na mjesto na koje pokazuje pokazivač `command_buffer` znak za kraj niza znakova, onemogućujemo tipku za brisanje brisanjem zastavice `backspace_enabled`, te pozivamo funkciju `command_interpret()` koja prevodi i izvršava naredbe koje smo poslali ljuski. `command_interpret()` je glavna funkcija ljuske.

```
                  if (!wait_for_sched_freq && !wait_for_task_number &&
                                          !wait_for_value && shell_running)
                  {
                        cprintf("\n"PROMPT);
                        backspace_enabled = 1;
                  }
```

Ako se ne čeka dodatni unos sa tipkovnice, odnosno ne čeka se unos frekvecije raspoređivača poslova, unos broja zadatka, vrijednost memorijske lokacije, a zastavica `shell_running` je postavljena, ispisujemo naredbeni upit i omogućujemo tipku za brisanje.

```
                  command_buffer=(char *)COMMAND_BUFFER;
                  break;
```

Postavljamo `command_buffer` na početak grupe znakova za naredbu tj. na mjesto u memoriji `COMMAND_BUFFER` koje je postavljeno u `shell.h`.

```
            case SPECIAL:
                  break;
```

U slučaju specijalnih znakova oni se jednostavno zanemaruju.

```
            //print the character on the screen so user knows what he's
            //typing
            default:
                  putchar(scancode_to_ascii[scancode]);
                  *(command_buffer++) = scancode_to_ascii[scancode];
                  backspace_enabled = 1;
                  break;
      }
}
```

Ako nije riječ o tipki za idući red, sa funkcijom `putchar()` ispisujemo znak na ekran, stavljamo znak u grupu znakova za naredbu pomoću pokazivača `command_buffer`, te omogućujemo korištenje tipke za brisanje pomoću zastavice `backspace_enabled`.

U nastavku je objašnjena funkcija `command_interpret()`, koja je zapravo prevoditelj naredbi i koja ih izvršava. Budući da je ovo jedan od najvažnijih dijelova u MyOS-u, bit će detaljnije objašnjen. Funkcija kao parametar uzima naredbu preuzetu od `manage_chars()` funkcije.

```
//command interpreter function
int command_interpret(char *command)
{
      static int reg_eax, reg_ebx, reg_ecx, reg_edx, reg_esp,reg_cs;
```

Ovdje se nalaze varijable koje koristimo u asemblerskom kodu unutar C koda kako bi dobili vrijednosti pojedinih registara.

```
if (!strcmp("help",command))
{
       help();
       return 1;
}
```

Ova funkcija pomoću funkcije `help()` ispisuje upute za korištenje ove ljuske.

```
if (!strcmp("cls",command))
{
       cls();
       return 1;
}
else if(!strcmp("registers",command))
{
       registers();
       return 1;
}
else if(!strcmp("taskdesc",command))
{
       taskdesc();
       return 1;
}
else if(!strcmp("date",command))
{
       date();
       return 1;
}
else if(!strcmp("printgdt",command))
{
       printgdt();
       return 1;
}
```

Naredba `cls` poziva `cls()` iz `video.c` za brisanje prostora za ljusku, `registers` ispisuje stanje registara, `taskdesc` ispisuje na ekran TSS i LDT opisnike GDT tablice, `date` ispisuje trenutno vrijeme i datum te `printgdt` ispisuje cijelu GDT tablicu.

```
else if(!strcmp("schedoff",command))
{
       sched_on_timer = 0;
       cprintf("\nScheduler type: %s Frequency: %d Sched_state: %s"
             " Simulation_state: %s", sched_types[sched_type],
             sched_frequency, sched_on_timer ? "on" : "off",
             sim_on ? "on" : "off");
       return 1;
}
else if(!strcmp("schedon",command))
{
       sched_on_timer = 1;
       cprintf("\nScheduler type: %s Frequency: %d Sched_state: %s"
                   "Simulation_state: %s", sched_types[sched_type],
                   sched_frequency, sched_on_timer ? "on" : "off",
                   sim_on ? "on" : "off");
       return 1;
}
```

`schedon` i `schedoff`, uključuju ili isključuju raspoređivač poslova koji se zatim izvodi u intervalu određenim sa `sched_frequency`.

```
else if(!wait_for_task_number && !wait_for_value &&
                    !wait_for_sched_freq && !strcmp("print "
                    "tasks",command)||!strcmp("pt",command))
{
      printtasks();
      return 1;
}
```

`pt` naredba ili `print tasks` ispisuje zadatke pomoću `printtasks()`, iz `task.c`.

```
else if(!strcmp("trace on",command)||!strcmp("ton",command))
{
      trace_on = 1;
      return 1;
}
else if(!strcmp("clts",command))
{
      clear_task_print_area();
      return 1;
}
else if(!strcmp("trace off",command)||!strcmp("toff",command))
{
      trace_on = 0;
      return 1;
}
```

`traceon` i `traceoff`, uključuje odnosno isključuje praćenje raspoređivanja zadataka, a `clts` briše prve dvije linije ekrana koje su predviđene za poseban ispis zadataka.

```
else if(!strcmp("schedfreq set",command) || !strcmp("schfs",command))
{
      wait_for_sched_freq = 1;
      cprintf("\nScheduler type: %s Frequency: %d Sched_state: %s"
                  "Simulation_state: %s", sched_types[sched_type],
                  sched_frequency, sched_on_timer ? "on" : "off",
                  sim_on ? "on" : "off");
      cprintf("\nScheduler frequency:");
}
```

Postavljanje frekvencije raspoređivača poslova.

```
else if(!strcmp("schedtype fifo",command))
{
      sched_type = SCHED_FIFO;
      cprintf("\nScheduler type: %s Frequency: %d Sched_state: %s"
                  "Simulation_state: %s", sched_types[sched_type],
                  sched_frequency, sched_on_timer ? "on" : "off",
                  sim_on ? "on" : "off");
      return 1;
}
```

Postavi tip raspoređivača poslova zadataka na raspoređivač poslova po redu prispijeća, tj. zadaci se raspoređuju po redu prispijeća, odnosno to je FCFS (engl. *first come first served)* raspoređivanje poslova.

```
else if(!strcmp("schedtype rr",command))
{
      sched_type = SCHED_ROUND_ROBIN ;
      cprintf("\nScheduler type: %s Frequency: %d Sched_state: %s"
                  " Simulation_state: %s", sched_types[sched_type],
                  sched_frequency, sched_on_timer ? "on" : "off",
                  sim_on ? "on" : "off");
```

```
        return 1;
    }
```

Raspoređivanje poslova prema jednakim intervalima izvođenja se postavlja sa `schedtype rr`.

```
    else if(!strcmp("sched",command) || !strcmp("scheduler",command))
    {
        cprintf("\nScheduler type: %s Frequency: %d Sched_state: %s"
                " Simulation_state: %s", sched_types[sched_type],
                sched_frequency, sched_on_timer ? "on" : "off",
                sim_on ? "on" : "off");
    }
```

Trenutne postavke raspoređivača poslova se ispisuju sa `sched` naredbom.

```
    else if(!strcmp("halt",command))
    {
        shell_running = 0;
        cprintf("\nShell stopped!!!");
        cprintf("\nHalt invoked by user!!!");
        halt();
    }
```

`halt` zaustavlja sve zadatke i zaustavlja procesor preko `halt()` funkcije, koja se također nalazi u `shell.c`.

```
    else if(!strcmp("reboot",command))
    {
        shell_running = 0;
        cprintf("\nShell stopped!!!");
        reboot();
    }
```

`reboot` ponovno pokreće računalo.

```
    else if(!strcmp("quit",command) || !strcmp("q",command) ||
                                         !strcmp("exit",command))
    {
        shell_running = 0;
        cprintf("\nShell stopped!!!");
        cprintf("\nRun it again with ESC key!!!");
    }
```

`quit`, `q` i `exit` zaustavljaju ljusku, te se ona može ponovno pokrenuti ESC tipkom.

```
    else if((!strcmp("show task switch",command) ||
                                !strcmp("sts",command)))
    {
        show_task_switch = 1;
        cprintf("\nShowing task switching");
    }
    else if((!strcmp("hide task switch",command) ||
                                !strcmp("hts",command)))
    {
        show_task_switch = 0;
        cprintf("\nHiding task switching");
    }
```

Prikazivanje prebacivanja izvođenja zadataka raspoređivača poslova sa `sts`, `show task switch` naredbama ili ne zaustavljamo prikaz prebacivanja sa `hts`, `hide task switch`.

```
    else if((!strcmp("simon",command)))
    {
        sim_on = 1;
```

```
        cprintf("\nScheduler type: %s Frequency: %d Sched_state: %s"
                    "Simulation_state: %s", sched_types[sched_type],
                    sched_frequency, sched_on_timer ? "on" : "off",
                    sim_on ? "on" : "off");
        cprintf("\nSimulation on(not running tasks)");
}
else if(!strcmp("simoff",command))
{
        sim_on = 0;
        cprintf("\nScheduler type: %s Frequency: %d Sched_state: %s"
                    "Simulation_state: %s", sched_types[sched_type],
                    sched_frequency, sched_on_timer ? "on" : "off",
                    sim_on ? "on" : "off");
        cprintf("\nSimulation off(running tasks and switching)");
}
```

`simon` naredba pokreće simulaciju izvođenja pomoću trenutnog reda izvođenja, ne pokreće zapravo zadatke, a to je početna postavka sustava, dok sa `simoff`, `simulation off`, raspoređivač poslova stvarno izvodi zadatke.

```
//if command starts with one of these
else if(!strncmp("start task",command,strlen("start task")) ||
                              !strncmp("st",command,strlen("st")))
{
        char *params;
        if(!strncmp("st",command,strlen("st")))
              params = command + strlen("st");
        else
              params = command + strlen("start task");

        parse_parameters(params);
        int i = 0;
        for (;i<20 && strcmp(parameters[i],"THE END");i++)
              new_task(parameters[i]);
        return 1;
}
else if(!strncmp("run task",command,strlen("run task")) ||
                        !strncmp("rt",command,strlen("rt")))
{
        char *params;
        if(!strncmp("rt",command,strlen("rt")))
              params = command + strlen("rt");
        else
              params = command + strlen("run task");

        parse_parameters(params);
        int i = 0;
        for (;i<20 && strcmp(parameters[i],"THE END");i++)
              run_task(parameters[i]);
        return 1;
}
else if(!strncmp("finish task",command,strlen("finish task")) ||
                              !strncmp("ft",command,strlen("ft")))
{
        char *params;
        if(!strncmp("ft",command,strlen("ft")))
              params = command + strlen("ft");
        else
              params = command + strlen("finish task");

        parse_parameters(params);
```

```
        int i = 0;
        for (;i<20 && strcmp(parameters[i],"THE END");i++)
              finish_task(parameters[i]);
        return 1;
  }
```

`start task, run task, finish task` ili njihove kraće inačice `st, rt, ft` su naredbe za raspoređivanje zadataka. `st` omogućuje stavljanje zadatka u red za izvođenje pomoću `new_task()` funkcije. `rt` prisilno pokreće zadani zadatak, bez obzira koji zadatak se trenutno izvodi, pomoću `run_task()`. `ft` postavlja stanje zadanog zadatka pomoću finish_task() na `TS_FINISHED`. `new_task()`, `run_task()` te `finish_task()` nalaze se u `task.c` datoteci. Parametri ovih naredbi se unose u istom naredbenom retku, a kasnije se dalje parsiraju pomoću `parse_parameters()` funkcije koja je također u `shell.c`.

```
  //empty commands just ignore or if not empty
  //and starts with blank is unknown command
  else if(command[1]==' ')
  {
        int empty=1;
        for (;*command!='\0';command++)
        {
              if (*command != ' ')
                    empty = 0 ;
        }
        if(!empty)
              cprintf("\nUnknown command: \"%s\"",command);
  }
  else if (*command=='r' && !wait_for_value)
  {
        command++;
        //after r follows one letter a,b,c,d, after which write to
        //registers
        switch(*command)
        {
              case 'a':   reg_eax=strtoi(++command,16);
                          __asm__ __volatile__("mov %0, %%eax\n\t": :"r"
                          (reg_eax): "%eax", "%ebx", "%ecx",  "%edx");
                    break;
              case 'b':   reg_ebx=strtoi(++command,16);
                          __asm__ __volatile__("mov %0, %%ebx\n\t":
                          :"r"(reg_ebx): "%eax", "%ebx", "%ecx",
                          "%edx");
                    break;
              case 'c':   reg_ecx=strtoi(++command,16);
                          __asm__ __volatile__("mov %0, %%ecx\n\t": :"r"
                          (reg_ecx): "%eax", "%ebx", "%ecx",  "%edx");
                    break;
              case 'd':   reg_edx=strtoi(++command,16);
                          asm__ __volatile__("mov %0, %%edx\n\t": : "r"
                          (reg_edx): "%eax", "%ebx", "%ecx",  "%edx");
                    break;
              default:
                    cprintf("\nREGISTER->%c<- unavailable\n",*command);
        }
        //print registers
        registers();
  }
  else if (*command=='m'&& !wait_for_value)
  {
        command++;
```

```
		if (*command=='r')
		{
			requested_address=strtoi(++command,16);
			cprintf("\n%x: ", requested_address);
			char *mem_slider;
			for(mem_slider=requested_address; mem_slider
						< requested_address+8;mem_slider++)
			{
				cprintf("[%x] ",*mem_slider);
			}
			cprintf("\n");
		}
		else if (*command=='w')
		{
			requested_address=strtoi(++command,16);
			cprintf("\nSet new value:");
			wait_for_value=1;
		}
	}
	else if (wait_for_value)
	{
		wait_for_value=0;
		int *mem_slider;
		mem_slider=requested_address;
		*mem_slider=strtoi(command,16);
	}
	else if (wait_for_task_number)
	{
		wait_for_task_number = 0;
		if (!finish_task_flag && !run_task_flag)
		{
			new_task(command);
		}
		else if (finish_task_flag && !run_task_flag)
		{
			finish_task(command);
		}
		else if (run_task_flag)
		{
		run_task(command);
		}
		finish_task_flag = 0;
		run_task_flag = 0;
		printtasks();
	}
	else if(wait_for_sched_freq)
	{
		wait_for_sched_freq = 0;
		if (strtoi(command,10))
		{
			sched_frequency = strtoi(command,10);
		}
		//else if zero, I would rather it be 10 ;)
		else
		{
		sched_frequency = 10;
		}
		cprintf("\nScheduler type: %s Frequency: %d",
				sched_types[sched_type], sched_frequency);
	}
```

```
    //empty string
    else if (!strcmp("",command));
    else
    {
          cprintf("\nUnknown command: \"%s\"",command);
    }
    return 1;
}
```

Ovaj dio koda služi za unos frekvencije raspoređivača poslova, broja zadatka, nove vrijednosti memorijske lokacije. Ako su unešene prazne naredbe, naredbe koje počinju razmakom, takve se naredbe zanemaruju.

3.4.3 Povezanost ljuske sa raspoređivačem poslova

Raspoređivač poslova koji se nalazi u datoteci `task.c` sa glavnom funkcijom `schedule()` ima zastavice kojima pratimo i postavljamo ponašanje raspoređivača poslova i njegovih dijelova. U `shell.c` definirane su zastavice:

```
int show_task_switch = 0;
int sim_on = 1;
int trace_on;
int sched_on_timer = 1;
```

`show_task_switch` u `task.c` postavlja ispisivanje prebacivanja zadataka na primjer, kratka poruka „RR[1]“ označava da je upravo raspoređivač poslova prema jednakim intervalima izvođenja prebacio izvođenje na zadatak broj 1. MyOS ima statički definirane zadatke i memorijski prostor za njih, jer nije ostvareno upravljanje memorijom, što podrazumijeva gomilu za sustav i zauzimanje memorije (engl. *system heap*).

`sim_on` pokazuje da li je riječ o simulaciji, dakle da li stvarno prebacujemo izvođenje zadataka ili samo mjenjamo pojedine strukture zadataka unutar reda zadataka za izvođenje (engl. *run queue*). Kada je zastavica 0, ne radi se o simulaciji, nego o stvarnom izvođenju zadataka.

`trace_on` kada je postavljena, raspoređivač ispisuje popis zadataka u redu izvođenja, svaki puta kada se pozove `schedule()` funkcija u `task.c`, koja se poziva preko prekida brojila sustava.

`sched_on_timer` kada je postavljena označava da se poziva raspoređivač poslova pomoću `schedule()` funkcije na prekide brojila, inače ne poziva raspoređivač poslova.

U datoteci `task.c` postoje dodatne zastavice koje putem ljuske možemo namjestiti:

```
//scheduler type
short sched_type = SCHED_ROUND_ROBIN;

//scheduler frequency is this number * TIMER_FREQUENCY in time.h
int sched_frequency = 10;
```

Ovdje se postavlja tip raspoređivača koji se koristi, a tipovi su definirani u `task.h`. Moguće je namjestiti i frekvenciju raspoređivača koja je u početnom stanju jednaka `10`, a označava stvarni period poziva raspoređivača od `sched_frequency * TIMER_FREQUENCY` poziva u sekundi. `TIMER_FREQUENCY` definiran je u `time.h`, a označava frekvenciju pojavljivanja prekida brojila.

Postavljanjem ovih zastavica ostvaruje se osnovno upravljanje radom operacijskog sustava kada je upogonjen (engl. *online*).

3.5 Princip rada korištenih tipova raspoređivača poslova

Postoje razne metode, kako se mogu raspoređivati zadaci odnosno procesi. No, u ovom diplomskom radu se posebno razmatraju slučajevi raspoređivača jednakih vremenskih intervala izvođenja zadataka – RR (engl. *round robin*) te po redu dospijeća zadataka - FIFO (engl. *first in first out*), tj. točnije rečeno po redu dolaska zadataka - FCFS (engl. *first come first served*). Svi raspoređivači u ovom radu surađuju sa zadacima i jezgrom u raspoređivanju(engl. *cooperative tasks*). Zadaci imaju pravo odlučiti kada će predati izvođenje nekom drugom zadatku. Na koji je način to ostvareno više je objašnjeno u poglavlju gdje se objašnjava ostvarenje raspoređivača poslova u datoteci `task.c`.

3.5.1 Upravljanje po jednakim vremenskim intervalima izvođenja zadataka

Raspoređivač poslova po jednakim intervalima izvođenja dodjeljuje svakome zadatku određeni interval vremena u kojem se može izvoditi, te kasnije prebacuje izvođenje na drugi zadatak. Kada dođe do posljednjeg zadatka u redu za izvođenje pokreće opet prvi zadatak u redu za izvođenje.

3.5.2 Upravljanje po redu prispijeća

Raspoređivač po redu prispijeća(FIFO) pokreće prvi zadatak u redu za izvođenje, a ostale zadatke koji dolaze u red za izvođenje stavlja na kraj tog reda. Sve dok trenutni zadatak koji se izvodi ne prepusti izvođenje operacijskom sustavu, zadatak se izvodi bez obzira koliko procesorskog vremena se na to troši. Ovdje je bitno napomenuti da na taj način blokira druge zadatke da se izvedu, pogotovo na primjer, ako se radi o zadatku koji stoji u beskonačnoj petlji.

3.5.3 Ostali načini korištenjem ljuske

Već je ranije bilo opisano kako se pomoću naredbi `st, ft, rt` može upravljati izvođenjem zadataka. Ako upotrijebimo naredbu `schedoff` isključujemo raspoređivanje zadataka na prekid brojila. Tada možemo pomoću gore navedenih naredbi pokretati, zaustavljati, te stavljati zadatke u red za izvođenje zadatke prema vlastitom izboru.

3.6 Memorijska mapa MyOS-a – prostor za stog, instrukcije i podatke

Memorijska mapa MyOS-a će u ovom poglavlju biti objašnjena pomoću `ldscript` datoteke.

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

Definiramo format izvršne datoteke kod postupka povezivanja svih objektnih datoteka te tip arhitekture.

```
ENTRY(start)
phys = 0x00100000;
stack = 0x00200000;
```

Programski kod naše jezgre započinje na 1MB (`phys`), a prostor za stog na 2MB(`stack`). `ENTRY` je početak instrukcija za naš program.

```
SECTIONS
{
	.text phys : AT(phys)
	{
		code = .;
```

```
        *(.text)
        *(.rodata)
        . = ALIGN(4096);
    }
    .data : AT(phys + (data - code))
    {
        data = .;
        *(.data)
        . = ALIGN(4096);
    }
    .bss : AT(stack)
    {
         bss = .;
        *(.bss)
         = ALIGN(4096);
    }
    end = .;
}
```

U ostatku datoteke `ldscript` postavljaju se prostor za `.text` segment što u asembleru označava segment instrukcija, `.data` segment što u asembleru označava segment podataka, te `.bss` prostor za stog. Stogovi za zadatke su postavljeni su u `task.c` kao statička polja, pa se nalaze u podatkovnom segmentu memorije. Svi dijelovi su poravnati na 4kB(`ALIGN(4096)`), što je priprema za straničenje.

Budući da prema datoteci `start.asm` stog zauzima 500kB MyOS koristi 2,5MB memorijskog prostora. Memorijska mapa je prikazana u tabeli 3.6-1. Za izgled Grub memorijske mape može se pogledati u Grub dokumentaciju[8].

Tabela 3.6-1 MyOS memorijska mapa

0MB – Memorijski prostor za Grub podizač sustava
1MB – MyOS instrukcije, podaci, stogovi, instrukcije, podaci zadataka
2MB – Stog sustava, neincijalizirani podaci(500kB)

4. Raspoređivač poslova

U ovom poglavlju pobliže će biti objašnjen izgled `task.c` datoteke, memorijski prostor za zadatke, programski kod samih zadataka, tipovi raspoređivača poslova te njihovo ostvarenje, `TASK_STRUCT` struktura zadataka koju koristi raspoređivač poslova. Bit će kratko razmotrena poboljšanja navedenih algoritama jednakih intervala i po redu prispijeća.

4.1 Komunikacija među zadacima i jezgrom

4.1.1 Memorijski prostor zadataka

U `task.h` definirani su opisnici LDT tablice za segment instrukcija i segment podataka svakog zadatka. Ako pogledamo opisnike jezgre za segment instrukcija i segment podataka vidjet ćemo da se razlikuju jedino u zastavicama prava pristupa, ali u biti pokazuju na isti memorijski prostor. Dakle, zadaci i jezgra dijele isti memorijski prostor, zato zadaci mogu pozivati instrukcije korištene unutar jezgre, kao i unutar drugih zadataka. Odvajanje memorijskog prostora može se u MyOS-u ostvariti pomoću straničenja, ali to nije ostvareno. Za ostvarenje straničenja pogledati poglavlje 2.1.3.

Opisnik segmenta instrukcija jezgre je `0x00cf9a000000ffff`, a opisnik segmenta podataka jezgre je `0x00cf92000000ffff`.

```
//descriptors for LDT table
#define DEFAULT_LDT_CODE    0x00cffa000000ffffULL

#define DEFAULT_LDT_DATA    0x00cff2000000ffffULL
```

U ovom dijelu objasnit ćemo kako je zauzet prostor u memoriji za svaki zadatak, budući da nije ostvareno upravljanje memorijom.

```
//these are in start.asm, using as an entry point to avoid checking stack
//frames, that is stack state

extern void task1_run(void);
extern void task2_run(void);
extern void task3_run(void);
extern void task4_run(void);
extern void task5_run(void);
extern void task6_run(void);
extern void task7_run(void);
extern void task8_run(void);
extern void task9_run(void);
```

Ovdje svaki zadatak ima svoju pokretačku funkciju u `start.asm` koja je ovdje zbog toga da se izbjegne previše preuređivanja stoga kod pozivanja pojedinog zadatka, zato su ove funkcije ostvarene u asembleru, a ne u C kodu.

```
//this sort of memory allocation for tasks that is, all is static, not
//dinamically assigned
struct TASK_STRUCT task1;
struct TASK_STRUCT task2;
struct TASK_STRUCT task3;
struct TASK_STRUCT task4;
struct TASK_STRUCT task5;
struct TASK_STRUCT task6;
struct TASK_STRUCT task7;
struct TASK_STRUCT task8;
struct TASK_STRUCT task9;
```

Struktura TASK_STRUCT za svaki pojedini zadatak od njih 9 koliko ih je u memoriji, predstavlja način na koji jezgra prebacuje izvođenje svakog zadatka. Detaljnije će biti objašnjena u poglavlju 4.1.3.

```
//stack0 for tasks
static unsigned int task1_stack0[5*1024] = {0xf};
static unsigned int task2_stack0[5*1024] = {0xf};
static unsigned int task3_stack0[5*1024] = {0xf};
static unsigned int task4_stack0[5*1024] = {0xf};
static unsigned int task5_stack0[5*1024] = {0xf};
static unsigned int task6_stack0[5*1024] = {0xf};
static unsigned int task7_stack0[5*1024] = {0xf};
static unsigned int task8_stack0[5*1024] = {0xf};
static unsigned int task9_stack0[5*1024] = {0xf};


//stack3 for tasks
static unsigned int task1_stack3[5*1024] = {0xf};
static unsigned int task2_stack3[5*1024] = {0xf};
static unsigned int task3_stack3[5*1024] = {0xf};
static unsigned int task4_stack3[5*1024] = {0xf};
static unsigned int task5_stack3[5*1024] = {0xf};
static unsigned int task6_stack3[5*1024] = {0xf};
static unsigned int task7_stack3[5*1024] = {0xf};
static unsigned int task8_stack3[5*1024] = {0xf};
static unsigned int task9_stack3[5*1024] = {0xf};
```

Ovdje su postavljeni stogovi za prstene 0 i 3 TSS strukture prema slici 2.4-2 .

```
//this is used by new_task below

//eips, where to jump after task switch
unsigned int task_eips[] = { 0, task1_run, task2_run,
                  task3_run, task4_run,task5_run, task6_run, task7_run,
                  task8_run, task9_run, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

//addresses of task structs
unsigned int task_structs[] = { &TASK0, &task1, &task2, &task3, &task4,
                   &task5, &task6, &task7, &task8, &task9 };
```

task_eips[] koristi se u new_task() i drugim funkcijama za određivanje početne adrese programskog koda samih zadataka, a task_structs[] za adresu TASK_STRUCT strukture za svaki pojedini zadatak.

```
//load tasks listed above, name and stacks
void load_tasks()
{
      strcpy(task1.name,"1");
      strcpy(task2.name,"2");
      strcpy(task3.name,"3");
      strcpy(task4.name,"4");
      strcpy(task5.name,"5");
      strcpy(task6.name,"6");
      strcpy(task7.name,"7");
      strcpy(task8.name,"8");
      strcpy(task9.name,"9");

      //setting the values in task struct for task stacks
      //setting stack3 for tasks, starting from bottom
      task1.tss.esp = task1_stack3 + sizeof(task1_stack3);
      task2.tss.esp = task2_stack3 + sizeof(task2_stack3);
```

```
	task3.tss.esp = task3_stack3 + sizeof(task3_stack3);
	task4.tss.esp = task4_stack3 + sizeof(task4_stack3);
	task5.tss.esp = task5_stack3 + sizeof(task5_stack3);
	task6.tss.esp = task6_stack3 + sizeof(task6_stack3);
	task7.tss.esp = task7_stack3 + sizeof(task7_stack3);
	task8.tss.esp = task8_stack3 + sizeof(task8_stack3);
	task9.tss.esp = task9_stack3 + sizeof(task9_stack3);

	//setting stack0 for tasks, starting from bottom
	task1.tss.esp0 = task1_stack0 + sizeof(task1_stack0);
	task2.tss.esp0 = task2_stack0 + sizeof(task2_stack0);
	task3.tss.esp0 = task3_stack0 + sizeof(task3_stack0);
	task4.tss.esp0 = task4_stack0 + sizeof(task4_stack0);
	task5.tss.esp0 = task5_stack0 + sizeof(task5_stack0);
	task6.tss.esp0 = task6_stack0 + sizeof(task6_stack0);
	task7.tss.esp0 = task7_stack0 + sizeof(task7_stack0);
	task8.tss.esp0 = task8_stack0 + sizeof(task8_stack0);
	task9.tss.esp0 = task9_stack0 + sizeof(task9_stack0);
}
```

`load_tasks()` postavlja imena svih zadataka u `TASK_STRUCT` strukturi, budući da `new_task()`, `run_task()`, `finish_task()` i ostale funkcije kao parametar uzimaju upravo ime zadatka u obliku niza znakova. Postavlja kazalo stoga za prsten 0 u TSS strukturi na ranije postavljene stogove, dakle `task1.tss.esp0` pokazuje na kraj `task1_stack0` dijela memorije zauzete za stog, jer se stog širi prema padajućim adresama. Kada je zadatak u korisničkom načinu rada tada koristi stog za prsten 3 i zato postavljamo `task1.tss.esp` na kraj memorijskog prostora `task1_stack3`. Svi navedeni stogovi i strukture nalaze se u segmentu podataka jer su postavljeni prije pokretanja same izvršne datoteke. To je učinjeno statički jer nije ostvaren sustav za upravljanje memorijom.

4.1.2 Izgled samog programskog koda zadataka MyOS-a

U ovom poglavlju ćemo razmotriti programski kod samih zadataka koji su po brojevima od 1-9. Svi zadaci koriste funkciju `set_video_memory()` iz `video.c` koja upisuje znak na ekran na željenom x, y mjestu ekrana. Zadaci pomoću te funkcije upisuju brojeve od 1-9 kako bi imali dojam gdje se zadatak zaustavio u izvođenju. Prva linija je mjesto gdje zadaci ispisuju korake izvođenja, a druga linija sadrži indekse zadataka, kako bi mogli pratiti koji zadatak upravo piše u taj poseban prostor za ispis.

```
//actual task functions, they get called by taskX_run above
void do_task1(void)
{
	cprintf("\nTASK1.........STARTED..........");
	char c = '0';
	int i = 0;
	for (i=0;i<10;i++)
	{
		set_video_memory(1,0,c+i);
	}
	cprintf("\nTASK1.........FINISHED..........");
	finish_me();
}
```

Zadatak 1 je jednostavan, ispisuje brojeve od 1-9 na prvoj liniji, te završava pomoću `finish_me()` funkcije, a prije toga ispisuje poruku da je završio. Ovdje je važno zapaziti da se zadaci uklanjaju iz reda izvođenja tek kada su u stanju `TS_FINISHED`, tj. tek kada se idući

puta pokrene `schedule()` funkcija na prekid brojila. Na taj način zadatak završava u beskonačnoj petlji i troši procesorsko vrijeme do idućeg poziva raspoređivača poslova.

```
//very demonstrative task giving birt to all kind of tasks, and then
//finishing
void do_task2(void)
{
	cprintf("\nTASK2.........STARTED...........");
	char c = '0';
	char buffer[100];
	int i=0;
	for (i=0;i<10;i++)
	{
		set_video_memory(2,0,c+i);
		if (i != 2)
			new_task(itoa(buffer,10,i));
	}
	cprintf("\nTASK2.........FINISHED...........");
	finish_me();
}
```

Zadatak 2 je glavni zadatak koji je korišten u ovom radu kao pokazni zadatak. On koristi funkciju `itoa()`iz `printf.c`, koja pretvara cjelobrojne brojeve u niz znakova, zatim tako dobiven niz znakova koristi kao ime za `new_task()` funkciju koja stavlja u red izvođenja novi zadatak. Zadatak pokreće sve ostale zadatke osim sebe samog. Izgled koda drugih zadataka pokazat će nam kako se oni pokreću i koje funkcije koriste. `new_task()` funkcija je dakle istovjetna `fork()` funkciji u operacijskim sustavima sličnim Unixu, jer stvara nove zadatke. MyOS ne bilježi u `TASK_STRUCT` strukturi koji je zadatak roditelj (engl. *parent*), a koji je zadatak dijete (engl. *child*). U MyOS-u nije dozvoljeno više zadataka istog imena, npr. ne mogu postojati 2 ista zadatka „5“ u redu izvođenja.

```
//TASK0 not using any of the slowing down functions
void do_task3(void)
{
	cprintf("\nTASK3.........STARTED...........");
	char c = '0';
	int i=0;
	while(1)
	for (i=0;i<10;i++)
	{
		set_video_memory(3,0,c+i);
	}
	cprintf("\nTASK3.........FINISHED...........");
	finish_me();
}
```

Zadatak 3 u beskonačnoj petlji ispisuje na prvoj liniji brojeve od 1-9 na svom pripadnom mjestu na ekranu (3. stupac ekrana).

```
//this task uses pause(1)
void do_task4(void)
{
	cprintf("\nTASK4.........STARTED...........");
	char c = '0';
	cprintf("\nTASK4 giving birth to 5 :)");
	new_task("5");
	int i=0;

	while(1)
	for (i=0;i<10;i++)
```

```
        {
                set_video_memory(4,0,c+i);
                pause(1);
        }
        cprintf("\nTASK4........FINISHED..........");
        finish_me();
}
```

Zadatak 4 koristi funkciju `new_task()` i stvara novi zadatak 5. Nakon toga u beskonačnoj petlji ispisuje brojeve od 1-9, nakon ispisa broja koristi funkciju `pause()` iz `timer.c`, koja zapravo koristi sekunde kao parametar, što znači da zaustavlja izvođenje 1 sekundu nakon ispisa broja. Zadatak ne završava ako ga ne prekinemo prisilno preko ljuske, što znači da ne prepušta izvođenje jezgri operacijskog sustava.

```
//TASK5 and TASK6 are in the recursion when scheduling by FCFS algorythim
void do_task5(void)
{
        cprintf("\nTASK5........STARTED..........");
        char c = '0';
        int i=0;
        cprintf("\nTASK5 giving birth to 6 :)");
        new_task("6");
        for (i=0;i<10;i++)
        {
                set_video_memory(5,0,c+i);
        }
        cprintf("\nTASK5........FINISHED..........");
        finish_me();
}
```

Zadatak 5 stvara zadatak 6, ispiše brojeve od 1-9, te zatim završava. Kod korištenja raspoređivača po redu prispjeća zadataci 5 i 6 se međusobno stvaraju što dovodi do rekurzivnog pozivanja i stvaranja zadataka.

```
//this task uses also timer_wait(10), 0.1s to slow down counting
void do_task6(void)
{
        cprintf("\nTASK6........STARTED..........");
        char c = '0';
        int i=0;
        cprintf("\nTASK6 giving birth to 5 :)");
        new_task("5");
        for (i=0;i<10;i++)
        {
                set_video_memory(6,0,c+i);
                timer_wait(10);
        }
        cprintf("\nTASK6........FINISHED..........");
        finish_me();
}
```

Zadatak 6 stvara zadatak 5, ispiše brojeve od 1-9, zaustavlja se funkcijom `timer_wait()` koja kao parametar uzima broj prekida brojila u ovom slučaju to je `10 * TIMER_FREQUENCY` (100 milisekundi). Zadatak zatim završava.

```
void do_task7(void)
{
        cprintf("\nTASK7........STARTED..........");
        char c = '0';
        int i=0;
```

```
        for (i=0;i<10;i++)
        {
                set_video_memory(7,0,c+i);
        }
        cprintf("\nTASK7.........FINISHED...........");
        finish_me();
}
```

Zadatak 7 ispiše brojeve od 1-9 i zatim završava.

```
//TASK0 uses pause(4)
void do_task8(void)
{
        cprintf("\nTASK8.........STARTED...........");
        char c = '0';
        int i=0;
        while(1)
        for (i=0;i<10;i++)
        {
                set_video_memory(8,0,c+i);
                pause(4);
        }
        cprintf("\nTASK8.........FINISHED...........");
        finish_me();
}
```

Zadatak 8 ispisuje broje od 1-9 sa zaustavljanjem izvođenja od 4 sekunde nakon ispisa svakog broja(`pause(4)`). Zadatak ne završava, nego stoji u beskonačnoj petlji.

```
void do_task9(void)
{
        cprintf("\nTASK9.........STARTED...........");
        char c = '0';
        int i = 0;
        cprintf("\nTASK9 killing TASK8, ups, that is \"finishing\"");
        finish_task("8");
        for (i=0;i<10;i++)
        {
                set_video_memory(9,0,c+i);
        }
        cprintf("\nTASK9.........FINISHED...........");
        finish_me();
}
```

Zadatak 9 prvo prisilno završava zadatak 8 pomoću `finish_task()`, te zatim ispisuje brojeve od 1-9, te tada završava.

4.1.3 TASK_STRUCT struktura raspoređivača poslova i TASK0 zadatak

`TASK_STRUCT` struktura te također i `TSS_STRUCT` struktura definirane su u `task.h`.

```
//TSS struct for hardware task switching
struct TSS_STRUCT
{
        int     back_link;
        int     esp0, ss0;
        int     esp1, ss1;
        int     esp2, ss2;
        int     cr3;
        int     eip;
        int     eflags;
        int     eax, ecx, edx, ebx;
```

```
	int   esp, ebp;
	int   esi, edi;
	int   es, cs, ss, ds, fs, gs;
	int   ldt;
	int   trace_bitmap;
}__attribute__((packed));
```

`TSS_STRUCT` struktura ostvarena je C strukturom koja je istovjetna onoj na slici 2.4-2 preuzetoj iz Intelovog priručnika[1].

```
//task structure for task scheduler
struct TASK_STRUCT
{
	int   state;
	char  name[100];
	struct      TASK_STRUCT *next;
	struct      TASK_STRUCT *previous;
	struct      TSS_STRUCT tss;
	unsigned    long  long  tss_entry;
	unsigned    long  long  ldt[2];
	unsigned    long  long  ldt_entry;
}__attribute__((packed));
```

`TASK_STRUCT` struktura za svaki zadatak sadrži stanje zadatka, koja su objašnjena u 4.1.4. Sadrži ime zadatka, pokazivač na sljedeći zadatak u redu izvođenja, pokazivač na prethodni zadatak u redu izvođenja, `TSS_STRUCT` strukturu za sklopovsko prebacivanje izvođenja zadataka, TSS opisnik koji koristi za zapis u GDT tablicu, te LDT tablicu (`ldt[2]`, sadrži dva opisnika segment instrukcija, te jedan zajednički segment za stog i podatke).

`TASK0` je početni zadatak, te kasnije zadatak čekanja sustava, koji je korišten u `new_task()` kao uzorak (engl. *template*) za stvaranje novih zadataka. Zadatak se postavlja u `task.c`, a također i njegov stog za prsten 0. Za prsten 3 koristi stog sustava prema `ldscript` i `start.asm` datotekama.

```
//stack for kernel, zero-task
static unsigned long TASK0_STACK[8192] = {0xf}; //8kB for TASKO_STACK0

//zero-task to be run, that is kernel, and template for other tasks
struct TASK_STRUCT TASK0 =
{
	/* state */
	TS_RUNNING,
	/* name */
	"TASK0",
	/* next */
	0,
	/* previous */
	0,
	/* tss */
	{/* back_link */
		0,
		/* esp0                                    ss0 */
		(unsigned)&TASK0_STACK+sizeof TASK0_STACK, DATA_SEL,
		//Make esp0 points to the "bottom" of the stack, because stack
		//grows downside, DATA_SEL and CODE_SEL appears in next few
		lines
		//are defined at include/task.h, they are selectors of data and
		//code segments in GDT
		/* esp1 ss1 esp2 ss2 */
		0, 0, 0, 0,
```

```
        /* cr3 */
        0,
        /* eip eflags */
        0, 0x3202,
        /* eax ecx edx ebx */
        0, 0, 0, 0,
        /* esp ebp */
        0, 0,
        /* esi edi */
        0, 0,
        /* es           cs              ds */
        USER_DATA_SEL, USER_CODE_SEL, USER_DATA_SEL,
        /* ss           fs              gs */
        USER_DATA_SEL, USER_DATA_SEL, USER_DATA_SEL,
        //Talk about these latter        /* ldt */
        0x20,
        /* trace_bitmap */
        0x80000000
    },
    /* tss_entry */
    0,
    /* ldt[2] */
    {DEFAULT_LDT_CODE, DEFAULT_LDT_DATA},
    /* ldt_entry */
    0
};
```

Kao što vidimo TASK0 struktura nema postavljena sva polja budući da ovaj zadatak napuni svoja polja kod prebacivanja u korisnički način rada, kao što je već objašnjeno u 3.3.4. Kada u redu za izvođenje ne postoji ni jedan drugi zadatak TASK0 ispisuje „kotačić" na mjestu 0,0 na ekranu.

4.1.4 Stanja zadataka i način komunikacije sa jezgrom

Za pojašnjenje komunikacije između zadataka i jezgre, koristit ćemo programski kod zadatka 2.

```
//very demonstrative task giving birt to all kind of tasks, and then
//finishing
void do_task2(void)
{
    cprintf("\nTASK2........STARTED..........");
    char c = '0';
    char buffer[100];
    int i=0;
    for (i=0;i<10;i++)
    {
        set_video_memory(2,0,c+i);
        if (i != 2)
            new_task(itoa(buffer,10,i));
    }
    cprintf("\nTASK2........FINISHED..........");
    finish_me();
}
```

Kao što je pojašnjeno svi zadaci i jezgra dijele sve segmente za instrukcije i podatke, pa je stoga lako ostvariti komunikaciju kao što je objašnjeno u 4.1.1. Ako pojedini zadatak želi ispisati nešto na ekran može koristiti sve funkcije jezgre kao što je to npr. `cprintf()`. Kod stvaranja zadataka koristi se `new_task()`, kao što to čini i ovdje zadatak 2 za stvaranje svih

ostalih zadataka. Može prisilno završiti neki drugi zadatak pomoću `finish_task()` što je istovjetno signalu za završavanje zadatka (engl. *kill*) u sustavima sličnim Unixu. Zadatak 2 koristi `itoa()` za pretvorbu cjelobrojnog broja u niz znakova. Kada bi memorijski prostor jezgre i zadataka bio odvojen ne bi mogli koristiti funkcije jezgre izravno, nego preko poziva jezgre, koji moraju biti ostvareni da bi se moglo komunicirati sa jezgrom. Komunikaciju je moguće ostvariti također preko signala, koji se izvršavaju kada je pojedini zadatak u izvođenju. Više o ostvarenju signala može se pogledati u izvornom kodu Linux 0.01 prve verzije Linux jezgre[5]. U tabeli 4.1-1 nalazi se popis stanja zadataka koji su na raspolaganju raspoređivaču poslova operacijskog sustava, a postavljenja su u datoteci task.h. Značenje određenog stanja ovisi o pojedinom tipu raspoređivača poslova što će biti objašnjeno u poglavljima 4.2.2 i 4.2.3.

Tabela 4.1-1 Popis stanja zadataka

Naziv stanja	Indeks stanja
TS_RUNNING	0
TS_RUNNABLE	1
TS_STOPPED	2
TS_FINISHED	3

4.2 Ostvarenje algoritama u MyOS-u

4.2.1 Pomoćne funkcije jezgre operacijskog sustava korištene za raspoređivanje zadataka

U ovom poglavlju su objašnjene pomoćne funkcije koje se mogu kasnije iskoristiti za ostvarenje poziva jezgre, dakle mogu biti izravno korištene za pojedini poziv jezgre. Objašnjene su funkcije `new_task()`, `run_task()` i `finish_task()` iz `task.c` datoteke.

4.2.1.1 New_task() funkcija

`new_task()` funkcija stvara nove zadatke, a bit će detaljno objašnjena u nastavku.

```
//this adds task to run_queue
void new_task(char * task_name)
{
   __asm__("cli");
```

Ulazimo u kritični odsječak pa onemogućujemo prekide instrukcijom `CLI`.

```
   //check if already in run queue
   if (already_in_queue(task_name))
   {
      cprintf("\nnew_task: Task \"%s\" is already in run"
                          "queue!!!",task_name);
      __asm__("sti");
      return;
   }
```

Provjeravamo da li se zadatak istog imena nalazi u redu za izvođenje pomoću `already_in_queue()`, a ako se nalazi obavijesti korisnika i vrati se iz funkcije. Prije izlaska iz zadatka omogućavamo prekide(`STI`).

```
//address of task is derived from it's name that is given by task_name
//using the task_eips
unsigned int eip = task_eips[atoi(task_name)];

//EIP can't be 0, except for TASKO, but we can't add TASK0 to queue, can
//we !?! :D
if(!eip)
{
   cprintf("\nnew_task: Task \"%s\" not valid EIP",task_name);
   __asm__("sti");
   return;
}
```

Pomoću imena zadatka i polja `task_eips[]` pronalazimo pripadni EIP za zadatak. Provjeravamo da li je EIP zadatka 0, a ako je obaviještavamo korisnika da je EIP zadatka neispravan.

```
//set pointer to added task struct
struct TASK_STRUCT *temp_task_struct = task_structs[atoi(task_name)];

//this saves stack pointers set above in this task.c
unsigned int esp = temp_task_struct->tss.esp;
unsigned int esp0 = temp_task_struct->tss.esp0;
```

Postavi pokazivač `temp_task_struct` na `TASK_STRUCT` strukturu koju dobivamo preko polja `task_structs[]` pomoću imena zadatka. Kazala stogova uzimamo iz dobivene strukture i čuvamo ih u `esp` i `esp0` varijablama.

```
//copy default values from TASK0, after which we set specific values for
//every task
memcpy(temp_task_struct, &TASK0, sizeof(struct TASK_STRUCT));
```

Punimo strukturu novog zadatka poljima `TASK0` zadatka, koji nam služi kao uzorak.

```
//TASK0 is used as a template, we just change some fields that will be
//modified
strcpy(temp_task_struct->name, task_name);
temp_task_struct->tss.esp = esp;
temp_task_struct->tss.esp0 = esp0;
temp_task_struct->tss.eip = eip;
temp_task_struct->tss.eflags = 0x3202;
temp_task_struct->next = 0;
```

Postavljamo ime zadatka, njegovo kazalo stoga prstena 3 na varijablu `esp`, kazalo stoga prstena 0 na varijablu `esp0`, njegov EIP na varijablu `eip`, te zastavice zadatka na `0x3202`. Pokazivač na idući zadatak u redu izvođenja postavljamo na 0, što znači da nema zadatka nakon novog zadatka koji smo upravo dodali u red izvođenja.

```
//jump over the TASK0, it's just template, and start of tasks, jump to
//first temp_task_struct runned
struct TASK_STRUCT *tasks = &TASK0;
//while you don't get do the last in the queue
while(tasks->next)
{
   tasks = tasks->next;
}
```

Postavljamo pokazivač `tasks` na adresu `TASK0` zadatka, a pomoću petlje tražimo zadnji dodani zadatak u redu izvođenja. `new_task()` funkcija novi zadatak uvijek dodaje na kraj reda za izvođenje.

```
    tasks->next = temp_task_struct;
    temp_task_struct->previous = tasks;
    temp_task_struct->state = TS_RUNNABLE;
    __asm__("sti");
}
```

Prethodno zadnji zadatak u redu izvođenja, sada pokazuje na novi zadatak, a novi zadatak pokazuje na prethodno zadnji zadatak u izvođenju. Postavljamo stanje novog zadatka na `TS_RUNNABLE`. Omogućujemo prekide sa `STI` i time izlazimo iz `new_task()`.

4.2.1.2 Run_task() funkcija

Funkcija `run_task()` jedina izvršava prebacivanje na novi zadatak. To je nužno ako želimo jedinstveno raspoređivanje poslova preko ljuske i ostalih tipova raspoređivača poslova. U nastavku slijedi detaljnije objašnjenje `run_task()` funkcije.

```
//run task by given name
void run_task(char *task_name)
{
      __asm__("cli");
      //using out tasks pointer from task.h
      struct TASK_STRUCT *run_queue = &TASK0;
      remove_finished_tasks();
```

Onemogućujemo prekide(`CLI`) te postavljamo `run_queue` pokazivač na adresu `TASK0` zadatka. `run_task()` poziva `remove_finished_tasks()` kako bi se uklonili sve završene zadaci iz reda za izvođenje.

```
      int task_found = 0;
```

`task_found` je zastavica koja nam govori da li je raspoređivač poslova uspio pronaći zadatak koji želimo izvesti.

```
      //we just find task, not setting it to current is set by scheduler,
      //or by user, but later on in run_task()
      while(run_queue)
      {
            //well, we could say we are forcibly stopping currently running
            //task
            if (run_queue->state == TS_RUNNING &&
                   strcmp(run_queue->name,task_name))
            {
                  run_queue->state = TS_STOPPED;
            }
```

Petljom prolazimo kroz sve strukture zadataka koji se nalaze u redu izvođenja, pomičući se prema naprijed pomoću pokazivača `run_queue`. Ako je pronađen zadatak koji je u stanju `TS_RUNNING`, on se zaustavlja odnosno postavlja mu se stanje u `TS_STOPPED`.

```
            //task we want to run is already running
            else if (run_queue->state == TS_RUNNING &&
                         !strcmp(run_queue->name,task_name))
            {
                  //supress waring if we where idling, that is wheel is
                  //running, that is TASK0 :D
```

```
            if (run_queue != &TASK0)
                  cprintf("\nrun_task: Task \"%s\" already"
                                                "running",task_name);
            return;
      }
```

Ako je zadatak koji želimo pokrenuti već u izvođenju, obaviještavamo korisnika ljuske da je zadatak u izvođenju.

```
      //task is found and is not running
      else if (!strcmp(run_queue->name, task_name))
      {
            task_found = 1;
            break;
      }
      run_queue = run_queue->next;
}
```

Ako je zadatak u stanju `TS_RUNNING` pronađen, prekidamo petlju. Inače, pomičemo `run_queue` na idući zadatak u redu izvođenja.

```
//run task if it's found
if (task_found )
{
      current = run_queue;
      current->state = TS_RUNNING;

      current->tss_entry = set_tss((unsigned long long)
                                    ((unsigned int)&current->tss));
      current->ldt_entry = set_ldt((unsigned long long)
                                    ((unsigned int)&current->ldt));
      __asm__ __volatile__("ljmp    $" TSS_SEL_STR ",    $0\n\t");
}
```

Ako je zadatak u stanju `TS_RUNNING` pronađen, `current` pokazivač se postavlja na `run_queue` pokazivač, stanje trenutnog zadatka koji želimo pokrenuti se postavlja na `TS_RUNNING`, upisujemo TSS opisnik zadatka koji želimo izvesti pomoću `set_tss()` u GDT tablicu, te upisujemo LDT opisnik zadatka pomoću `set_ldt()` u GDT tablicu. Prebacujemo izvođenje na zadataka koji želimo izvesti dugim skokom, odnosno instrukcijom `LJMP` i izbornikom TSS segmenta.

```
      else if (!task_found)
      {
            cprintf("\nrun_task: Task \"%s\" not in run_queue",task_name);
      }
      __asm__("sti");
}
```

Ako zadatak koji želimo izvesti nije pronađen u redu izvođenja, obavijesti korisnika ljuske.

4.2.1.3 Finish_task() funkcija

Ova funkcija postavlja stanje zadanog zadatka preko njegovog imena na `TS_FINISHED`.

```
//just set of task state to TS_FINISHED
void finish_task(char *task_name)
{
      __asm__("cli");
      struct TASK_STRUCT *run_queue = (&TASK0)->next;
      int found_task = 0;
```

Koristimo pokazivač run_queue za pomicanje kroz red izvođenja, te found_task zastavicu postavljamo kada pronađemo zadani zadatak.

```
    while(run_queue)
    {
        if (!strcmp(run_queue->name,task_name))
        {
            found_task = 1;
            run_queue->state = TS_FINISHED;
            cprintf("\nfinish_task(\"%s\"): forcibly FINISHED");
            break;
        }
        run_queue = run_queue->next;
    }
```

Prolazimo kroz petlju sve dok ne pronađemo zadani zadatak, te mu zatim postavljamo na stanje TS_FINISHED. Tada obavještavamo korisnika ljuske da je zadatak prisilno završen, jer je završen bez suradnje dotičnog zadatka, što znači zadatak nije sam postavio svoje stanje na TS_FINISHED pomoću finish_me() funkcije.

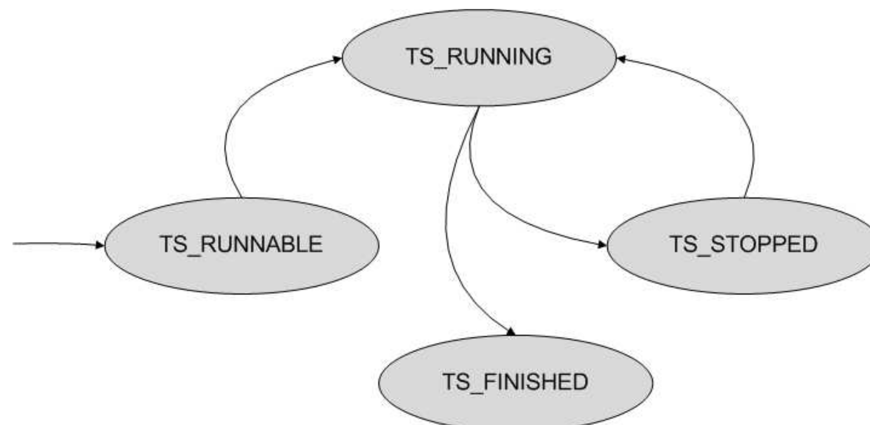
```
    //if we finished all tasks
    if (!(&TASK0)->next)
    {
        current = &TASK0;
        current->state == TS_RUNNING;
    }
```

Ako TASK0 ne pokazuje na nijedan idući zadatak u redu izvođenja postavljamo zadatak TASK0 kao trenutni, jer su svi ostali zadaci završili sa izvođenjem.

```
    if (!found_task)
        cprintf("\nfinish_task: Task \"%s\" not in run queue",
                                                task_name);
    __asm__("sti");
}
```

Ako nismo pronašli zadani zadatak obavještavamo korisnika ljuske da zadatak nije pronađen u redu za izvođenje. Omogućujemo prekide nakon izlaska iz kritičnog odsječka(STI).

4.2.2 Raspoređivač poslova prema jednakim intervalima izvođenja zadataka



Slika 4.2-1 Stanja zadataka za raspoređivač poslova prema jednakim intervalima izvođenja

Slika 4.2-1 prikazuje prijelaze između stanja zadatka kada se radi o ovom tipu raspoređivača poslova. Ako koristimo naredbu ljuske `finish task(ft)` tada iz bilo kojeg stanja prelazimo u stanje `TS_FINISHED` , a to vrijedi za bilo koji tip raspoređivača poslova. Ovdje će biti razmatran slučaj kod kojeg smatramo da ne postoji ljuska kojom korisnik može prisilno završiti izvođenje zadatka. U nastavku slijedi detaljno objašnjenje koda ovog raspoređivača poslova.

```
//ROUND ROBIN scheduler (time_sliced)
void rr_scheduler(void)
{
      struct TASK_STRUCT *run_queue = (&TASK0)->next;
      int current_found = 0;
      struct TASK_STRUCT *current_old;//setting to old current
      remove_finished_tasks();
```

Pokazivač `run_queue` koristimo za pomicanje po redu izvođenja. `current_found` je zastavica koja se postavlja kada se pronađe zadatak koji je u stanju `TS_RUNNING`. `current_old` pokazuje na stari zadatak koji je bio u izvođenju. Uklanjamo sve zadatke koji su završili sa `remove_finished_tasks()` funkcijom.

```
      //this is neccessary when run_task is called from shell "rt" command
      while(run_queue)
      {
            if (run_queue->state == TS_RUNNING)
            {
                  run_queue->state = TS_STOPPED;
                  current_old = run_queue;
                  if(!run_queue->next)
                  {
                        run_queue = &TASK0;
                  }
                  current_found = 1;
            }
```

Ako je pronađen zadatak koji je u izvođenju, kojem je stanje `TS_RUNNING`, zastavica `current_found` se postavlja. Ako je zadatka bio zadnji u redu izvođenja idući trenutni zadatak će biti `TASK0`. Stanje trenutnog zadatka se postavlja na `TS_STOPPED`.

```
            else if((current != run_queue) && current_found &&
                        ((run_queue->state == TS_STOPPED) ||
                        (run_queue->state == TS_RUNNABLE)))
            {
                  current = run_queue;
                  break;
            }
            run_queue = run_queue->next;
      }
```

Ako je pronađen novi zadatak za izvođenje postavlja se `current` pokazivač na njega, te se osigurava da nije isti kao prethodni trenutni zadatak koji je sada u izvođenju.

```
      //running the first task
      if (TASK0.next && !current_found)
      {
            current = TASK0.next;
      }
```

Ako nije pronađen trenutni zadatak i red izvođenja nije prazan, trenutni zadatak postaje prvi u redu izvođenja.

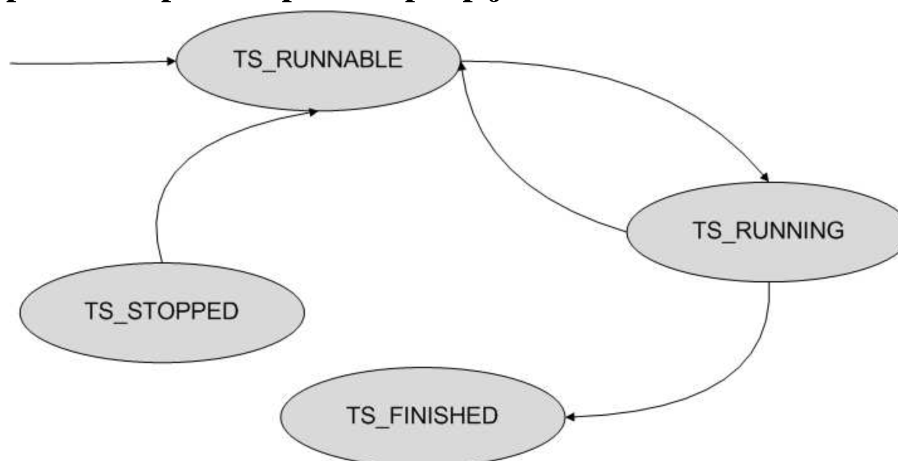
```
        else if(!current_found)
        {
                current = &TASK0;
        }
```

Ako ne postoji ni jedan drugi zadatak u redu izvođenja, TASK0 postaje trenutni zadatak.

```
        //if we are not just simulating, run the task!!!
        if (!sim_on)
        {
                //well this is from shell.c
                if (show_task_switch)
                        cprintf("(RR[%s])", current->name );
                run_task(current->name);
        }
}
```

Zastavice show_task_switch i sim_on mogu se postaviti od strane korisnika ljuske. show_task_switch omogućuje ispis prebacivanja izvođenja na novi trenutni zadatak. sim_on zastavica omogućava, ako je obrisana, stvarno prebacivanje izvođenja zadatka, a ako je uključena raspoređivač poslova samo simulira prebacivanje izvođenja, tj. ne poziva run_task() funkciju.

4.2.3 Raspoređivač poslova po redu prispijeća zadataka



Slika 4.2-2 Stanja zadataka za raspoređivač poslova po redu prispijeća zadataka

Slika 4.2-2 prikazuje stanja zadataka kod raspoređivača poslova po redu prispijeća zadataka. U nastavku će biti objašnjena funkcija fifo_scheduler().

```
//FIFO queued, that is FCFS scheduler (First Come First Served)
//(cooperative)
void fifo_scheduler()
{
        struct TASK_STRUCT *run_queue;
        int current_exists = 0;
        int found_new_current = 0;
```

Pokazivač run_queue nam služi za pomicanje po redu za izvođenje. current_exists se postavlja kad se pronađe zadatak u stanju TS_RUNNING. found_new_current se postavlja kada pronađemo zadatak koji je trenutno u stanju TS_RUNNABLE, a idući je u redu izvođenja.

```
        //this is if we switched from ROUND ROBIN and we set all TS_STOPPED
        //to TS_RUNNABLE, that is FIFO don't recognize TS_STOPPED
        run_queue = &TASK0;
        while(run_queue)
```

```
{
        if (run_queue->state == TS_STOPPED)
                run_queue->state = TS_RUNNABLE;
        run_queue = run_queue->next;
}
```

Budući da operacijski sustav raspolaže sa više tipova raspoređivača poslova, ne koriste svi tipovi sva stanja. Raspoređivač po redu prispijeća prvo postavlja stanja zadataka koji su u `TS_STOPPED` u stanje `TS_RUNNABLE`, zato da bi se mogao nastaviti rad raspoređivača, koji ne prepoznaje stanje `TS_STOPPED`. On za razliku od raspoređivača prema jednakim intervalima ne podržava prekidljive zadatke (engl. *preemptive tasks*).

```
//check if we forcibly removed current ?!?! by shell maybe
run_queue = &TASK0;
while(run_queue)
{
        if (!strcmp(run_queue->name,current->name))
        {
                current_exists = 1;
                break;
        }
        run_queue = run_queue->next;
}
if (!current_exists)
        current = &TASK0;
```

Provjeravamo da li je uklonjen zadatak koji je bio u izvođenju od strane korisnika ljuske. Ako je uklonjen iz reda izvođenja onda postavljamo trenutni zadatak za izvođenje na `TASK0`.

```
//run fifo scheduling only if current finished
if (current->state == TS_FINISHED && current != &TASK0)
{
        //searching next available task in queue starting from current
        run_queue = current;
        while(run_queue)
        {
                if (!found_new_current && current != run_queue &&
                                    run_queue->state == TS_RUNNABLE)
                {
                        current = run_queue;
                        found_new_current = 1;
                        break;
                }
                run_queue = run_queue->next;
        }
        remove_finished_tasks();
        //if we emptied the queue
        if(!TASK0.next)
        {
                current = &TASK0;
                current->state = TS_RUNNABLE;
        }
}
```

Raspoređivač poslova provjerava da li je trenutni zadatak prešao u stanje `TS_FINISHED`, te ako trenutni zadatak nije `TASK0`, petljom tražimo idući zadatak u redu za izvođenje koji je u stanju `TS_RUNNABLE`, kojega postavlja kao trenutni zadatak. Nakon toga, odstranjujemo sve završene zadatke, a tu je također prethodni zadatak koji se izvodio sa `remove_finished_tasks()`. Ako nakon toga ne postoji nijedan zadatak u redu za izvođenje

(!TASK0.next) tada postavljamo kao trenutni zadatak TASK0, te mu stanje postavljamo na TS_RUNNABLE.

```
	//this is if we are starting with FSFC and we need to stop TASK0
	else if (current == &TASK0)
	{
		//run next of TASK0
		if (current->next)
		{
			current->state = TS_STOPPED;
			current = current->next;
		}
	}
```

U slučaju da je trenutni zadatak TASK0 postavljen kao trenutni idući zadatak u redu izvođenja, provjeri da li je red izvođenja prazan (current->next). Ako nije prazan, postavi kao trenutni zadatak prvi zadatak u redu izvođenja i zaustavi zadatak TASK0.

```
	//if current is still running this is FIFO queue, who are we to stop
	//it ;)
	else if (current->state == TS_RUNNING)
	{
		//this is from shell.c
		if (show_task_switch)
			cprintf("FIFO(wait[%s])", current->name);
		return;
	}
```

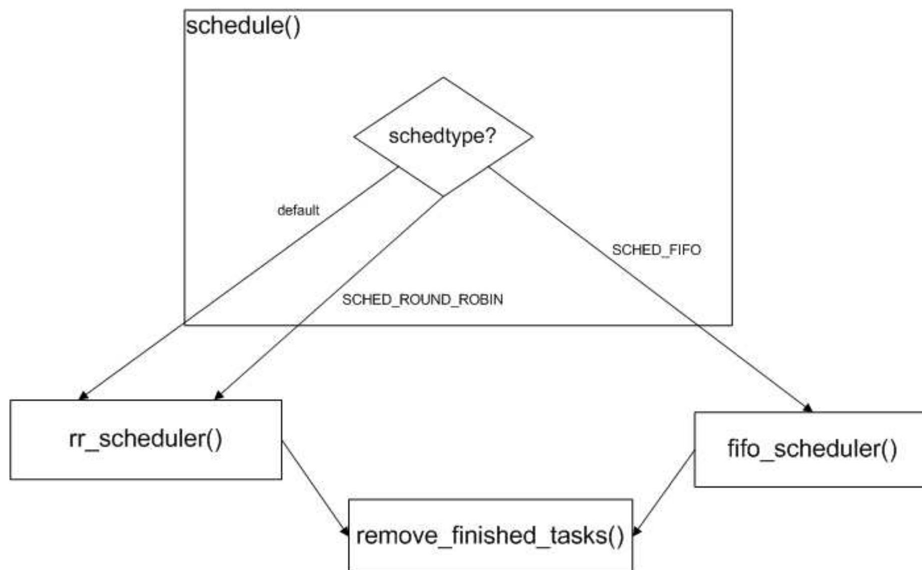
Ako unatoč svim provjerama trenutni zadatak je u stanju TS_RUNNING, raspoređivač poslova se vraća iz funkcije fifo_scheduler() i obavještava korisnika da čeka na zadatak, jer nema pravo prekinuti trenutni zadatak sve dok on ne pređe u stanje TS_FINISHED, prisilno preko ljuske ili kad se sam završi. Za to vrijeme ako je uključena zastavica show_task_switch ispisuje se poruka da raspoređivač čeka da trenutni zadatak završi.

```
	//if we are not just simulating, run the task!!!
	if (!sim_on)
	{
		//well this is from shell.c
		if (show_task_switch)
			cprintf("(FIFO[%s])", current->name );
		run_task(current->name);
	}
}
```

Kao u slučaju prethodnog tipa raspoređivača poslova, zastavice postavljaju što će se ispisivati na ekranu u prostoru za ljusku.

4.2.4 Mogućnost ostvarenja ostalih algoritama raspoređivača poslova u MyOS-u

Svi ostali algoritmi koje možemo ostvariti moraju se postaviti u datoteci task.h kao novi tipovi raspoređivača poslova, te pripadne funkcije koje pozivamo za raspoređivanje zadataka. Slika 4.2-3 prikazuje tok izvođenja nakon što se pozove schedule() funkcija task.c datoteke.



Slika 4.2-3 Schedule() funkcija raspoređivača poslova

Prikazano je kako funkcija na temelju `schedtype` varijable određuje koji je tip raspoređivača poslova u uporabi, a osnovni početni tip je postavljen na raspoređivač poslova jednakih intervala (`rr_scheduler()` funkcija) odnosno `schedtype` je u početku `SCHED_ROUND_ROBIN`. Drugi tip ovdje definiran je `SCHED_FIFO`, te ako je `schedtype` postavljen na ovu vrijednost, pokreće raspoređivač poslova po redu prispijeća zadataka. U nastavku slijedi komentar `schedule()` funkcije. Svaki od tipova raspoređivača poslova uklanja završene zadatke iz reda izvođenja pomoću `remove_finished_tasks()` funkcije.

```
//main schedule function
void schedule(void)
{
        __asm__("cli");
        //installs irq's again, because they misteriously get erased?!?!?
        reset_system();

        switch(sched_type)
        {
                case  SCHED_FIFO:
                        fifo_scheduler();
                      break;
                case  SCHED_ROUND_ROBIN:
                //default is round_robin
                default:
                        rr_scheduler();
        }
        __asm__("sti");
}
```

U ovom dijelu osiguravamo kritični odsječak sa `CLI` i `STI` instrukcijama. `reset_system()` postavlja samo ponovno rukovatelje prekida u IDT tablici, jer se u MyOS-u taj dio iz nekog razloga obriše kod prebacivanja izvođenja na drugi zadatak, pa sustav ne zna gdje se nalazi rukovatelj prekida za brojilo i tipkovnicu. Pomoću C naredbe `switch` ostvareno je pokretanje postavljenog tipa raspoređivača. Novi tip raspoređivača poslova nekog novog algoritma se ostvaruje ubacivanjem novog slučaja(`case`) u gornju `switch` naredbu. U ostvarenju nekog drugog algoritma potrebno je koristiti stanja kojima raspolaže `task.h` datoteka i voditi računa o tome što se događa kada korisnik iz ljuske promijeni tip

raspoređivača poslova, odnosno da li svaki tip raspoređivača poslova prepoznaje sva dostupna stanja zadataka. Stanja zadatka su detaljno objašnjena u poglavlju 4.1.4.

4.3 Poboljšanje ostvarenja algoritama raspoređivača poslova

Algoritmi koji su korišteni za raspoređivače poslova, mogu se dodatno poboljšati tako da se uvedu dodatne varijable u strukturu `TASK_STRUCT`. Na primjer, može se za svaki zadatak uvesti prioritet. Raspoređivač poslova prema jednakim intervalima je bolji od raspoređivača poslova po redu prispijeća jer osigurava da nijedan zadatak ne onemogući izvođenje drugih zadataka, ako neki od zadataka stoji u beskonačnoj petlji. Prema komentaru izvornog koda Linuxa 0.01 [5] prioritet trenutnog zadatka u izvođenju se može smanjivati za svako pokretanje raspoređivača, a ostalih povećavati, a ako dođe do 0 tada se taj zadatak zaustavlja i pokreće se idući zadatak sa najvećim prioritetom. Postoje mnogi algoritmi koji bi se mogli koristiti te druga poboljšanja ovisno o potrebama operacijskog sustava.

5. Primjeri korištenja ljuske i raspoređivača poslova

```
|
#123456789_______________________________________________
---------------------------------------------------
Examples:
Write to register: ra12 -> EAX=12
Write to memory: mw1000 -> [1000] = value
Read from memory: rw1000 -> print value of [1000]
Print registers: registers
TSS i LDT descriptors: taskdesc ; Print GDT table: printgdt
Clear screen: cls
Print tasks table: print tasks| pt ; Clear task print area: clts
Start task: start task | st ; Finish task: finish task | ft ; Run task: run task
 | rt
Scheduler info: sched ; Scheduler on/off: schedon/schedoff
Scheduler set freq: schedfreq set| schfs ; Scheduler set type: schedtype rr|sche
dtype fifo
Show task switch: show task switch | sts ; Hide task switch: hide task switch |
hts
Turn simulation on/off(default:on, not actually switching): simon/simoff
Turn trace onoff: ton | toff | trace on | trace off
Enter shell: press the ESC key
Quit shell: quit | q | exit
Addresses and number inputs are hexadecimal!!!
---------------------------------------------------
Input#:$
```

Slika 4.3-1 Popis naredaba MyOS-a

Slika 4.3-1 prikazuje ispis dostupnih naredbi pomoću `help` naredbe ljuske.

```
-
#123456789_______________________________________________
To start shell press ESC, if you made quit from shell
Shell inside kernel version 0.03:
Input#:$simoff
Scheduler type: SCHED_ROUND_ROBIN Frequency: 10 Sched_state: on Simulation_state
: off
Simulation off(running tasks and switching)
Input#:$st2
```

Slika 4.3-2 Isključivanje simulacije i pokretanje zadatka 2

Prema slici 4.3-2 ovdje isključujemo simulaciju i time omogućujemo stvarno prebacivanje izvođenja na drugi zadatak u ovom slučaju sa `TASK0` na zadatak 2, koji pokrećemo sa `st2`.

```
|998899909
#123456789_______________________________________________
TASK1.........FINISHED...........
TASK3.........STARTED...........
TASK4.........STARTED...........
TASK4 giving birth to 5 :)
new_task: Task "5" is already in run queue!!!
TASK5.........STARTED...........
TASK5 giving birth to 6 :)
new_task: Task "6" is already in run queue!!!
TASK5.........FINISHED...........
TASK6.........STARTED...........
TASK6 giving birth to 5 :)
TASK7.........STARTED...........
TASK7.........FINISHED...........
TASK8.........STARTED...........
TASK9.........STARTED...........
TASK9 killing TASK8, ups, that is "finishing"
finish_task("8"): forcibly FINISHED
TASK9.........FINISHED...........
TASK5.........STARTED...........
TASK5 giving birth to 6 :)
new_task: Task "6" is already in run queue!!!
TASK5.........FINISHED...........
TASK6.........FINISHED...........
```

Slika 4.3-3 Ispis nakon što se pokrene zadatak 2

Zadatak 2 se pokreće, ali on stvara sve druge zadatke osim sebe, koji stvaraju druge zadatke pomoću `new_task()`, a neki završavaju izvođenje drugih zadataka pomoću `finish_task()`što je vidljivo na slici 4.3-3.

6. Zaključak

Diplomski rad ostvaruje neke osnovne dijelove operacijskog sustava. Naravno, ti dijelovi su ostali nedorađeni, pa se pojavljuju greške ovisno o načinu korištenja ljuske operacijskog sustava.

Nastavak rada na ovom operacijskom sustavu uključivao bi izradu sustava za upravljanje memorijom uporabom straničenja prema poglavlju 2.1.3. Svaki zadatak bi dobio svoj zaštićeni memorijski prostor za segment instrukcija te segment podataka i stoga.

U ovom radu koristi se i386 arhitektura, ali uz neke promjene asemblerskog koda mogla bi se koristiti 64-bitna arhitektura procesora. Kako se to može učiniti može se pronaći u već navedenom poglavlju Intelovog priručnika[1]. Promjene se sastoje od korištenja 64-bitnih registara, te omogućivanja dodatnih zastavica koje omogućuju korištenje opisnika tablica sustava prema 64-bitnoj arhitekturi.

U sklopu ostvarenja sustava za upravljanje memorijom potrebno je napraviti gomilu sustava, kako bi se memorijski prostor mogao zauzimati dinamički u toku izvođenja.

Potrebno je ostvariti pogonitelje raznih uređaja za pohranu podataka, na primjer za čvrsti disk kako bi se sa njega moglo učitati strojni kod zadataka, te punitelj (engl. *loader*) i povezivač (engl. *linker*) operacijskog sustava koji bi ostvarili učitavanje zadataka dok je operacijski sustav u izvođenju.

Nakon toga poželjno je napisati programski kod ljuske operacijskog sustava kako bi ona bila zaseban zadatak, a da se onda svakom zadatku preko prekida tipkovnice šalje signal umjesto da ljuska odlučuje što će koji zadatak napraviti kada se pritisne tipka na tipkovnici.

Prema komentaru Linux 0.01 izvornog koda [4] poželjno je ostvariti više prividnih sučelja za pristupanje jezgri operacijskog sustava (engl. *virtual terminals*) kako bi se moglo izvoditi više raznih zadataka na posebnim sučeljima te kako bi svaki od zadataka imao prostor za ispis u području za ispis ljuske operacijskog sustava, kako bi se lakše pratilo njihovo izvođenje.

Na kraju, dovoljno je reći kako rad na razvoju operacijskog sustava iziskuje puno proučavanja raznih specifikacija, ali isto tako poznavanja osnova rada operacijskih sustava. Svakako je preporučljivo koristiti literaturu koja je navedena u ranijem seminarskom radu [2].

7. Popis korištene literature i pomoći

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, 2008.

[2] *Alati za izgradnju operacijskog sustava*, Antonio Arbutina, 2008.

[3] *Skelix OS,* http://en.skelix.org/skelixos

[4] *Bran's kernel development tutorial*, http://www.osdever.net/bkerndev/Docs/title.htm

[5] *The Linux Kernel 0.01 Commentary*, Gopakumar C.E., Pramode C.E., 2003.

[6] *Osdev.org,* http://wiki.osdev.org

[7] *GCC inline assembly*, http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

[8] *Grub memory map,* http://www.gnu.org/software/grub/manual/html_node/Memory-map.html#Memory-map

8. Sažetak

Naslov: Upravljanje procesima u jednostavnim operacijskim sustavima

Ključne riječi: arhitektura, operacijski sustav, raspoređivač poslova, zadatak, proces, straničenje, zaštita memorije

Sažetak: Ovaj rad ostvaruje osnovne zadaće operacijskog sustava: upravljanje procesima, odnosno raspoređivanje poslova, upravljanje prekidima te pozive jezgre. Arhitektura i386 je korištena zbog svoje dostupnosti korisnicima kućnih računala. U radu se pretpostavlja poznavanje osnova rada te poznavanja rada emulatora i raznih alata za razvoj operacijskog sustava. Nadogradnja ovog sustava podrazumijeva ostvarenje sustava za upravljanje memorijom, kao idući korak u razvoju sustava.

Title: Process management for simple operating systems

Keywords: architecture, operating system, task scheduler, task, process, paging, memory protection

Abstract: This thesis implements main functions of operating system: process management, that is task scheduling, interrupt handling and system calls. Architecture of i386 is used because it is available to users of personal computers. In this thesis, it is assumed foreknowledge of operating system basics, using emulators and different tools for operating system development. Upgrade of this system needs implementation of memory manager as the next step in system development.