

# Datoteke

Veljko Petrović

2023-04

## **Administrativni detalji**

Kako se organizuje predmet

### **O predavaču**

- Veljko Petrović
- pveljko@uns.ac.rs
- NTP 330
- Preliminarni termin konsultacija: Četvrtak 14:30, u kancelariji.
- **Uvek** je dobra ideja da se najavite za konsultacije.
- Možemo se dogovoriti i oko dodatnih termina za konsultacije u zavisnosti od mog i vašeg rasporeda i obaveza.
- Na predmetu učestvuju i prof. dr Dinu Dragan i prof. dr Dušan Gajić, koga bi trebalo da se sećate.

### **Asistenti**

- Anja Delić
- Dunja Gojković
- Dane Milišić
- Branislav Ristić
- Radovan Turović
- Jovana Jovanović

### **Obaveštenja**

- Jedino mesto gde možemo pouzdano da vam objavimo podatke je ACS sajt.
- Naročito važne su vesti o predmetu dostupne na <http://www.acs.uns.ac.rs/sr/os>
- Dok niste gotovi sa predmetom proveravajte ovo često da bi bili sigurni da ne propustite važne informacije.

## **Materijali**

- Operativni Sistemi, Problemi i Struktura prof. Hajdukovića
- Sav materijal će biti dostpuan na ACS sajtu.
- Nemojte koristiti materijal prošlih generacija: svake godine bude promena.
- Glavni izvor za predmet je, predvidivo udžbenik.
- Takođe važni su ovi slajdovi koji često sadrže modernizacije, i adaptacije materijala kao i modernije primere.

## **O slajdovima**

- Slajdovi su dostupni i kao interaktivran sajt dostupan iz svakog pretraživača i kao PDF dostupan na ACS sajtu
- I sajt i PDF su generisani iz istog izvora i ekvivalentni su, sa tim da sajt može da sadrži i, npr. animacije.
- Slajdovi za svako predavanje će biti dostupni pred čas i nalaziće se na:
  - Za **interaktivnu verziju** <https://pveljko-ftn.github.io/predavanja-OS/01/slides.html> gde se 01 menja sa brojem predavanja.
  - Za **PDF verziju** <http://www.acs.uns.ac.rs/sr/node/237/4408649>

## **O predavanjima**

- Periodično se pušta papir.
- Prisustvo je obavezno, formalno, ali mnogo bitnije prisustvo je *korisno*.
- Nikada nije pogrešno vreme da se postavi pitanje ili napravi komentar.

## **O vežbama**

- Automatska i ručna evidencija pristupa
- Apsolutno obavezno prisustvo
- Samostalna izrada zadataka
- Apsolutno se očekuje da se spremite za izradu zadataka pre svake vežbe
- Počinju od ove nedelje.
- Morate imati spremno okruženje za rad.
- Više o tome šta je 'okruženje za rad' kasnije danas.

## **Formiranje ocene**

Broj bodova	Ocena
51-60	6
61-70	7
71-80	8
81-90	9
91-100	10

## Odakle bodovi?

- Predispitne obaveze (do 70 bodova)
- Ispitne obaveze (do 30 bodova)

## Predispitne obaveze

Obaveza	Opis	Bodovi
Test T1234	Konkurentno programiranje	do 40
Složeni oblik vežbi	Konkurentno programiranje	do 30
SOV	(konkurentni problem)	

- Oba testa se rade na vežbama
- Imate *samo jednu priliku* da ih uradite tokom nastave, ali imamo pravo, za sada, da ponovimo predispitne obaveze
- To ponavljanje će biti posle kraja semestra ali mora biti pre granice za davanje bodova: dakle biće tokom leta.
- Ponavljanje se mora prijaviti i plaća se: to nije naša ideja niti naša želja, proceduralno je obavezno.

## Kako položiti?

- Prolaznu ocenu možete imati samo ako važi svaki od sledećih uslova:
- $T1234 + SOV \geq 36$
- $Ispit \geq 16$
- $T1234 + SOV + Ispit \geq 51$

## Kako pasti?

- Ako imate manje od 36 bodova sa predispitnih obaveza onda su svi bodovi koje imate nevažeći i morate predmet slušati opet iduće godine. Nećete dobiti potpis.
- Nemojte dozvoliti da ovo budete vi, **molim vas**.
- Stvarno, ali stvarno je bitno da steknete svojih 36 bodova. Nećete imati previše šansi da to učinite.

## Ispit

- Održava se u ispitnom roku
- Nosi najviše 30 bodova
- Namjenjen je isključivo studentima koji na predispitnim obavezama imaju barem 36 bodova
- Integralni ispit obuhvata celo gradivo.
- Mora se prijaviti ispit.
- Radi se na papiru, ima četiri pitanja i traje 60 minuta.

## Struktura ispita

- Prvo pitanje nosi 6 bodova, odnosi se na celo gradivo i biće baziрано na principu zaokruživanja. Ovo pitanje će uvek biti baziрано na listi pitanja koja će objaviti, ali neće odgovarati tačno nijednom pitanju, zato što ona nisu na zaokruživanje.
- Drugo pitanje nosi 10 bodova, odgovara se u okviru jednog do dva pasusa i mora biti sa liste pitanja koja će objaviti.
- Treće i četvrto pitanje nose zajedno 14 bodova i to tipično, ali ne garantovano, po 7 bodova i potpuno su slobodne forme: odnose se na celokupno gradivo predmeta bilo sa predavanja, udžbenika, slajdova i zahtevaju da se mogu povezati različite činjenice naučene u okviru predmeta i da se o njima može rezonovati.

## Šta neće nikada biti na ispitu

- Na ispitu nikada neće biti bilo koje pitanje koje očekuje da se kod uči napamet.
- Ako se, nekim slučajem, u pitanju pomene klasa ili neki algoritam, onda se isključivo misli na namenu te klase odnosno algoritma, na to kako radi i zašto postoji ali nikada i nikako koje su metode, šta tačno nasleđuje ili bilo šta slično.
- Ovo garancija se odnosi na sva četiri pitanja i na sve rokove, zauvek.

## Primer kompletног ispita sa odgovorima

- 1. Koja su validna stanja binarnog semafora?  
a) \*\*0\*\*      b) \*\*1\*\*      c) 2      d) SLOBODAN  
e) ZAUZET      f)-1      g) 3
- 2. Šta je mrtva petlja?  
- Mrtva petlja nastaje kada više niti/procesa pristupaju deljenim resursima koji su međusobno
- 3. Kakav format diska bi izabrali za particiju koja čuva video snimke sisitema za video-naz

- Koristio bih nekakav fajl sistem koji koristi kontinualne datoteke. Ovo je dobra ideja za...
- 4. Ako bi imali pristup direktno disku (blokovima) koliko bi vam minimalno trebalo pristup...
- Odgovor zavisi od fajl sistema, i okolnosti. Ako, kao na predavanjima, koristimo ext2fs, o...
- (Napomena: Molim vas, vodite računa da je ovaj odgovor malo neprecizan i preskače barem jed...

## Prepisivanje

- Ako se utvrdi da je neko prepisivao ili koristio bilo kakva nedozvoljena sredstva na proveri znanja bilo koje vrste, preduzeće se mere predviđenje za to pravilnikom fakulteta.
- Molim vas, molim vas nemojte. Oko svega možemo da se dogovorimo, sve može da se adaptira vašim potrebama, ali kada je prepisivanje u pitanju niko od vaših nastavnika na ovom predmetu nema smisao za humor.

## ChatGPT

- Nemojte ni ovo.
- Korišćenje bilo kakvog automatskog metoda generisanja koda se smatra prepisivanjem.
- Šta više, dok ima mnogih legitimnih svrha ovog softvera, ne preporučuje se nikako da ga koristite da radi vaš posao za vas: svrha zadatka koje dobijate je obrazovna, tj. nije svrha da se napravi rešenje (već imamo rešenja!) nego da vi prođete kroz proces razumevanja rešenja.
- Naročito, *naročito* nemojte da koristite da preko ChatGPT odgovarate na ispitna pitanja pa da iz toga učite.
- Jako jako puno vaših kolega je palo zbog ovoga.
- Takođe nemojte da učite iz bilo kakve skripte ili bilo kog izvora koji ne dolazi od mene.

## Potpis

- Potpis zahteva barem 36 bodova na predispitnim obavezama. Svako ko ima 36 bodova ili više na predispitnim ispitima, dobiće potpis.
- Nema drugih zahteva

## Tehničko okruženje

Kako biti spreman za vežbe i praćenje primera sa predavanja

### Tehničko okruženje

- Na ovom kursu se koristi Linux.
- Iako se primeri mogu kompajlirati na Windows-u ovo *nije preporučeno*.
- Zašto? Naleteće te na suptilne nekompatibilnosti koje su takve da niste dovoljno dobri programeri da ih otklonite.
- Možda jeste dovoljno dobri programeri, u kom slučaju, svaka čast.
- Ovo nije prva generacija: svako ko je koristio Windows u ovom kursu je naleteo na problem.
- Ako hoćete da koristite Windows i spremni ste da budete za to odgovorni, WSL je verovatno najbolji izbor.
- Od softvera se koristi GCC (kao kompjaler), GNU make i Visual Studio Code kao editor. Možete editovati kod ma kojim editorom: to neće uticati na vašu ocenu.
- Možete da instalirate Linux na vaš računar kao jedan od operativnih sistema i koristite to. Ali, ako vam je primaran OS Windows i ne želite da instalirate nešto novo, može i tako. Onda morate raditi u virtuelnoj mašini.

### Virtuelna mašina

- Trebaće vam VirtualBox <https://www.virtualbox.org/> koji se instalira kao i svaka druga aplikacija. Dostupan je za svaki operativni sistem.
- Dalje, trebaće vam ISO linux distribucije. Ako želite verziju koja je ista onoj u većini FTN laboratorija, to bi trebalo da je Ubuntu 20.04 LTS.
- Proces instalacije je bezbolan i trebalo bi da ne stvara ozbiljne probleme.
- Molim vas uradite ovo što pre

## Uvod

Vrtoglavica operativnog sistema

## Zadatak operativnog sistema

- Operativni sistem:
  - Objedinjuje raznorodne delove računara tako što upravlja procesorom, kontrolerima i RAM-om.
  - Skriva od korisnika detalje funkcionisanja tako što pretvara računar od mašine koja rukuje bitima, bajtima i blokovima u mašinu koja rukuje **datotekama i procesima**.

## Što baš datoteke i procesi?

- Datoteka predstavlja apstrakciju nad sposobnošću računara da čuva i pristupa nekakvim proizvoljnim podacima.
- Procesi predstavljaju apstrakciju nad sposobnošću računara da izvršava nezavisne tokove operacija nad podacima.

## Zašto nam treba operativni sistem?

- Sa tačke gledišta onoga što se zaista dešava na nivou hardvera, datoteka kojoj pristupate preko mreže, na hard disku magnetno-rotacionog tipa, i na SSD hard disku su *potpuno različite stvari*.
- To znači da ako pišete softver koji treba da radi sa sve tri varijante vi morate, onda, napisati poseban kod za sve tri mogućnosti, plus još i kod za fajlove koji se nalaze na optičkim medijima, plus još podršku za razni hardver koji se može koristiti da ostvari sve ove stvari.
- Ovako je nekada odista bilo.

## Zašto nam treba operativni sistem

- Tako da je jedna (veoma bitna) funkcija operativnog sistema da omogućava da softver koji pišete može da radi sa prijatnim apstrakcijama koje su uniformne umesto da direktno priča sa hardverom.
- To što stoji između vas i hardvera znači da operativni sistem može da uradi još dve jako bitne stvari:
  - Raspoređuje resurse između procesa koji se oko njih mogu takmičiti i omogućava sinhronizaciju
  - Omogućava *bezbednost*.

## Pojam datoteke

- Datoteka ima:
  - Sadržaj (korisničke podatke)

- Atribute (npr. veličina ili vreme kreiranja) - u deskriptoru datoteke
- Uloga datoteke:
  - Trajno čuvanje podataka.
  - Pristup podacima je čitanje i pisanje (kojima predstoji otvaranje i nakon kojih sledi zatvaranje datoteke).
- Što datoteke zatvaramo?

## Pojam procesa

- Aktivnost procesa - angažovanje procesora na izvršavanju korisničkog programa.
- Slika procesa - adresni prostor procesa (naredbe, stek i podaci).
- Atributi - stanje, prioritet - čuvaju se u deskriptoru procesa.

## Stanje i prioritet procesa

- Tipična stanja procesa su:
  - aktivan
  - čeka
  - spremam.
- Prioritet procesa određuje kada je proces aktivan:
- Skoro uvek je aktivan proces (odnosno procesi) sa najvišim prioritetom.
- Ako postoji nekoliko procesa sa najvišim prioritetom, vrši se raspodela procesorskog vremena između njih uz pomoć dva mehanizma:
  - Raspoređivanje između sistemskih niti
  - Kvantum/kvant

## Sistemski niti

- Moderni procesori tipično imaju sposobnost istinskog paralelnog izvršavanja, tj. mogu stvarno da rade stvari u paraleli
- Ako budete radili sa mikrokontrolerima, recimo, onda to obično nije dostupno
- Koliko jedan računar može da radi stvari u paraleli se zove broj *sistemskih niti*.
- Vama je ovo možda poznato kao broj 'jezgara' ali jedan računar može imati više procesora sa više jezgara a jedno individualno jezgro može raditi više poslova.

## **Kvantum/kvant**

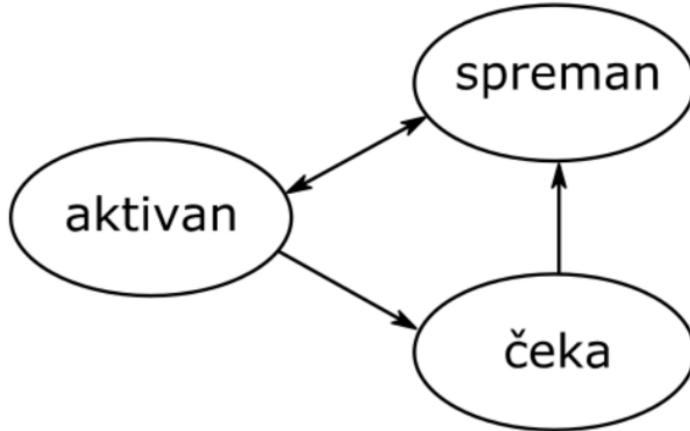
- Na mašinama koje nemaju više sistemskih niti ili na mašinama koje prekardaše svoj broj sistemskih niti (vrlo čest slučaj) ili na mašinama koje štede struju pa rade sa manje jezgara koristi se kvantum mehanizam
- Ovo znači se procesorsko vreme jedne sistemske niti deli između više programa gde se oni smenjuju, a svako dobija malecnu količinu vremena (poznatu kao kvant)
- Ko drži kvant se smenjuje tako brzo da ljudski operater neće primetiti da se operacije ne izvršavaju u paraleli.
- Isticanje kvantuma regulišu prekidi sata.

## **Terminološka zavrzlama**

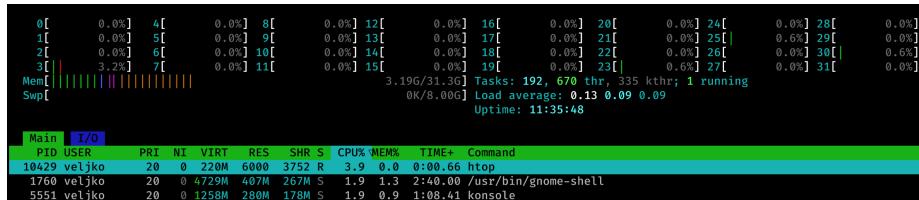
- Za potrebe ovog kursa 'kvantum' i 'kvant' je isto.
- Striktno kovoreći trebalo bi 'kvant' pošto je to adekvatan termin koji koristimo u fizici
- No, 'kvantum' vas podesća na taj termin na engleskom 'quantum.'
- Biće puno engleskih termina na ovom kursu: to je neizbežno, engleski je *lingua franca* računarskih nauka.

## **Stanje i prioritet procesa**

- Aktivan proces prelazi u stanje "čeka" kada je nemoguć nastavak njegove aktivnosti (npr. UI radnja).
- Nakon tog čekanja prelazi u stanje "spreman".



## Stanje i prioritet procesa, praktična ilustracija



## Uloga procesa

- Procesi omogućuju bolje iskorišćenje računara (procesora) i njegovu bržu reakciju na dešavanje spoljnih događaja (npr. unos teksta).
- Istovremeno postojanje više procesa omogućuje da se procesor preključi sa aktivnog procesa na spremam proces kada aktivan proces prelazi u stanje "čeka".
- Dobar primer ovakvog ponašanja je čekanje hitnog procesa na spoljni događaj (npr. unošenje teksta sa tastature).

## Pojam niti

- Redosled naredbi programa (procesa) naziva se trag (trace) procesa.
- Proces je sekvencijalan ako je njegov trag poznat u vreme programiranja za dati ulaz.

- Trag sekvencijalnog procesa naziva se nit (thread) koja povezuje izvršavane naredbe u redosledu njihovog izvršavanja.

## **Mana sekvencijalnih procesa**

- Mana sekvencijalnih procesa je da su neosetljivi na spoljne događaje (npr. editovanje teksta).
- Za editovanje su potrebne dve radnje (interakcija sa korisnikom i čuvanje unešenog teksta).
- Sekvencijalan editorski proces izvršava te dve radnje jednu za drugom:

## **Pseudokod 1**

```
for(;;) {
    do_editor_command();
    if(time_to_save_data())
        save_data();
}
```

## **Pseudokod nesekvencijalnog editorskog procesa**

```
for(;;) {
    do_editor_command_in_foreground();
}
...
for(;;) {
    if(time_to_save_data())
        save_data_in_background();
}
```

## **Nesekvencijalan editorski proces (odvojene radnje)**

- Prioritetnija radnja je posvećena interakciji sa korisnikom, a manje prioritetna pozadinska radnja je posvećena čuvanju teksta.
- Pod prepostavkom da je hitna radnja zaustavljena, jer nema komandi od korisnika, pozadinska radnja može da se odvija sve dok, na primer, spoljni događaj poput pritiska dirke na tastaturi ne najavi početak interakcije sa korisnikom.
- Tada se zaustavlja pozadinska radnja, radi nastavljanja hitne radnje.

- Kada se obavi korisnička komanda u okviru hitne radnje, a hitna radnja se zaustavi u očekivanju nove komande, nastavlja se pozadinska radnja.
- Zahvaljujući preplitanju hitne i pozadinske radnje, u toku editiranja nema perioda bez odziva.
- Podrazumeva se da opisanom nesekvencijalnom editorskom procesu odgovaraju dve niti.
- Da bi opisano rukovanje nitima bilo moguće, prioritet, stanje i stek se ne vezuju za proces, nego za njegove niti.
- Znači svaka nit procesa ima svoj prioritet, svoje stanje, svoj stek, pa i svoj deskriptor.
- Za niti istog procesa se podrazumeva da nisu potpuno nezavisne, odnosno da sarađuju razmenom podataka.
- Tako, u slučaju nesekvencijalnog editorskog procesa, manje prioritetna nit se brine o čuvanju teksta koga pripremi prioritetna nit.

## **Konkurentni procesi**

- Procesi sa više niti nazivaju se konkurentni procesi (konkurentni programi).
- U nekim okruženjima samo jedna od niti može biti "aktivna", dok su ostale u stanju "spremna" ili "čeka".
- U nekim (kao ono u kome mi pišemo) proizvoljan podskup niti može biti aktivan, ograničen samo mogućnostima računara.
- Preključivanje procesora sa jedne niti na drugu, uzrokuju redosled izvšavanja koji nije određen u vreme programiranja, što znači da je izvršavanje konkurentnih procesa u opštem slučaju stohastično zbog stohastične prirode dešavanja spoljnih događaja.

## **Terminološka zavrzlama**

- Prvo, 'konkurentnost' u ovom kontekstu nema *nikavke* veze sa tim kako program uspeva na tržištu: reči su slučajno iste.
- Drugo, možda neki od vas razmišljaju da ovi 'konkurentni' programi zvuče neobično mnogo kao *paralelni* programi.
- Ovo su veoma slični termini ali se razlikuju suptilno.

## Konkurentnost vs. paralelnost

- Konkurentni programi su programi koji imaju različite zadatke (niti) koje se izvršavaju sa preklapanjem u nekom periodu vremena.
- Paralelni programi imaju različite zadatke (niti) koje se izvršavaju jednovremeno u istom fizičkom trenutku. Paralelnost zahteva hardver koji to podržava, tj. podržava jednovremeno izvršavanje različitih poslova.
- Postoji razlika i u nameni, kako se terminologija koristi: konkurentni programi (tipično) izvršavaju više različitih zadataka sa preklapanjem u jednom periodu vremena. Paralelni programi (tipično) izvršavaju više delova jednog zadatka zbog ubrzanja.

## Proces i nit

- Može da deluje zbumujuće što imamo dva metoda za konkurentno programiranje
- Uprkos tome, proces i nit imaju odvojene funkcije iako su procesi, istorijski, korišćeni i da obezbede posao niti
- Nit služi da u okviru *jednog procesa* omogući konkurentnost
- Tu je konkurentnost takva da između niti nema nikakve separacije: svaka nit ima pristup svim podacima svake niti za maksimalne performanse.
- Zato svaka nit ima svoj stek i svoj deskriptor, ali nema svoju sliku: to deli sa matičnim procesom.
- Proces pruža konkurentnost i istorijski se ponekad koristio umesto niti, ali mu je glavna funkcija i osobina *separacija*
- Jedan proces ne može da 'zaviri' u drugi bez eksplisitne dozvole koja je vrlo kompleksna da se ostvari.
- Odnosno, kada se to desi, to bude bezbednosna mana i veliki problem.
- Da li postoji razlog u modernom programu da se konkurentnost omogući kroz procese a ne kroz niti?
- Da! *Bezbednost*.
- Chrome izvršava svaki tab u posebnom procesu i to znači da je potencijalno maliciozan JS kod u jednom tab-u izolovan, višestruko, od nekog drugog tab-a gde je, recimo, vaš e-banking.

## **Struktura operativnog sistema**

- Zadatak operativnog sistema je da upravlja fizičkim i logičkim delovima računara u okviru svog jezgra (kernela).
- Fizičkim delovima upravljaju moduli za rukovanje:
  - Procesorom
  - Kontrolerima
  - Radnom memorijom
- Logičkim delovima upravljaju moduli za rukovanje:
  - Datotekama
  - Procesima

## **Modul za rukovanje procesorom**

- Zadatak ovog modula je preključivanje jedne niti na drugu.
- Moguće je preključivanje na niti u istom procesu ili u različitim procesima.
- Preključivanje procesora između niti istog procesa je brže nego preključivanje niti u okviru različitih procesa zato što se niti istog procesa nalaze u istom adresnom prostoru.
- Ovaj modul uvodi operaciju preključivanja.

## **Modul za rukovanje kontrolerima**

- Zadatak ovog modula je upravljanje ulaznim i izlaznim uređajima koji su zakačeni za kontrolere.
- Modul se sastoji od niza komponenti nazvanih drajveri.

## **Drajveri**

- Cilj drajvera jeste da uređaje predstavi u apstraktnom obliku sa jednoobraznim i pravilnim načinom korišćenja (primer drajvera diska).
- Drajveri uvođe operacije ulaza i izlaza, u okviru kojih se rukuje preključivanjem (zaustavljanjem) niti koja je zatražila operaciju.
- Drajveri takođe rukuju i obradom prekida kao reakcijom na javljanje uređaja putem mehanizma prekida.

## **Modul za rukovanje radnom memorijom**

- Zadatak ovog modula je da vodi evidenciju o slobodnoj radnoj memoriji radi zauzimanja i oslobođanja.
- U slučaju da podržava virtualnu memoriju, ovaj modul se brine i o prebacivanju sadržaja između radne i masovne memorije.
- Ovaj modul uvodi operacije zauzimanja i oslobođanja.

## **Modul za rukovanje datotekama**

- Zadatak modula za rukovanje datotekama je da omogući otvaranje i zatvaranje datoteka, kao i čitanje i pisanje njihovog sadržaja.
- Ovaj modul vodi evidenciju o blokovima (HDD/SSD) u kojima se nalaze sadržaji datoteka.
- Takođe ovaj modul vodi računa o prebacivanju sadržaja između radne i masovne memorije uz pomoć operacija čitanja (ulaza) i pisanja (izlaza), kao i bafera potrebnih za smeštanje sadržaja (modul za rukovanje memorijom).
- Pored ovog modul uvodi i operacije otvaranja i zatvaranja.

## **Modul za rukovanje procesima**

- Zadatak ovog modula je da omogući stvaranje i uništavanje procesa, kao i stvaranje i uništavanje njihovih niti.
- Na ovaj način se uvodi višeprocesni i višenitni režim rada koji omogućava:
  - Bolje iskorišćenje procesora
  - Podršku većeg broja korisnika
  - Bržu reakciju na spoljne događaje
- Modul za rukovanje procesima poziva i operacije drugih modula (za upravljanje datotekama i memorijom).
- Modul za rukovanje procesima uvodi operacije stvaranja i uništavanja (procesa i niti).

## **Slojevit operativni sistem**

modul za rukovanje procesima
modul za rukovanje datotekama
modul za rukovanje radnom memorijom
modul za rukovanje kontrolerima
modul za rukovanje procesorom

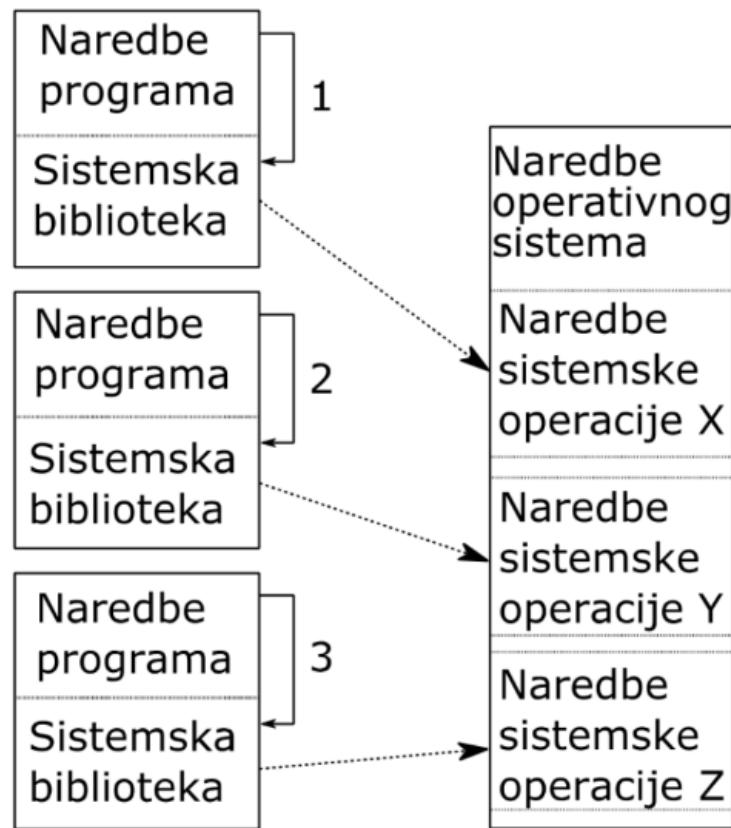
## **Slojeviti operativni sistem**

- Za razliku od slojevitog operativnog sistema u praksi se uglavnom sreću monolitni operativni sistemi, koji nemaju hijerarhijsku strukturu jer saradnja nije ograničena kao kod slojevitog OS.

## **Sistemski pozivi**

- Svaki proces se nalazi u korisničkom sloju (gornjem sloju OS), i poseduje poseban adresni prostor koji se naziva korisnički prostor (user space).
- Operativni sistem poseduje poseban adresni prostor koji se naziva sistemski prostor (kernel space).
- Zbog razdvojenosti ova dva prostora neophodno je uvođenje sistemskih poziva, radi poziva operacija OS.
- Sistemski pozivi zahtevaju korišćenje ASM naredbi i sakrivaju se unutar sistemskih potprograma (sistemske operacije).
- Svaki proces u sistemskom prostoru ima svoj sistemski stek.
- Sistemski potprogrami obrazuju sistemsku biblioteku.
- Zahvaljujući sistemskim potprogramima, odnosno sistemskoj biblioteci, operativni sistem predstavlja deo korisničkog programa, iako za njega nije direktno linkovan.
- Istovremeno postojanje više procesa i nepredvidivost preključivanja, uzrokuje da je moguće da istovremeno postoji više procesa, koji su započeli, a nisu završili svoju aktivnost u okviru operativnog sistema, odnosno, čija aktivnost je zaustavljena unutar sistemskih operacija operativnog sistema.

## Preplitanje izvršavanja tri sistemske operacije



## Praktična ilustracija sistemskih poziva

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    unsigned char buff[32];
    ssize_t count = 0;
    int fd = open('open.c', O_RDONLY);
    while(count = read(fd, buff, 31)){
        buff[count] = '\0';
        printf("%s", buff);
```

```
    }
    close(fd);
}
```

## Interakcija korisnika i OS

- Komande komandnog jezika omogućavaju korišćenje OS na interaktivnom nivou.
- Za interpretiranje i preuzimanje komandi komandnog jezika zadužen je poseban proces iz korisničkog sloja koji se zove interpreter komandnog jezika (shell).
- Interpreter komandnog jezika koristi OS na programskom nivou, jer u toku svog rada poziva sistemske operacije.

## Pitanja

Šta smo naučili?

## Pitanja

- Koje poslove obavlja operativni sistem?
- Šta obuhvata pojam datoteke?
- Šta se nalazi u deskriptoru datoteke?
- Šta omogućuju datoteke?
- Šta obavezno prethodi čitanju i pisanju datoteke?
- Šta sledi iza čitanja i pisanja datoteke?
- Šta obuhvata pojam procesa?
- Šta se nalazi u deskriptoru procesa?
- Koja stanja procesa postoje?
- Kada je proces aktivran?
- Šta je kvantum?
- Šta je sistemska nit?
- Šta se dešava nakon isticanja kvantuma?
- Po kom kriteriju se uvek bira aktivran proces?
- Koji prelazi su mogući između stanja procesa?
- Koji prelazi nisu mogući između stanja procesa?

- Šta omogućuju procesi?
- Šta karakteriše sekvensijalni proces?
- Šta karakteriše konkurentni proces?
- Šta ima svaka nit konkurentnog procesa?
- Koje su razlike između procesa i niti?
- Koju operaciju uvodi modul za rukovanje procesorom?
- Po čemu se razlikuju preključivanja između niti istog procesa i preključivanja između niti raznih procesa?
- Koje operacije uvodi modul za rukovanje kontrolerima?
- Šta je cilj drajvera?
- Koje operacije uvodi modul za rukovanje radnom memorijom?
- Koje operacije poziva modul za rukovanje radnom memorijom kada podržava virtuelnu memoriju?
- Koje operacije uvodi modul za rukovanje datotekama?
- Koje operacije poziva modul za rukovanje datotekama?
- Šta omogućuju multiprocesing i multithreading?
- Koje operacije uvodi modul za rukovanje procesima?
- Koje operacije poziva modul za rukovanje procesima?
- Koje module sadrži slojeviti operativni sistem?
- Šta omogućuju sistemski pozivi?
- Koje adresne prostore podržava operativni sistem?
- Šta karakteriše interpreter komandnog jezika?
- Koji nivoi korišćenja operativnog sistema postoje?

## **Primeri štetnog preplitanja**

### **Svojstva konkurentnih programa**

- Mešanje izvršavanja raznih niti ili niti i obrađivača prekida naziva se *preplitanje* (eng. interleaving).
- Preplitanje niti i obrada prekida imaju slučajan karakter jer unapred nije poznato kada će se desiti prekid i preključivanje.
- Stohastično izvršavanje konkurentnih programa može da menja rezultate izvršavanja od slučaja do slučaja, što može dovesti do pojave štetnog preplitanja.

## Primeri štetnog preplitanja

- Primeri štetnog preplitanja su mogući i u OS (sistemske pozivne nizove i mogu se preplitati sa obrađivačima prekida).
- Rukovanje pozicijom kurzora je dobar primer štetnog preplitanja.

## Rukovanje pozicijom kurzora

```
1 class Position {  
2     int x, y;  
3 public:  
4     Position();  
5     void set(int new_x,int new_y);  
6     void get(int* current_x,int* current_y);  
7 };  
8 Position::Position(){  
9     x = 0;  
10    y = 0;  
11 }  
12  
13 void Position::set(int new_x, int new_y){  
14     x = new_x;  
15     y = new_y;  
16 }  
17 void Position::get(int* current_x, int* current_y){  
18     *current_x = x;  
19     *current_y = y;  
20 }  
21 Position position;  
22  
23 - Neka operacija `position.set()` bude ono što koristi obrađivač prekida da podesi poziciju  
24 - Neka operacija `position.get()` bude na raspolaganju procesima iz korisničkog sloja i služi  
25 - Tada je moguće da u toku izvršavanja operacije `position.get()` proces bude prekinut ili poništ  
26 - Ako se to desi dok je izvršavanje get u liniji 17, a izvršavanje set stigne do linije 14, tada će se
```

## Rukovanje slobodnim baferima

```
1 struct List_member {  
2     List_member* next;  
3     char buffer[512];  
4 };  
5  
6 class List {  
7     List_member* first;  
8 public:
```

```

9     List() : first(0) {};
10    void link(List_member* member);
11    List_member* unlink();
12 };
13
14 void List::link(List_member* member){
15     member->next=first;
16     first=member;
17 }
18
19 List_member* List::unlink(){
20     List_member* unlinked;
21     unlinked=first;
22     if(first != 0)
23         first=first->next;
24     return unlinked;
25 }
26
27 List list;
28
29 - Neka su operacije `list.link()` i `list.unlink()` na raspolaganju samo modulu za rukovanje
30 - Tada je moguće da izvršavanje operacije `list.unlink()` bude pokrenuto u toku aktivnosti neke
31 - Kao rezultat izvršavanja ove operacije na steku pomenute niti nastane primerak njene lokalne
32 - Izvršavanje iskaza `unlinked = first` smešta u ovaj primerak lokalne promenljive adresu pr
33
34 - Neka, nakon izvršavanja prethodnog iskaza, pod uticajem obrade prekida dođe do preključiva
35 - Tako na steku i ove druge niti nastaje njen primerak lokalne promenljive `unlinked`.
36 - Posledica ovakvog sleda događaja je da posmatrana dva procesa koriste isti bafer. To nem
37 - Prema tome, preplitanje dva izvršavanja operacije `list.unlink()`, je štetno. Isto važi i

```

## Rukovanje komunikacionim baferom

- Štetna međusobna preplitanja niti su moguća i za niti koje pripadaju istom procesu.
- Ovakva saradnja se može ostvariti tako što jedna od niti šalje podatke drugoj niti.
- Takva razmena podataka između niti se obično obavlja posredstvom komunikacionog bafera.
- Nit koja puni ovaj bafer podacima ima ulogu proizvođača (podataka), a nit koja prazni ovaj bafer ima ulogu potrošača (podataka).
- U pojednostavljenom slučaju, rukovanje komunikacionim baferom obuhvata punjenje: `put()` celog bafera, kao i pražnjenje: `get()` celog bafera.

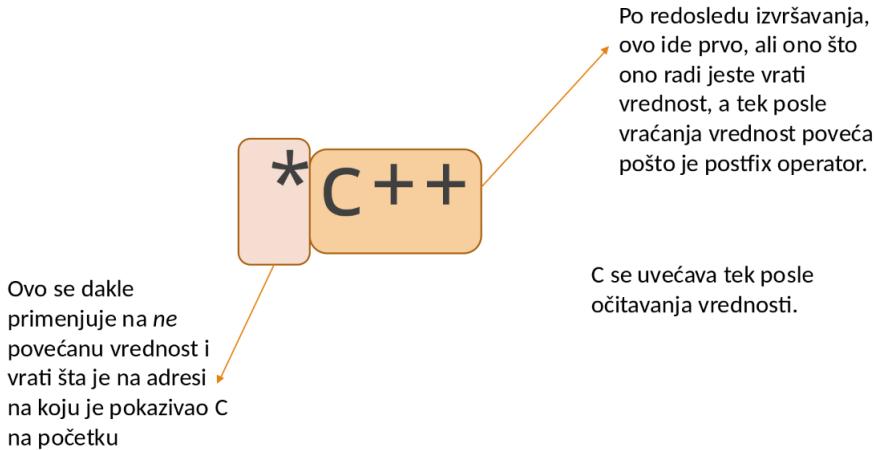
```
1 const unsigned int BUFFER_SIZE = 512;
```

```

2
3 class Buffer {
4     char content[BUFFER_SIZE];
5 public:
6     Buffer() {};
7     void put(char* c);
8     void get(char* c);
9 };
10
11 void Buffer::put(char* c){
12     unsigned int i;
13     for(i = 0; i < BUFFER_SIZE; i++)
14         content[i] = *c++;
15 }
16
17 void Buffer::get(char* c){
18     unsigned int i;
19     for(i = 0; i < BUFFER_SIZE; i++)
20         *c++ = content[i];
21 }
22 Buffer buffer;

```

## C haiku



## Rukovanje komunikacionim baferom

- Operaciju `buffer.put()` poziva nit proizvođač.
- Operaciju `buffer.get()` poziva nit potrošač.

- U ovoj situaciji moguće je, da se desi prekid sata za vreme aktivnosti proizvođača u operaciji `buffer.put()`.
- Ako operacija `buffer.get()` bude pozvana u toku aktivnosti potrošača, tada postoji mogućnost da potrošač preuzme sadržaj delimično popunjeno bafera.

## Sprečavanje štetnih preplitanja

### Međusobna isključivost

- Promenljive kojima pristupaju više niti ili niti i obrađivači prekida istovremeno (`position`, `list` i `buffer`) se nazivaju **deljene promenljive**, a klase koje opisuju te promenljive **deljene klase**.
- Deljene klase su pravljene pod pretostavkom da se rukovanja deljenim promenljivim obavljaju sekvensijalno odnosno strogo jedna za drugom.
- Štetna preplitanja negiraju pomenutu pretpostavku jer dopuštaju da novo izvršavanje neke operacije deljene promenljive započne pre nego što se završilo već započeto izvršavanje neke od operacija te promenljive.
- Problem štetnih preplitanja ne postoji ako se obezbedi međusobna isključivost (mutual exclusion) izvršavanja operacija deljenih promenljivih.

### Kritične sekcije i sinhronizacija

- Tela operacija deljenih klasa ili delovi ovih tela, čije izvršavanje je kritično za konzistentnost deljenih promenljivih, se nazivaju **kritične sekcije**.
- Međusobna isključivost kritičnih sekcija se ostvaruje sinhronizacijom pristupa.
- Pored obične sinhronizacije postoji i uslovna sinhronizacija.

### Fundamentalna Rešenja problema međusobne isključivosti

- Postoje dva fundamentalna pristupa problemu međusobne isključivosti:
  - Pristup baziran na softveru
  - Pristup baziran na hardveru

## Atomski regioni

- U primeru rukovanja pozicijom cursora, štetna preplitanja nastupaju kao posledica obrade prekida.
- Atomski regioni omogućavaju neprekidnost izvršavanja unutar kritičnih sekcija deljene promenljive.
- Onemogućenje prekida odlaže obradu novih prekida i usporava reakciju procesora, pa atomski regioni treba da budu što kraći.

## Propusnice i isključivi regioni

- Propusnice su drugi način ostvarenja međusobne isključivosti kritičnih sekcija.
- Propusnica može biti slobodna ili zauzeta.
- Samo jedna nit od svih niti koje se takmiče za propusnicu dobija istu i ulazi u kritičnu sekciju, dok sve ostale niti zaustavljaju svoju aktivnost i prelaze u stanje "čeka".
- Kada nit koja napušta kritičnu sekciju oslobodi propusnicu (vrati), neka sledeća nit dobija propusnicu i prelazi iz stanja "čeka" u stanje "spremna".
- Nit, koja tada dobije propusnicu, odmah prelazi iz stanja "čeka" u stanje "spremna", ali u kritičnu sekciju ulazi tek kada postane aktivna (odnosno, kada se procesor preključi na nju).
- Rukovanje propusnicom deljene promenljive je, takođe, ugroženo štetnim preplitanjima.
- Zbog toga se konzistentnost propusnica mora zaštititi.
- To se može učiniti kroz algoritam ili kroz hardversku podršku

## Algoritmi koji štite propusnicu

- Ovo je plodno polje istraživanja koje je *naročito* važno ako se bavite distribuiranim sistemima (nešto više o ovome kasnije)
- Ovde ćemo baciti pogled na Lamportovo rešenje poznato još i kao "Lamportova Pekara"
- Lamportova Pekara je mehanizam gde zamišljamo da imamo pekaru (radila bi i neka druga prodavnica) gde mušterije uzmu broj da bi ušle u prodavnici.
- Na ulazu je znak koji prikazuje koji se broj uslužuje

## Lamportova Pekara

- Algoritam je onda da uzmemo broj i čekamo (bukvalno čekanje: koristimo petlju za čekanje - tehnika koja je još poznata i kao spinlock ili busy-wait) dok naš broj nije 'na redu'
- Pošto se svi možemo složiti oko toga koji je broj veći ili manji, dok god možemo da delimo memoriju, možemo da se sinhronizujemo
- Algoritam se mora adaptirati za upotrebu na računaru (a ne u pekari) ali centralna ideja je ista
- Uvećemo prvo dva niza, Choosing i Number
- Prvi je bool niz koji je 1 ako je neka nit u procesu biranja broja
- Drugi je niz brojeva koje niti uzimaju

## Lamportova Pekara - pseudokod

```
1  bool choosing[NUM_THREADS] = {false};
2  int number[NUM_THREADS] = {0};
3  void lock(int i){
4      choosing[i] = true;
5      number[i] = 1 + max(number);
6      choosing[i] = false;
7      for(int j = 0; j < NUM_THREADS; j++){
8          while(choosing[j]); //busy-wait
9          while
10          (
11              (number[j] != 0) &&
12              (
13                  (number[j] < number[i]) ||
14                  (number[j] == number[i] && j < i)
15              )
16          ); //busy-wait
17      }
18  }
```

## Lamportova Pekara

- Primetite da je ograničenje Lamportove pekare da može da se desi da dve niti dobiju isti broj
- To nije problem: u tom slučaju broj **niti** postaje prioritet
- Jedino što je onda potrebno je imati brojeve niti koje se ne ponavaljaju

## Lamportova Pekara - pseudokod

```
19 void unlock(int i){  
20     number[i] = 0;  
21 }  
22  
23 void nit(int pid){  
24     while(true){  
25         lock(pid);  
26         //Kritična sekcija  
27         unlock(pid);  
28     }  
29 }
```

## Ograničenje Lamportove pekare i sličnih algoritama

- Glavni problem sa svim algoritmima ove vrste jeste što se zasnivaju na tome da se kod izvršava *tačno kako piše*
- Ovo apsolutno nije nešto u šta možemo da se uzdamo na modernim sistemima gde je često izvršavanje koda van redosleda
- Ako imamo izvršavanje koda van redosleda, onda algoritmi ovog tipa *neće raditi*.

## Hardverska podrška

- Hardverska podrška je ono što se koristi na jedinstvenim računarima, tj. van distribuiranih sistema.
- Jednostavno je, brzo i efektno
- Na procesorima jednostrukog tipa se koriste *atomski regioni* koji su konstruisani tako što se zabrani obrada prekida. Bez prekida na takvim sistemima nema preplitanja.
- Procesori sa više sistemskih niti zahtevaju nešto malo komplikovanije.

## Zabranjana prekida

- Ako imamo jednostruk procesor jedini način na koji on može prestati da izvršava sekvensu instrukcija koju je uzeo da izvršava jeste da se desi prekid
- Taj prekid može da nešto promeni i da vrati izvršavanje (kao što je slučaj sa obrađivačem za kurzor ili vreme), ali jednakom lako može i da potpuno promeni šta se izvršava budući da obrađivač prekida može da promeni pointer instrukcija i sadržaj registara i steka i time potpuno i permanentno izmeni šta sistem radi.

- Ako *zabranimo* prekid što se relativno lako može uraditi kroz instrukciju procesora koja tome služi (sti i cli na x86\_64 arhitekturi), onda garantujemo da nas niko drugi neće prekinuti
- Ako nema potrebe da se bavimo nečim drugim, mi predpostavljamo da koristimo baš ovaj pristup, mada se on u praksi ne koristi.

### **Šta na procesorima sa više sistemskih niti?**

- Jedan od metoda jeste da se zabrane prekidi i zaključa memorjska magistrala
- Ovim se garantuje da niti može da se desi prekid nit druga nit može da dirne memoriju
- U praksi ovo je previše sporo
- Umesto toga, kada je samo potrebna međusobna isključivost između dve korisničke niti onda se koriste posebne instrukcije koje implementiraju sve što je potrebno da bi se implementirala propusnica.

### **Poredi i zameni**

- Poredi i zameni (eng. Compare and Swap) je instrukcija koja postoji na nekim arhitekturama i koristi se kao gradivni element da implementira različite primitive sinhronizacije.
- Na Intel arhitekturi ovo se zove CMPXCHNG i (uz prefiks LOCK) omogućava da se vrlo jednostavno napravi struktura koja bezbedno upravlja memorijom propusnice
- Ako pogledamo uputstvo možemo da vidimo kako je to Intel implementirao.

### **Citat iz Intel uputstva za x86-64 arhitekturu 7.3.1.2**

“The CMPXCHG (compare and exchange) and CMPXCHG8B (compare and exchange 8 bytes) instructions are used to synchronize operations in systems that use multiple processors. The CMPXCHG instruction requires three operands: a source operand in a register, another source operand in the EAX register, and a destination operand.”

“If the values contained in the destination operand and the EAX register are equal, the destination operand is replaced with the value of the other source operand (the value not in the EAX register). Otherwise, the original value of the destination operand is loaded in the EAX register.”

“The status flags in the EFLAGS register reflect the result that would have been obtained by subtracting the destination operand from the value in the EAX register. The CMPXCHG instruction is commonly used for testing and modifying semaphores. It checks to see if a semaphore is free. If the semaphore is free, it is marked allocated; otherwise it gets the ID of the current owner.”

“This is all done in one uninterruptible operation. In a single-processor system, the CMPXCHG instruction eliminates the need to switch to protection level 0 (to disable interrupts) before executing multiple instructions to test and modify a semaphore. For SMP systems, CMPXCHG can be combined with the LOCK prefix to perform the compare and exchange operation atomically.”

### Kako dalje

- Nije bitno (mnogo) koja tehnika se koristi da se implementira propusnica
- Bitno je da radi
- Za naše potrebe, ako nas baš ne zanima neka varijacija na temu, plan će da se pratvaramo da imamo jednostruk procesor gde koristimo zabranu prekida zato što je to najprostije za razumevanje.

### Propusnice i isključivi regioni

- Interesantno je uočiti da preključivanja u isključivim regionima omogućuju pojavu štetnih preplitanja, ali i omogućuju njihovo sprečavanje.
- Tako, ako nit, čiju aktivnost je omogućilo preključivanje, pokuša da uđe u isključivi region deljene promenljive sa zauzetom propusnicom, tada opet preključivanje dovodi do zaustavljanja aktivnosti ove niti.
- Prema tome, prvo preključivanje je stvorilo uslove za pojavu štetnog preplitanja, a drugo preključivanje je sprečilo tu pojavu.

### Poželjne osobine konkurentnih programa

- Poželjne osobine konkurentnih programa uvode:
  - Tvrđnu isključivanja nepoželjnog (safety property).
  - Tvrđnu uključivanja poželjnog (liveness property).
- Primer tvrdnje isključivanja nepoželjnog je tvrdnja da se u izvršavanjima konkurentnog programa ne javlja nekonzistentnost date deljene promenljive.

- Primer tvrdnje uključivanja poželjnog je tvrdnja da se u toku izvršavanja konkurentnog programa dese svi zatraženi ulasci u dati isključivi region.
- Za konkurentni program se može reći da ima neku od poželjnih osobina samo ako se dokaže (na neformalan ili formalan način) da važi tvrdnja kojoj pomenuta osobina odgovara.
- Ovakvo rezonovanje o ispravnosti konkurentnog programa je neophodno, jer slučajna priroda preplitanja može da bude uzrok nedeterminističkog (nepredvidivog) izvršavanja konkurentnog programa.
- U takvoj situaciji, ponovljena izvršavanja konkurentnog programa, u toku kojih se obrađuju isti podaci, ne moraju da imaju isti ishod.
- Zbog toga, provera ispravnosti konkurentnog programa ne može da se zasniva samo na pokazivanju da pojedina izvršavanja konkurentnog programa imaju ispravan rezultat, jer tačan rezultat, dobijen u jednom ili više izvršavanja, ne isključuje mogućnost postojanja izvršavanja koja za iste ulazne podatke daju netačan rezultat.

## **Programski jezici za konkurentno programiranje**

- Konkurentno programiranje se razlikuje od sekvencijalnog po rukovanju nitima i deljenim promenljivama.
- Konkurentni programski jezik može nastati kao rezultat pravljenja potpuno novog programskega jezika ili kao rezultat proširenja postojećeg sekvencijalnog programskega jezika konkurentnim iskazima.
- Konkurentna biblioteka omogućuje da se za konkurentno programiranje koristi već postojeći, poznat programski jezik.

## **Uvod u konkurentnu biblioteku C++**

- Međunarodni standard C++11 predviđa rukovanje nitima i deljenim promenljivama.
- Implementacije C++11 standarda i kasnijih revizija su dužne da pruže nekakvu implementaciju ovih standardnih osobina jezika
- Na Linux platformi GCC počevši od verzije ~5 podržava sve što je potrebno
- Potrebna je pomoć operativnog sistema da se niti implementiraju kako valja
- Ta pomoć se pruža preko POSIX Threads (pthreads) mehanizma.

## Klasa thread

```
1 void thread_example(){
2     double x;
3     cout << "ZADAJ VREDNOST BROJA X" << endl;
4     cin >> x;
5     cout << endl << "x = " << x << endl;
6 }
7
8 int main(){
9     thread example(thread_example);
10    example.join();
11 //example.detach();
12 }
13
14 - Sve što nam je potrebno jeste da napravimo instancu klase `thread` kojoj prosledimo funkciju
15 - U tom trenutku nit počinje da radi
16 - Ono što nam je ostalo i što moramo da uradimo jeste da definišemo odnos između niti koja je
17 - Imamo dva izbora: `join` i `detach`
```

### join i detach

- Prva opcija kaže da će nit koja je pokrenula drugu nit (nit-roditelj i nit-dete su tipični termini) da pauzira izvršavanje dok nit-dete ne terminira.
- Većinu vremena ovo je tačno ono što želimo zato što ne želimo da se nit `main` terminira (zajedno sa programom) dok nam se sve niti nisu završile.
- Druga opcija služi kada nam ne treba da čekamo da se nit-dete završi (možda se izvršava u beskonačnoj petlji) i jednostavno terminiramo program kada stignemo do kraja `main` niti.

## Primer štetnog preplitanja

```
int main(){
    thread example1(thread_example);
    thread example2(thread_example);
    example1.join();
    example2.join();
}
```

- Šta je ovde deljeni resurs?
- Šta izaziva štetno preplitanje?
- Kako bi se manifestovalo?

## Sprečavanje štetnog preplitanja 1

```
1 mutex terminal;
2
3 void thread_example(){
4     double x;
5     terminal.lock();
6     cout << "ZADAJ VREDNOST BROJA X" << endl;
7     cin >> x;
8     cout << endl << "X = " << x << endl;
9     terminal.unlock();
10 }
```

## Sprečavanje štetnog preplitanja 2

```
1 mutex terminal;
2
3 void thread_example(){
4     double x;
5     unique_lock<mutex> lock(terminal);
6     cout << "ZADAJ VREDNOST BROJA X" << endl;
7     cin >> x;
8     cout << endl << "X = " << x << endl;
9 }
```

## Uslovna sinhronizacija

- Sinhronizacija kroz `mutex` je mehanizam koji omogućava da kontrolišemo koliko niti mogu biti u nekoj ključnoj sekciji koda
- Ponekad nam ne treba samo to, nego nam je neophodno da neku nit pauziramo dok neki proizvoljan uslov nije zadovoljen.
- Nije bitno koji je to uslov, samo da se nit mora pauzirati dok uslov nije ispunjen.
- Klasa koja se koristi da ovo implementira je `condition_variable`
- Ako hoćemo da čekamo mi proverimo uslov (i to u while petlji) i u telu petlje pozivemo `.wait` nad `condition_variable` instancom.
- `.wait` prima lock pod kojim smo trenutno i oslobađa ga dok čekamo da ne bi blokirali ceo program tako što čekamo pod bravom
- Ostaćemo u čekanju dok neka druga nit, negde drugde, ne pozove komandu da nas otkoči, to može biti `.notify` ili `.notify_all` gde prva otkoči nit koja najduže čeka na uslov, a druga otkoči sve.

## Slobodni baferi - neblokirajuća verzija

```
1  struct List_member {
2      List_member* next;
3      char buffer[512];
4  };
5  class List {
6      mutex mx;
7      List_member* first;
8  public:
9      List() : first(0) {};
10     void link(List_member* member);
11     List_member* unlink();
12 };
13 void List::link(List_member* member){
14     unique_lock<mutex> lock(mx);
15     member->next=first;
16     first=member;
17 }
18 List_member* List::unlink(){
19     List_member* unlinked;
20     {
21         unique_lock<mutex> lock(mx);
22         unlinked=first;
23         if(first != 0)
24             first=first->next;
25     }
26     return unlinked;
27 }
28 }
```

## Slobodni baferi - blokirajuća verzija

```
1  struct List_member {
2      List_member* next;
3      char buffer[512];
4  };
5  class List {
6      mutex mx;
7      List_member* first;
8      condition_variable nonempty;
9  public:
10     List() : first(0) {};
11     void link(List_member* member);
12     List_member* unlink();
13 }
```

```

14
15 void List::link(List_member* member){
16     unique_lock<mutex> lock(mx);
17     member->next=first;
18     first=member;
19     nonempty.notify_one();
20 }
21
22 List_member* List::unlink(){
23     List_member* unlinked;
24     {
25         unique_lock<mutex> lock(mx);
26         while (first == 0)
27             nonempty.wait(lock);
28         unlinked=first;
29         first=first->next;
30     }
31     return unlinked;
32 }
```

## Komunikacioni bafer

```

1 const unsigned int BUFFER_SIZE = 512;
2 enum Buffer_states {EMPTY, FULL};
3 class Buffer {
4     mutex mx
5     char content[BUFFER_SIZE];
6     Buffer_states state;
7     condition_variable full;
8     condition_variable empty;
9 public:
10    Buffer() {state = EMPTY;};
11    void put(char* c);
12    void get(char* c);
13 };
14
15
16 void Buffer::put(char* c){
17     unsigned int i;
18     unique_lock<mutex> lock(mx);
19     while (state == FULL)
20         empty.wait(lock);
21     for(i = 0; i < BUFFER_SIZE; i++)
22         content[i] = *c++;
23     state = FULL;
```

```

24     full.notify_one();
25 }
26
27 void Buffer::get(char* c){
28     unsigned int i;
29     unique_lock<mutex> lock(mx);
30     while (state == EMPTY)
31         full.wait(lock);
32     for(i = 0; i < BUFFER_SIZE; i++)
33         *c++ = content[i];
34     state = EMPTY;
35     empty.notify_one();
36 }
```

## Komunikacioni kanal kapaciteta jedne poruke

- Saradnja niti proizvođača i niti potrošača, u toku koje prva od njih prosledjuje rezultate svoje aktivnosti drugoj niti, može da se prikaže kao razmene poruka.
- U toku ove razmene proizvođač odlaže poruku u poseban pregradak iz koga tu poruku preuzima potrošač.
- Takvu razmenu poruka podržava templejt klasa Message\_box

```

1 template<class MESSAGE>
2 class Message_box {
3     mutex mx;
4     enum Message_box_states { EMPTY, FULL };
5     MESSAGE content;
6     Message_box_states state;
7     condition_variable full;
8     condition_variable empty;
9 public:
10    Message_box() : state(EMPTY) {};
11    void send(const MESSAGE* message);
12    MESSAGE receive();
13 };
14
15 template<class MESSAGE> void Message_box<MESSAGE>::send(const MESSAGE* message){
16     unique_lock<mutex> lock(mx);
17     while(state == FULL)
18         empty.wait(lock);
19     content = *message;
20     state = FULL;
21     full.notify_one();
22 }
23
```

```

24 template<class MESSAGE> MESSAGE Message_box<MESSAGE>::receive(){
25     unique_lock<mutex> lock(mx);
26     while(state == EMPTY)
27         full.wait(lock);
28     state = EMPTY;
29     empty.notify_one();
30     return content;
31 }
32
33
34 - Templejt klasa Message_box omogućuje uspostavljanje komunikacionog kanala između niti pošiljalice i primatelja.
35 - Njene operacije `send()` i `receive()` omogućuju asinhronu razmenu poruka jer se pošiljalica i primatela mogu koristiti u isto vreme.
36 - Ako se kapacitet komunikacionog kanala poveća na dve ili više poruka, tada svakom prijemu će biti pridružena vlastita niti.
37
38 - S druge strane, uz zadržavanje kapaciteta komunikacionog kanala na jednoj poruci, razmena je asinhrona.
39 - Aktivnost ove niti ostaje zaustavljena dok i druga nit ne započne razmenu poruka (dok se preuzima).
40 - Pri tome se podrazumeva da pošiljalac nastavlja svoju aktivnost tek kada primalac preuzeće poruku.
41 - Prethodno dozvoljava da se u komunikacionom kanalu ne čuva poruka, nego njena adresa.
42 - To doprinosi brzini sinhronne razmene poruka, jer primalac može direktno preuzeti poruku od pošiljalice.
43
44 - Time se izbegava potreba da se poruka prepisuje u komunikacioni kanal, što je neizbežno kod asinhronih razmena.
45 - Iako na ovaj način primalac pristupa lokalnoj promenljivoj pošiljaoca, u kojoj se nalazi poruka.
46 - Pošto sinhrona razmena poruka zahteva da se pošiljalac i primalac poruke sretnu, ona se ne može realizovati u realnom vremenu.
47
48 # Pitana

```

## Pitana

- Šta je preplitanje?
- Da li preplitanje ima slučajan karakter?
- Šta izaziva pojavu preplitanja?
- Da li preplitanje može uticati na rezultat izvršavanja programa?
- Šta su deljene promenljive?
- Šta je preduslov očuvanja konzistentnosti deljenih promenljivih?
- Šta su kritične sekcije?
- Šta je sinhronizacija?
- Koje vrste sinhronizacije postoje?
- Šta je atomski region?
- Šta sužava primenu atomskih regiona?
- Čemu služi propusnica?

- Šta se dešava sa niti koja zatraži, a ne dobije propusnicu?
- Šta se dešava kada nit vrati propusnicu?
- Kako se štiti konzistentnost propusnica?
- Šta je isključivi region?
- Šta uvode poželjne osobine konkurentnih programa?
- Po čemu se konkurentno programiranje razlikuje od sekvensijalnog?
- Koje prednosti ima konkurentna biblioteka u odnosu na konkurentni programske jezike?
- Kako se opisuju niti?
- Kako se kreiraju niti?
- Kada se zauzima propusnica deljene promenljive?
- Kada se oslobađa propusnica deljene promenljive?
- Kakvu ulogu ima klasa mutex?
- Kakvu ulogu ima klasa unique\_lock?
- Kakvu ulogu ima klasa condition\_variable?
- Koje vrste razmene poruka postoje?
- 

### **U čemu se razlikuju sinhrona i asinhrona razmena poruka?**

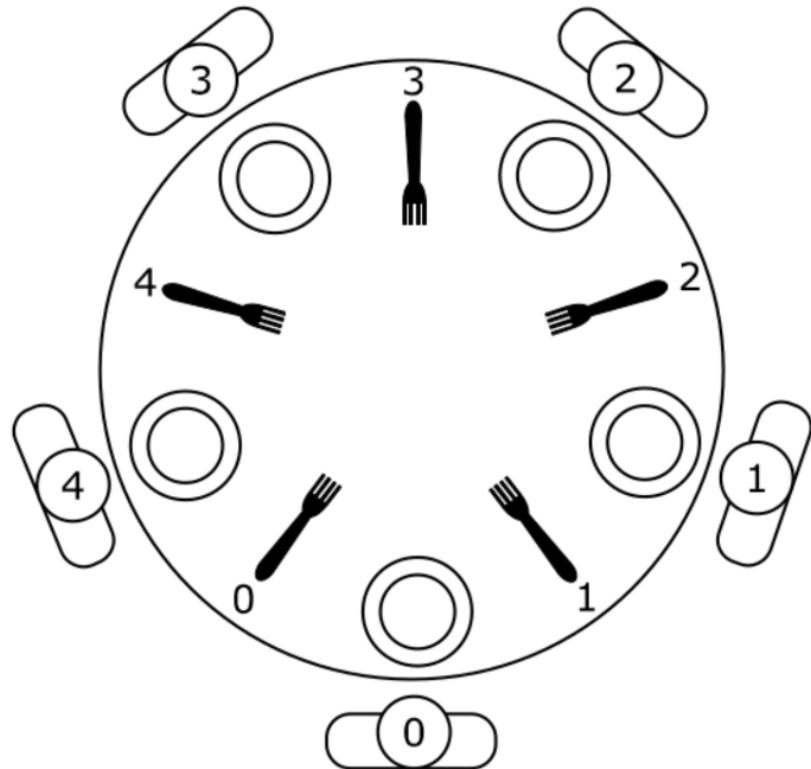
title: Operativni Sistemi - Konkurentno programiranje 2 author: Veljko Petrović date: Mart, 2024 —

## **Problem Pet Filozofa**

### **Postavka**

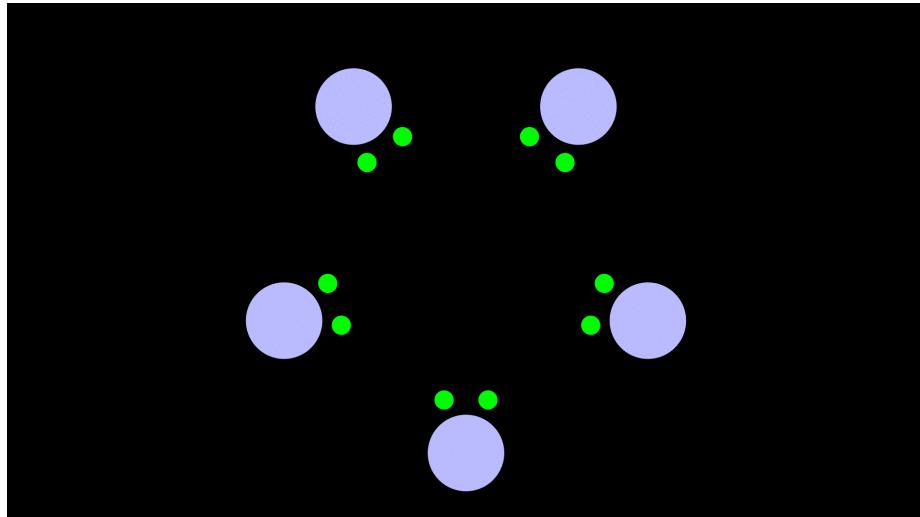
- Zauzimanje više primeraka resursa iste vrste, neophodnih za aktivnost svake niti iz neke grupe niti, predstavlja tipičan problem konkurentnog programiranja.
- On se, u literaturi, ilustruje primerom problema pet filozofa (dining philosophers).
- Svaki od njih provodi život razmišljajući u svojoj sobi i jedući u zajedničkoj trpezariji.

- U njoj se nalazi pet stolica oko okruglog stola sa pet tanjira i pet viljuški između njih.

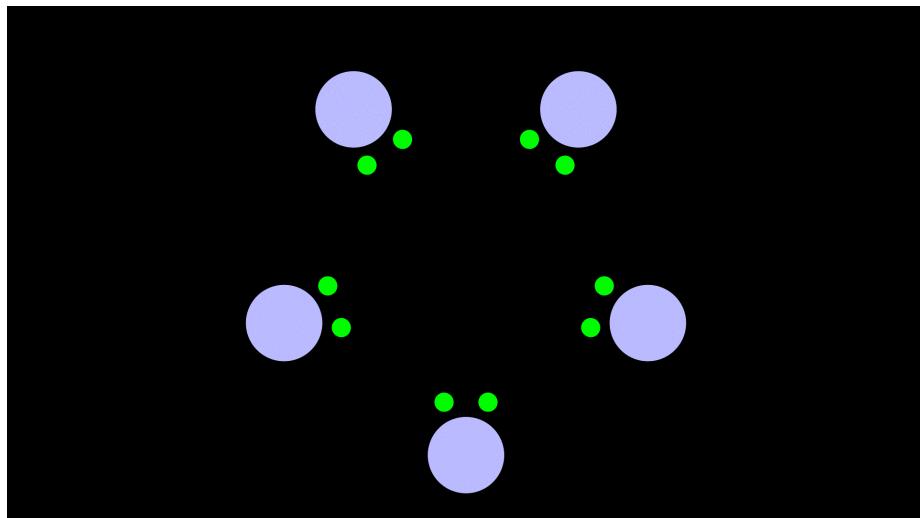


- Pošto se, po želji filozofa, u trpezariji služe uvek špagete, svakom filozofu su za jelo potrebne dve viljuške.
- Ako svi filozofi istovremeno ogladne, uđu u trpezariju, sednu na svoje mesto za stolom i uzmu viljušku levo od sebe, tada nastupa mrtva petlja (deadlock), s kognim ishodom po život filozofa.

## **Normalno ponašanje**



## **Mrtva petlja**



## **Simulator**

- Ponašanje svakog filozofa opisuje funkcija `thread_phosopher()`.
- Razmišljanje filozofa se predstavlja kao odlaganje aktivnosti niti koja reprezentuje filozofa.
- Trajanje ovog odlaganja određuje konstanta `THINKING_PERIOD`.

- Na sličan način se predstavlja jedenje filozofa.
- Trajanje obroka filozofa određuje konstanta EATING\_PERIOD.
- Uzimanje viljuške pre jela i njeno vraćanje posle jela, opisuju operacije `take_fork()` i `release_fork()` klase `Dining_table`.
- Svaki filozof uzima viljuške jednu po jednu samo ako su one slobodne.
- U suprotnom, filozof čeka da svaka viljuška postane raspoloživa.
- Prilikom vraćanja viljušaka filozof oslobađa viljuške jednu po jednu i omogući nastavak aktivnosti svojih suseda.
- Operacije `take_fork()` i `release_fork()` klase `Dining_table`, dopuštaju pojavu mrtve petlje.
- Da bi se ona sigurno desila uvedena su odlaganja aktivnosti niti, koja reprezentuje filozofa, u trajanju koje određuje konstanta MEANTIME.
- Polje `fork_available` deljene klase `Dining_table` omogućuje očekivanje ispunjenja uslova da je viljuška raspoloživa, kao i objavljivanje ispunjenosti ovog uslova.
- Klasa `Dining_table` sadrži i polja `philosopher_state` i `fork_state`.
- Prvo od njih izražava stanja filozofa (THINKING, WAITING\_LEFT\_FORK, HOLDING\_ONE\_FORK, WAITING\_RIGHT\_FORK, EATING), a drugo stanja viljuški (FREE, BUSY).
- Operacija `show()` klase `Dining_table` omogućuje prikazivanje svake promene stanja filozofa.
- Za svakog filozofa se u zagradama navode njegova numerička oznaka i njegovo stanje.
- Funkcija `mod5()` podržava modulo aritmetiku.
- U funkciji `main()` se kreiraju niti filozofi, a zatim se sačeka kraj njihove aktivnosti.
- Svaka od ovih niti preuzme svoj identitet (`Dining_table::take_identity()`): 0, 1, 2, 3 i 4 koji omogućuje razlikovanje filozofa.

## Simulator kod

```

1 #include<thread>
2 #include<iostream>
3
4 using namespace std;
5 using namespace chrono;
```

```

6  using namespace this_thread;
7
8  int mod5(int a)
9  {
10     return (a > 4 ? 0 : a);
11 }
12
13 enum Philosopher_state {THINKING = 'T',
14                         WAITING_LEFT_FORK = 'L',
15                         HOLDING_ONE_FORK = 'O',
16                         WAITING_RIGHT_FORK = 'R',
17                         EATING = 'E'};
18
19 enum Fork_state {FREE, BUSY};
20
21 class Dining_table {
22     mutex mx;
23     int philosopher_identity;
24     Philosopher_state philosopher_state[5];
25     Fork_state fork_state[5];
26     condition_variable fork_available[5];
27     void show();
28 public:
29     Dining_table();
30     int take_identity();
31     void take_fork(int fork, int philosopher, Philosopher_state waiting_state,
32                     Philosopher_state next_state);
33     void release_fork(int fork, int philosopher, Philosopher_state next_state);
34 };
35
36 Dining_table::Dining_table(){
37     philosopher_identity=0;
38     for(int i = 0; i < 5; i++) {
39         philosopher_state[i] = THINKING;
40         fork_state[i] = FREE;
41     }
42 }
43
44 void Dining_table::show(){
45     for(int i = 0; i < 5; i++) {
46         cout << '(' << (char)(i+'0') << ':'
47             << (char)philosopher_state[i] << ") ";
48     }
49     cout << endl;
50 }
51

```

```

52     int Dining_table::take_identity(){
53         unique_lock<mutex> lock(mx);
54         return philosopher_identity++;
55     }
56
57     void Dining_table::take_fork(int fork, int philosopher,
58                                     Philosopher_state waiting_state,
59                                     Philosopher_state next_state){
60         unique_lock<mutex> lock(mx);
61         if(fork_state[fork] == BUSY) {
62             philosopher_state[philosopher] = waiting_state;
63             show();
64             do {fork_available[fork].wait(lock);}while(fork_state[fork] == BUSY);
65         }
66         fork_state[fork] = BUSY;
67         philosopher_state[philosopher] = next_state;
68         show();
69     }
70
71     void Dining_table::release_fork(int fork, int philosopher,
72                                     Philosopher_state next_state){
73         unique_lock<mutex> lock(mx);
74         fork_state[fork] = FREE;
75         philosopher_state[philosopher] = next_state;
76         show();
77         fork_available[fork].notify_one();
78     }
79
80     Dining_table dining_table;
81     const milliseconds THINKING_PERIOD(10);
82     const milliseconds MEANTIME(5);
83     const milliseconds EATING_PERIOD(10);
84     void thread_philosopher(){
85         int philosopher = dining_table.take_identity();
86         int fork = philosopher;
87         for(;;) {
88             sleep_for(THINKING_PERIOD);
89             dining_table.take_fork(fork, philosopher,
90                                   WAITING_LEFT_FORK, HOLDING_ONE_FORK);
91             sleep_for(MEANTIME);
92             dining_table.take_fork(mod5(fork+1), philosopher,
93                                   WAITING_RIGHT_FORK, EATING);
94             sleep_for(EATING_PERIOD);
95             dining_table.release_fork(fork, philosopher, HOLDING_ONE_FORK);
96             sleep_for(MEANTIME);
97             dining_table.release_fork(mod5(fork+1), philosopher, THINKING);

```

```

98     }
99 }
100
101 int main(){
102     cout << endl << "DINING PHILOSOPHERS" << endl;
103     thread philosopher0(thread_philosopher);
104     thread philosopher1(thread_philosopher);
105     thread philosopher2(thread_philosopher);
106     thread philosopher3(thread_philosopher);
107     thread philosopher4(thread_philosopher);
108     philosopher0.join();
109     philosopher1.join();
110     philosopher2.join();
111     philosopher3.join();
112     philosopher4.join();
113 }
```

## Rezultat simulacije

( 0: T )	( 1: T )	( 2: 0 )	( 3: T )	( 4: T )
( 0: 0 )	( 1: T )	( 2: 0 )	( 3: T )	( 4: T )
( 0: 0 )	( 1: T )	( 2: 0 )	( 3: 0 )	( 4: T )
( 0: 0 )	( 1: 0 )	( 2: 0 )	( 3: 0 )	( 4: T )
( 0: 0 )	( 1: 0 )	( 2: 0 )	( 3: 0 )	( 4: 0 )
( 0: 0 )	( 1: 0 )	( 2: R )	( 3: 0 )	( 4: 0 )
( 0: R )	( 1: 0 )	( 2: R )	( 3: 0 )	( 4: 0 )
( 0: R )	( 1: 0 )	( 2: R )	( 3: R )	( 4: 0 )
( 0: R )	( 1: R )	( 2: R )	( 3: R )	( 4: 0 )
( 0: R )	( 1: R )	( 2: R )	( 3: R )	( 4: R )

## Rešenje?

- Postoje mnoga rešenja ovog problema

- Najlakši način jeste da se uvede *konobar* odnosno centralni sinhronizacioni entitet koji služi da se garantuje da se viljuške uzimaju odjednom, odn. ili obe, ili nijedna.
- Alternative jeste da se razbije cikličnost tako što se primeti da imamo simetriju: uvek uzimamo levu pa desnu. Šta ako makar jedan filozof uvek prvo uzme prvo desnu pa levu ili parni filozofi uzimeju uvek levu pa desnu?

## Rešenje

- Jos jedan problem koji rešava problem stohastički za relativno retke petlje jeste onaj koji se koristi u određenim verzijama Etherneta
- Umesto da čekamo zauvek (`.wait`) koristimo čekanje sa `wait_for` ili `wait_until` što omogućava da se čeka do ispunjenja uslova *ili* dok ne istakne neko vreme.
- Naravno ako čekamo svi isto vreme možemo da upadnemo u petlju gde se svi 'sudarimo' i onda čekamo neko (isto) vreme, pa se opet 'sudarimo' i tako u beskonačnost
- To se zove i 'živa' petlja odn. 'livelock'
- Umesto da čekamo uvek isto vreme, mi oslobodimo ono što smo zauzeli, čekamo nasumično vreme, i probamo opet
- Ovo razbije simetriju i omogući da se sistem eventualno odglavi

## Mrtva petlja

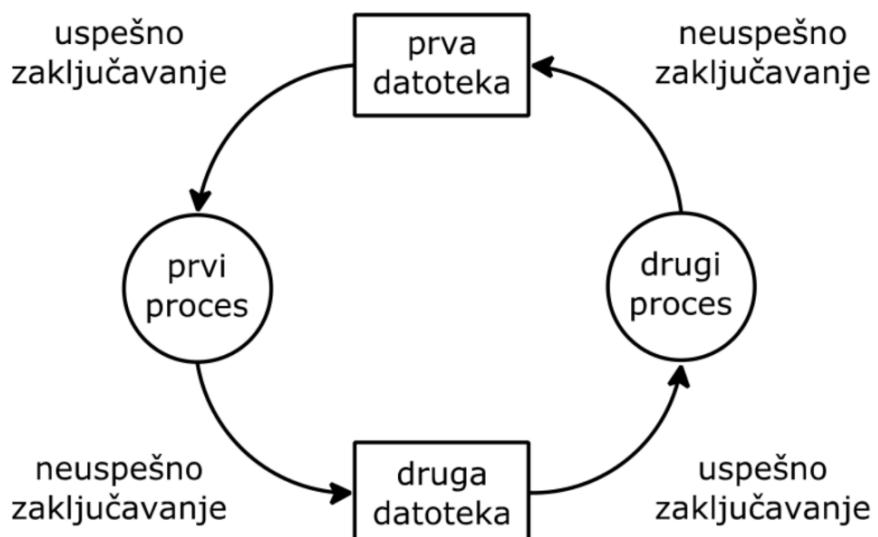
### Pet filozofa i mrtve petlje

- Pet filozofa je namerno upečatljiv primer mrtve petlje, ali ona može da nastane na raznim mestima
- Svaki višenitni program je ranjiv
- Takođe svaki operativni sistem koji zaključava fajlove za pristup
- Ako se steknu uslovi, onda gde god da imamo zaključavanje i višestruko izvršavanje, mrtva petlja je moguća

### Uslovi za pojavu mrtve petlje

- Mrtva petlja je problematična pojava trajnog zaustavljanja aktivnosti međusobno zavisnih procesa.
- Na primer, ona se javi kada dva procesa žele da u režimu međusobne isključivosti pristupaju dvema datotekama.

- Ako prvi od njih zaključa prvu datoteku, a zatim drugi od njih zaključa drugu datoteku, tada nema mogućnosti za nastavak aktivnosti tih procesa, bez obzira da li je sistemska operacija zaključavanja blokirajuća ili ne.
- U slučaju blokirajućih sistemskih operacija zaključavanja, pokušaj prvog procesa da zaključa drugu datoteku dovodi do trajnog zaustavljanja njegove aktivnosti.
- Isto se dešava sa drugim procesom prilikom njegovog pokušaja da zaključa prvu datoteku.
- Zaustavljanje aktivnosti ova dva procesa je trajno, jer je svaki od njih zauzeo datoteku koja treba onom drugom procesu za nastavak njegove aktivnosti i nema nameru da tu datoteku osloboodi.



- U slučaju neblokirajuće sistema operacije zaključavanja, procesi upadaju u beskonačnu petlju (starvation), pokušavajući da zaključaju datoteku, koju je zaključao drugi proces.
- Ovakav oblik međusobne zavisnosti procesa se naziva i živa petlja (livelock).
- Ona se, po svom ishodu, suštinski ne razlikuje od mrtve petlje.
- Pojava mrtve petlje je vezana za zauzimanje resursa, kao što su, na primer, prethodno pomenute datoteke.
- Pri tome, za pojavu mrtve petlje je potrebno da budu ispunjena

četiri uslova.

1. zauzimani resursi se koriste u režimu međusobne isključivosti
2. resursi se zauzimaju jedan za drugim, tako da proces, nakon zauzimanja izvesnog broja resursa, mora da čeka da zauzme preostale resurse
3. resurse oslobođaju samo procesi koji su ih zauzeli
4. postoji cirkularna međuzavisnost procesa (prvi proces čeka oslobođanje resursa koga drži drugi proces, a on čeka oslobođanje resursa koga drži treći proces, i tako redom do poslednjeg procesa iz lanca procesa, koji čeka oslobođanje resursa koga drži prvi proces).

## Tretiranje mrtve petlje

- Postoje četiri pristupa tretiranja problema mrtve petlje:
- sprečavanje (prevention) pojave mrtve petlje (onemogućavanjem važenja nekog od četiri neophodna uslova za njenu pojavu)
- izbegavanje (avoidance) pojave mrtve petlje
- otkrivanje (detection) pojave mrtve petlje i oporavak (recovery) od nje
- ignorisanje pojave mrtve petlje.
- Kod sprečavanja pojave mrtve petlje, važenje prvog uslova obično nije moguće sprečiti, jer se resursi najčešće koriste u režimu međusobne isključivosti.
- Važenje drugog uslova se može sprečiti, ako se unapred zna koliko treba resursa i ako se oni svi zauzmu pre korišćenja.
- Pri tome, neuspeh u zauzimanju bilo kog resursa dovodi do oslobođanja prethodno zauzetih resursa, što je moguće učiniti bez posledica, jer nije započelo njihovo korišćenje.
- Važenje trećeg uslova se obično ne može sprečiti, jer najčešće ne postoji način da se zauzeti resurs privremeno oduzme procesu.
- I konačno, važenje četvrtog uslova se može sprečiti, ako se resursi uvek zauzimaju u unapred određenom redosledu, koji isključuje mogućnost cirkularne međuzavisnosti procesa (primer pozitivnog rešenja 5 filozofa).
- Izbegavanje pojave mrtve petlje zahteva poznavanje podataka:
  - o maksimalno mogućim zahtevima za resursima
  - ukupno postavljenim zahtevima za resursima

- stanju resursa

- Podrazumeva se da se udovoljava samo onim zahtevima za koje se proverom ustanovi da, nakon njihovog ispunjavanja, postoji redosled zauzimanja i oslobođanja resursa u kome se mogu zadovoljiti maksimalno mogući zahtevi svih procesa.
- Praktična vrednost ovoga pristupa nije velika, jer se obično unapred ne znaju maksimalno mogući zathevi procesa za resursima, a to je neophodno za proveru da li se može udovoljiti pojedinim zahtevima.
- Sem toga, ovakva provera je komplikovana, a to znači i neefikasna.
- Otkrivanje pojave mrtve petlje se zasniva na sličnom pristupu kao i izbegavanje pojave mrtve petlje.
- U ovom slučaju se proverava da li postoji proces, čijim zahtevima se ne može udovoljiti ni za jedan redosled zauzimanja i oslobođanja resursa.
- Pored komplikovanosti ovakve provere, problem je i šta učiniti, kada se i otkrije pojava mrtve petlje.
- Ako se resursi ne mogu privremeno oduzeti od procesa, preostaje jedino uništavanje procesa, radi oslobođanja resursa koje oni drže.
- Međutim, to nije uvek prihvatljiv zahvat.
- Zbog toga ni ovaj pristup nema veliki praktični značaj.
- Ignorisanje pojave mrtve petlje je pristup koji je najčešće primjenjen u praksi, jer se ovaj problem ne javlja tako često da bi se isplatilo da ga rešava operativni sistem.
- Prema tome, kada se mrtva petlja javi, na korisniku je da se suoči sa ovim problemom i da ga reši na način, koji je primeren datim okolnostima.

## **Problem čitanja i pisanja**

### **Postavka**

- Problem čitanja i pisanja (readers-writers problem) se može objasniti na primeru kao što je rukovanje bankovnim računima.
- Bankovni računi pripadaju komitentima banke i sadrže ukupan iznos novčanih sredstava svakog od komitenata.

- U najjednostavnijem slučaju, rukovanje bankovnim računima se svodi: na prenos sredstava (s jednog računa na drugi) i na proveru (stanja svih) računa.
- Prenos sredstava obuhvata četiri koraka:
  - Čitanje stanja računa s koga se prenose sredstva
  - Pisanje novog stanja na ovaj račun. Novo stanje se dobije umanjivanjem pročitanog stanja za prenošeni iznos
  - Čitanje stanja računa na koji se prenose sredstva.
  - Pisanje novog stanja na račun na koji se prenose sredstva. Novo stanje se dobije uvećavanjem pročitanog stanja za prenošeni iznos
- Uz pretpostavku da su prenosi sredstava mogući samo između posmatranih bankovnih računa, ukupna suma njihovih stanja je nepromenljiva.
- Prema tome, provera računa se svodi na čitanja, jedno za drugim, stanja svih računa, radi njihovog sumiranja.
- Ispravnost prenosa sredstava zavisi od očuvanja konzistentnosti stanja svih računa, za šta je neophodna međusobna isključivost raznih prenosa sredstava. U suprotnom, moguće su razne greške.
- Iz prethodne analize sledi:
  - Da je za ispravnost prenosa bitno da prenosi budu međusobno isključivi.
  - Da je za ispravnost provera bitno da provere i prenosi budu međusobno isključivi.
- Pošto prenosi sadrže pisanja, a provere samo čitanja, sledi da operacije sa pisanjem moraju biti međusobno isključive, kao što moraju biti međusobno isključive operacije sa pisanjem i operacije sa čitanjem.
- Za operacije koje sadrže samo čitanja međusobna isključivost nije potrebna.

## Kod

```

1 #include<thread>
2 #include<iostream>
3 using namespace std;
4 using namespace chrono;
5 using namespace this_thread;
6 const unsigned ACCOUNTS_NUMBER = 10;
7 const int INITIAL_AMOUNT = 100;

```

```

8   class Bank {
9     mutex mx;
10    int accounts[ACCOUNTS_NUMBER];
11    short readers_number;
12    short writers_number;
13    short readers_delayed_number;
14    short writers_delayed_number;
15    condition_variable readers_q;
16    condition_variable writers_q;
17
18    void show();
19    void reader_begin();
20    void reader_end();
21    void writer_begin();
22    void writer_end();
23 public:
24    Bank();
25    void audit();
26    void transaction(unsigned source, unsigned destination);
27 };
28
29 Bank::Bank(){
30     for(unsigned i = 0; i < ACCOUNTS_NUMBER; i++)
31         accounts[i] = INITIAL_AMOUNT;
32     readers_number = 0;
33     writers_number = 0;
34     readers_delayed_number = 0;
35     writers_delayed_number = 0;
36 }
37
38 void Bank::show(){
39     cout << "RN: " << readers_number << " RDN: "
40             << readers_delayed_number << " WN: "
41             << writers_number << " WDN: "
42             << writers_delayed_number << endl;
43 }
44
45 void Bank::reader_begin(){
46     unique_lock<mutex> lock(mx);
47     if((writers_number > 0) || (writers_delayed_number > 0)){
48         readers_delayed_number++;
49         show();
50         do { readers_q.wait(lock); }
51         while((writers_number > 0) ||
52                (writers_delayed_number > 0));
53     }

```

```

54     readers_number++;
55     show();
56     if(readers_delayed_number > 0){
57         readers_delayed_number--;
58         show();
59         readers_q.notify_one();
60     }
61 }
62 }
63
64 void Bank::reader_end(){
65     unique_lock<mutex> lock(mx);
66     readers_number--;
67     show();
68     if((readers_number == 0) && (writers_delayed_number > 0)){
69         writers_delayed_number--;
70         show();
71         writers_q.notify_one();
72     }
73 }
74
75 void Bank::writer_begin(){
76     unique_lock<mutex> lock(mx);
77     if((readers_number > 0) || (writers_number > 0)){
78         writers_delayed_number++;
79         show();
80         do { writers_q.wait(lock); }
81         while((readers_number > 0) || (writers_number > 0));
82     }
83     writers_number++;
84     show();
85 }
86
87 void Bank::writer_end(){
88     unique_lock<mutex> lock(mx);
89     writers_number--;
90     show();
91     if(writers_delayed_number > 0){
92         writers_delayed_number--;
93         show();
94         writers_q.notify_one();
95     } else if(readers_delayed_number > 0) {
96         readers_delayed_number--;
97         show();
98         readers_q.notify_one();
99     }

```

```

100 }
101
102 const milliseconds WRITING_PERIOD(1);
103 void Bank::transaction(unsigned source,unsigned destination){
104     int amount;
105     writer_begin();
106     sleep_for(WRITING_PERIOD);
107     amount = accounts[source];
108     accounts[source] -= amount;
109     accounts[destination] += amount;
110     writer_end();
111 }
112 Bank bank;
113 void thread_reader(){
114     bank.audit();
115 }
116
117 const milliseconds READING_PERIOD(1);
118 void Bank::audit(){
119     int sum = 0;
120     reader_begin();
121     sleep_for(READING_PERIOD);
122     for(unsigned i = 0; i < ACCOUNTS_NUMBER; i++)
123         sum += accounts[i];
124     reader_end();
125     if(sum != ACCOUNTS_NUMBER*INITIAL_AMOUNT) {
126         unique_lock<mutex> lock(mx);
127         cout << " audit error " << endl;
128     }
129 }
130
131 const milliseconds WRITING_PERIOD(1);
132 void Bank::transaction(unsigned source,unsigned destination){
133     int amount;
134     writer_begin();
135     sleep_for(WRITING_PERIOD);
136     amount = accounts[source];
137     accounts[source] -= amount;
138     accounts[destination] += amount;
139     writer_end();
140 }
141 Bank bank;
142 void thread_reader(){
143     bank.audit();
144 }
145

```

```

146 void thread_writer0to1(){
147     bank.transaction(0, 1);
148 }
149 void thread_writer1to0(){
150     bank.transaction(1, 0);
151 }
152 int main(){
153     cout << endl << "READERS AND WRITERS" << endl;
154     thread reader0(thread_reader);
155     thread reader1(thread_reader);
156     thread writer0(thread_writer0to1);
157     thread reader2(thread_reader);
158     thread writer1(thread_writer1to0);
159     reader0.join();
160     reader1.join();
161     writer0.join();
162     reader2.join();
163     writer1.join();
164 }
```

## Komentar o rešenju

- Ovo rešenje, eksplisitno, je napravljeno tako da pisanje ima prednost u odnosu na čitanje
- Ako je plan da bude obrnuto onda su potrebne promene

# Rizici konkurentnog programiranja

## Rizici

- Opisivanje obrada podataka je jedini cilj sekvencijalnog, a osnovni cilj konkurentnog programiranja.
- Bolje iskorišćenje računara i njegovo čvršće sprezanje sa okolinom su dodatni ciljevi konkurentnog programiranja, po kojima se ono i razlikuje od sekvencijalnog programiranja.
- Od suštinske važnosti je da ostvarenje dodatnih ciljeva ne ugrozi ostvarenje osnovnog cilja, jer je on neprikosnoven, pošto je konkurentni program upotrebljiv jedino ako iza svakog od njegovih izvršavanja ostaju samo ispravno obrađeni podaci.
- Tipične poželjne osobine sekvencijalnog, a to znači i konkurentnog programa obuhvataju tvrdnje uključivanja poželjnog, kao što je tvrdnja da nakon izvršavanja programa ostaju ispravno

obrađeni podaci, i tvrdnje isključivanja nepoželjnog, kao što je tvrdnja da program ne sadrži beskonačne petlje.

- Tipične dodatne poželjne osobine konkurentnog programa obuhvataju tvrdnje uključivanja poželjnog, kao što je tvrdnja da su, u toku izvršavanja programa, deljene promenljive stalno konzistentne, i tvrdnje isključivanja nepoželjnog, kao što je tvrdnja da u toku izvršavanja programa ne dolazi do trajnog zaustavljanja aktivnosti niti.
- Ispravnu obradu podataka ugrožava narušavanje konzistentnosti deljenih promenljivih u toku izvršavanja konkurentnog programa.
- Do narušavanja konzistentnosti deljenih promenljivih dolazi, ako na kraju isključivog regiona deljena promenljiva nije u konzistentnom stanju ili ako se operacija `wait()` pozove pre nego je deljena promenljiva dovedena u konzistentno stanje

## Pozivanje `wait` pre dovodenja u konzistentno stanje

```
1  {
2      unique_lock<mutex> lock(mx);
3      //< exclusive region 1 >
4      some_condition.wait(lock);
5      //< exclusive region 2 >
6 }
```

## Mrtva petlja među nitima

- Upotrebljivost konkurentnih programa ugrožava i pojava međuzavisnosti niti, poznata pod nazivom mrtva petlja.
- Ona dovodi do trajnog zaustavljanja aktivnosti niti, a to ima za posledicu da izvršavanje konkurentnog programa nema kraja.
- Konkurentni program, u toku čijeg izvršavanja je moguća pojava mrtve petlje, nije upotrebljiv, jer pojedina od njegovih izvršavanja, koja nemaju kraja, ne dovode do uspešne obrade podataka.
- Do mrtve petlje može da dođe, na primer, ako se iz jedne deljene klase pozivaju operacije druge deljene klase, pod uslovom da je bar jedna od pozivanih operacija blokirajuća.

## Minimalni primer mrtve petlje

```
1 class Activity {
2     mutex mx_activity;
```

```

3     condition_variable activity_permission;
4 public:
5     void stop();
6     void start();
7 };
8 void Activity::stop(){
9     unique_lock<mutex> lock(mx_activity);
10    activity_permission.wait(lock);
11 }
12 void Activity::start(){
13     unique_lock<mutex> lock(mx_activity);
14     activity_permission.notify_one();
15 }
16
17 class Manager {
18     mutex mx_manager;
19     Activity activity;
20 public:
21     void disable_activity();
22     void enable_activity();
23 };
24 void Manager::disable_activity(){
25     unique_lock<mutex> lock(mx_manager);
26     activity.stop();
27 }
28 void Manager::enable_activity(){
29     unique_lock<mutex> lock(mx_manager);
30     activity.start();
31 }
32 Manager manager;

```

## Minimalna mrtva petlja

- Mrtve petlje, koje ilustruje prethodni primer, se mogu sprečiti, ako se blokirajuća operacija ne poziva iz isključivog regiona.
- Nenamerno izazivanje konačnog, ali nepredvidivo dugog zaustavljanja aktivnosti niti u toku isključivog regiona može da ima negativne posledice na izvršavanje programa.
- To se, na primer, desi, kada se iz isključivog regiona pozivaju potencijalno blokirajuće operacije, poput funkcije `sleep_for()`.
- Globalne const promenljive, koje služe za smeštanje podataka, raspoloživih svim nitima, ne spadaju u deljene promenljive.

## **Pitanja**

### **Pitanja**

- Opisati problem pet filozofa.
- Kako bi izgledala verzija problema pet filozofa koja bi se realistično mogla sresti tokom razvoja softvera?
- Šta je mrtva petlja?
- Po čemu se živa petlja razlikuje od mrtve petlje?
- Koji uslovi su potrebni za pojavu mrtve petlje?
- Kako se u praksi tretira problem mrtve petlje?
- Na čemu se temelji sprečavanje mrtve petlje?
- Šta karakteriše izbegavanje mrtve petlje?
- Šta karakteriše otkrivanje i oporavak od mrtve petlje?
- Šta karakteriše ignorisanje mrtve petlje?
- 

### **Opisati problem čitanja i pisanja.**

title: Operativni Sistemi - Konkurentno programiranje 3 author: Veljko Petrović date: Mart, 2024 —

## **Semafori**

### **Sinhronizacija pomoću semafora**

- Sinhronizacija niti može da se zasnove i na ideji saobraćajnog semafora koji reguliše ulazak vozova u stanicu sa jednim kolosekom.
- Kada se jedan voz nalazi na staničnom koloseku, pred semaforom se moraju zaustaviti svi vozovi koji treba da dođu na stanični kolosek.
- Po analogiji sa saobraćajnim semaforom prolaz niti kroz (softversku) kritičnu sekciju bi regulisao (softverski) semafor.
- Sinhronizacija niti, koju omogućuje semafor, se zasniva na zauzimanju aktivnosti niti, kao i na omogućavanju nastavljanja njihove aktivnosti.
- Ulazak niti u kritičnu sekciju zavisi od stanja semafora.

- Kada stanje semafora dozvoli ulazak niti u kritičnu sekciju, pri ulasku se semafor prevodi u stanje koje onemogućuje ulazak druge niti u kritičnu sekciju.
- Ako se takva nit pojavi, njena aktivnost se zaustavlja pred kritičnom sekcijom.
- Pri izlasku niti iz kritične sekcije semafor se prevodi u stanje koje dozvoljava novi ulazak u kritičnu sekciju i ujedno omogućuje nastavak aktivnosti niti koja najduže čeka na ulaz u kritičnu sekciju (ako takva nit postoji).

## Semafor simuliran sa propusnicom

```

1  class Semaphore {
2      mutex mx;
3      int state;
4      condition_variable queue;
5  public:
6      Semaphore(int value = 1) : state(value) {}
7      void stop();
8      void resume();
9  };
10
11
12 void Semaphore::stop(){
13     unique_lock<mutex> lock(mx);
14     while(state < 1)
15         queue.wait(lock);
16     state--;
17 }
18 void Semaphore::resume(){
19     unique_lock<mutex> lock(mx);
20     state++;
21     queue.notify_one();
22 }
```

## Semafor u C++ biblioteci

- C++ biblioteka, od standarda iz 2020 podržava semafore
- Nalaze se u zaglavljima `<semaphore>`
- Klasa je `counting_semaphore` i šablonska je gde je (jedini) parametar maksimalna numerička vrednost ugrađenog brojača
- stop i resume opcije se zovu acquire i release ali je centralna ideja ista

## Vrste i upotreba semafora

- Semafor čije stanje ne može preći vrednost 1 se zove binarni semafor.
- Ako se njegovo stanje inicijalizuje na vrednost 1, tada su njegove operacije `stop()` i `resume()` slične operacijama `lock()` i `unlock()` klase `mutex`.
- Ako se njegovo stanje inicijalizuje na vrednost 0, tada su njegove operacije `stop()` i `resume()` slične operacijama `wait()` i `notify_one()` klase `condition_variable`
- Operacije klase `condition_variable` su namenjene za ostvarenje uslovne sinhronizacije u okviru kritičnih sekcija u kojima je međusobna isključivost ostvarena pomoću operacija klase `mutex`.

## Raspodeljeni binarni semafor

- Međutim, upotreba operacija binarnog semafora (sa stanjem inicijalizovanim na vrednost 0) po uzoru na operacije klase `condition_variable` u okviru kritičnih sekcija u kojima je međusobna isključivost ostvarena pomoću operacija drugog binarnog semafora (sa stanjem inicijalizovanim na vrednost 1) izaziva mrtvu petlju.
- Zato se uvodi posebna vrsta binarnog semafora, nazvana raspodeljeni binarni semafor (split binary semaphore).
- Realizuje se uz pomoć više binarnih semafora za koje važi ograničenje da suma njihovih stanja ne može preći vrednost 1.
- Pomoću raspodeljenog binarnog semafora se ostvaruje uslovna sinhronizacija tako što se na ulazu u svaku kritičnu sekciju poziva operacija `stop()` jednog od njegovih binarnih semafora, a na izlazu iz nje operacija `resume()` tog ili nekog od preostalih binarnih semafora.
- Na taj način najviše jedna nit se može nalaziti najviše u jednoj od pomenutih kritičnih sekcija, jer su stanja svih semafora manja od vrednosti 1 za vreme njenog boravka u dotičnoj kritičnoj sekciji.

## Mesage\_box sa semaforima

```
1 #include "sem.hh"
2
3 template<class MESSAGE>
4 class Message_box {
```

```

5     MESSAGE content;
6     Semaphore empty;
7     Semaphore full;
8 public:
9     Message_box() : full(0) {};
10    void send(const MESSAGE* message);
11    MESSAGE receive();
12 };
13
14
15 template<class MESSAGE>
16 void Message_box<MESSAGE>::send(const MESSAGE* message){
17     empty.stop();
18     content = *message;
19     full.resume();
20 }
21
22
23 template<class MESSAGE>
24 MESSAGE Message_box<MESSAGE>::receive(){
25     MESSAGE message;
26     full.stop();
27     message = content;
28     empty.resume();
29     return message;
30 }
```

## Generalni semafor - primer sa slobodnim baferima

- Semafor, čije stanje može sadržati vrednost veću od 1, se naziva generalni semafor (general semaphore).
- On omogućuje ostvarenje uslovne sinhronizacije prilikom rukovanja resursima.
- Pozitivno stanje generalnog semafora može predstavljati broj slobodnih primeraka nekog resursa.
- Zahvaljujući tome, zauzimanje primerka resursa se može opisati pomoću operacije stop(), a njegovo oslobođanje pomoću operacije resume() generalnog semafora.
- Ova klasa sadrži binarni semafor mex i generalni semafor list\_member\_count.
- Binarni semafor omogućuje međusobnu isključivost prilikom uvezivanja i izvezivanja slobodnog bafera.

- Generalni semafor omogućuje uslovnu sinhronizaciju, jer njegovo stanje pokazuje broj slobodnih bafera (ono je na početku inicijalizovano na 0).

## Generalni semafor i slobodni baferi

```

1 #include "sem.hh"
2
3 struct List_member {
4     List_member* next;
5     char buffer[512];
6 };
7
8 class List {
9     List_member* first;
10    Semaphore list_member_count;
11    Semaphore mex;
12 public:
13     List () : first(0), list_member_count(0) {};
14     void link(List_member* member);
15     List_member* unlink();
16 };
17
18 void List::link(List_member* member){
19     mex.stop();
20     member->next=first;
21     first=member;
22     mex.resume();
23     list_member_count.resume();
24 }
25
26 List_member* List::unlink(){
27     List_member* unlinked;
28     list_member_count.stop();
29     mex.stop();
30     unlinked=first;
31     first=first->next;
32     mex.resume();
33     return unlinked;
34 }
```

## Rešenje problema pet filozofa pomoću semafora

- Rešenje problema pet filozofa pomoću semafora sprečava pojavu mrtve petlje, jer za parne filozofe zauzima prvo levu, a za

- neparne filozofe zauzima prvo desnu viljušku.
- Viljuške predstavljaju binarni semafori forks[5], koji omogućuju uslovnu sinhronizaciju prilikom zauzimanja viljuški.
  - Međusobnu isključivost omogućuje binarni semafor mex.
  - Stanja filozofa su promenjena, jer nije moguća mrtva petlja, pa nije bitno koju viljušku filozof čeka.

## Pet filozofa i semafori

```

1 #include<thread>
2 #include<iostream>
3
4 using namespace std;
5 using namespace chrono;
6 using namespace this_thread;
7
8 #include "sem.hh"
9
10 int mod5(int a){
11     return (a > 4 ? 0 : a);
12 }
13
14 enum Philosopher_state { THINKING = 'T', HUNGRY = 'H', EATING = 'E' };
15 Philosopher_state philosopher_state[5];
16 Semaphore forks[5];
17 Semaphore mex;
18
19 int philosopher_identity(0);
20 const milliseconds THINKING_PERIOD(10);
21 const milliseconds EATING_PERIOD(10);
22
23 void show(){
24     for(int j = 0; j < 5; j++) {
25         cout << '(' << (char)(j+'0') << ':'
26             << (char)(philosopher_state[j]) << ")" <<
27     }
28     cout << endl;
29 }
30
31 void thread_philosopher(){
32     mex.stop();
33     int pi = philosopher_identity++;
34     philosopher_state[pi] = THINKING;
35     mex.resume();
36     for(;;) {

```

```

37     sleep_for(THINKING_PERIOD);
38     mex.stop();
39     philosopher_state[pi] = HUNGRY;
40     show();
41     mex.resume();

42     forks[pi%2 == 0 ? pi : mod5(pi+1)].stop();
43     forks[pi%2 == 0 ? mod5(pi+1) : pi].stop();
44     mex.stop();
45     philosopher_state[pi] = EATING;
46     show();
47     mex.resume();
48     sleep_for(EATING_PERIOD);
49     mex.stop();
50     philosopher_state[pi] = THINKING;
51     show();
52     mex.resume();
53     forks[pi].resume();
54     forks[mod5(pi+1)].resume();
55 }
56 }
57 }

58 int main(){
59     cout << endl << "DINING PHILOSOPHERS" << endl;
60     thread philosopher0(thread_philosopher);
61     thread philosopher1(thread_philosopher);
62     thread philosopher2(thread_philosopher);
63     thread philosopher3(thread_philosopher);
64     thread philosopher4(thread_philosopher);
65     philosopher0.join();
66     philosopher1.join();
67     philosopher2.join();
68     philosopher3.join();
69     philosopher4.join();
70 }
71 }
```

## Čitanje i pisanje i semafori

```

1 #include<thread>
2 #include<iostream>
3
4 using namespace std;
5 using namespace chrono;
6 using namespace this_thread;
7
```

```

8  #include "sem.hh"
9
10 const int ACCOUNTS_NUMBER = 10;
11 const int INITIAL_AMOUNT = 100;
12
13 class Bank {
14     Semaphore mex;
15     int accounts[ACCOUNTS_NUMBER];
16     short readers_number;
17     short writers_number;
18     short readers_delayed_number;
19     short writers_delayed_number;
20     Semaphore readers;
21     Semaphore writers;
22     void show();
23     void reader_begin();
24     void reader_end();
25     void writer_begin();
26     void writer_end();
27
28 public:
29     Bank();
30     void audit();
31     void transaction(unsigned source, unsigned destination);
32 };
33
34 Bank::Bank() : mex(1), readers(0), writers(0){
35     for(int i = 0; i < ACCOUNTS_NUMBER; i++)
36         accounts[i] = INITIAL_AMOUNT;
37     readers_number = 0;
38     writers_number = 0;
39     readers_delayed_number = 0;
40     writers_delayed_number = 0;
41 }
42
43 void Bank::show(){
44     cout << "RN: " << readers_number << " RDN: "
45             << readers_delayed_number
46             << " WN: " << writers_number << " WDN: "
47             << writers_delayed_number << endl;
48 }
49
50 void Bank::reader_begin(){
51     mex.stop();
52     if((writers_number > 0) || (writers_delayed_number > 0)) {
53         readers_delayed_number++;

```

```

54         show();
55         mex.resume();
56         readers.stop();
57     }
58
59     readers_number++;
60     show();
61     if(readers_delayed_number > 0) {
62         readers_delayed_number--;
63         show();
64         readers.resume();
65     } else
66         mex.resume();
67 }
68
69 void Bank::reader_end(){
70     mex.stop();
71     readers_number--;
72     show();
73     if((readers_number == 0) && (writers_delayed_number > 0)) {
74         writers_delayed_number--;
75         show();
76         writers.resume();
77     } else
78         mex.resume();
79 }
80
81 void Bank::writer_begin(){
82     mex.stop();
83     if((readers_number > 0) || (writers_number > 0)) {
84         writers_delayed_number++;
85         show();
86         mex.resume();
87         writers.stop();
88     }
89     writers_number++;
90     show();
91     mex.resume();
92 }
93
94 void Bank::writer_end(){
95     mex.stop();
96     writers_number--;
97     show();
98     if(writers_delayed_number > 0) {
99         writers_delayed_number--;

```

```

100         show();
101         writers.resume();
102     }
103
104     const milliseconds READING_PERIOD(1);
105
106     void Bank::audit(){
107         int sum = 0;
108         reader_begin();
109         sleep_for(READING_PERIOD);
110         for(unsigned i = 0; i < ACCOUNTS_NUMBER; i++)
111             sum += accounts[i];
112         reader_end();
113         if(sum != ACCOUNTS_NUMBER*INITIAL_AMOUNT) {
114             mex.stop();
115             cout << " audit error " << endl;
116             mex.resume();
117         }
118     }
119
120
121     const milliseconds WRITING_PERIOD(1);
122
123     void Bank::transaction(unsigned source, unsigned destination){
124         int amount;
125         writer_begin();
126         sleep_for(WRITING_PERIOD);
127         amount = accounts[source];
128         accounts[source] -= amount;
129         accounts[destination] += amount;
130         writer_end();
131     }
132
133     Bank bank;
134
135     void thread_reader(){
136         bank.audit();
137     }
138     void thread_writer0to1(){
139         bank.transaction(0, 1);
140     }
141     void thread_writer1to0(){
142         bank.transaction(1, 0);
143     }
144
145     int main(){

```

```

146     cout << endl << "READERS AND WRITERS" << endl;
147     thread reader0(thread_reader);
148     thread reader1(thread_reader);
149     thread writer0(thread_writer0to1);
150     thread reader2(thread_reader);
151     thread writer1(thread_writer1to0);
152     reader0.join();
153     reader1.join();
154     writer0.join();
155     reader2.join();
156     writer1.join();
157 }
158 # Konkurentno programiranje bez zaključavanja

```

## Atomici i programiranje bez zaključavanja

- Posle svog ovog truda da napravimo i propusnicu i semafor i sve ovo, zašto za ime sveta bi hteli da sada *odustanemo* od svega toga?
- Kao i obično kada je u pitanju c++ egzotika za ovim posežemo kada nam treba još performansi: svo zaključavanje (propusnice, semafori, sinhronizacija, inače) su *spore*.
- Komparativno govoreći, za većinu tema ovo je više nego zadovoljavajuće
- Šta je alternativa?

## Hardver i portabilnost

- Setite se kada smo pričali o compare-and-swap i hardverskoj podršci?
- Ima još dosta takvih instrukcija koje omogućavaju da se rade operacije atomički
- Takođe imaju načini da se zatraži da instrukcije poštuju određene *memorijske modele*.
- Ovo naravno zavisi kako od arhitekture na kojoj ovo koristite.

## Memorijski modeli

- Normalno procesor i kompjajler su u zaveri da se određenim instrukcijama menja redosled ne bi li se povećale performanse.
- Sve se vrti oko procesa koji se zove **pipelining**
- Mi možemo zatražiti da se na to ponašanje nametnu ograničenja od limitiranih do toga da zatražimo 'sekvencijalnu konzistentnost' što znači da se izvršavanje dešava tačno kako mi kažemo.

## Kako koristiti hardversku podršku?

- C++ pruža zaglavje `<atomic>` koje sadrži tipove koji označavaju da su određeni primitivni tipovi (int, recimo) atomički
- To znači da se nad njima mogu pozivati isključivo *atomičke operacije*
- Umesto da slobodno pristupamo memoriji moramo da je učitamo sa `.load()` (i tako dobijemo povratnu vrednost) ili da smeštamo vrednosti sa `.store()`
- Iza kulisa, kompajler pretvara ove naše zahteve u kompleksne pozive instrukcija koje su same po sebi atomičke, a ako nema podršku, varalj tako što koristi iste mutekse itd. koje bi i mi koristili.

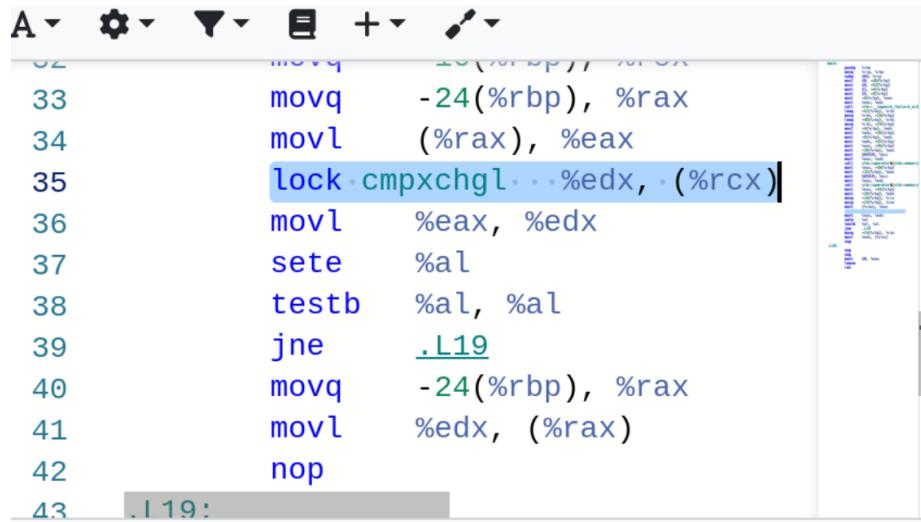
## Trivijalan primer

```
#include <atomic>

using namespace std;

int main(){
    int z = 0;
    atomic<int> a(0);
    a.compare_exchange_strong(z, 1);
    return 0;
}
```

## Izlaz kompjajlera



```
33     movq    -24(%rbp), %rax
34     movl    (%rax), %eax
35     lock   cmpxchgl %edx, (%rcx)
36     movl    %eax, %edx
37     sete    %al
38     testb   %al, %al
39     jne     .L19
40     movq    -24(%rbp), %rax
41     movl    %edx, (%rax)
42     nop
43 .L19:
```

## Zar to nismo negde videli?

- Da li je ova instrukcija poznata?
- I to čak sa *lock* prefiksom, kao što smo pričali.
- Naravno u zavisnosti od toga za koji kompjajler ovo radimo, ovo može da iza kulisa ima ili nema stvarne atomičke instrukcije

## Tipovi atomičkih operacija

- Možda ste primetili neobično ime 'compare\_exchange\_strong'
- To je zato što ova operacija dolazi u dve forme: snažna i slaba
- Snažna uvek radi ali je nešto sporija
- Slaba ponekad može da zakaže, ali samo tako što vrati rezultat da stvari koje se porede nisu iste kada jesu.
- Slaba operacija se preferira u petljama budući da će petlja da garantuje da će konverigrati tačnom rezultatu, a performanse su bolje.

## Primer upotrebe rukovanje baferima

```
1 #include <iostream>
2 #include <thread>
3 #include <atomic>
4 #include <cassert>
5
6 using namespace std;
```

```

7
8  struct List_member {
9      List_member* next;
10     char buffer[512];
11 };
12
13 class List {
14     atomic<List_member*> first;
15
16 public:
17     List () {
18         first.store(nullptr);
19     };
20     void link(List_member* member);
21     List_member* unlink();
22 };
23
24 void List::link(List_member* member){
25     member->next = first.load();
26     while(!first.compare_exchange_weak(member->next, member));
27 }
28
29 List_member* List::unlink(){
30     List_member* unlinked;
31     unlinked=first.load();
32     List_member* next = nullptr;
33     do{
34         if(unlinked == nullptr) return nullptr;
35
36         next = unlinked->next;
37     }while(!first.compare_exchange_weak(unlinked, next));
38     return unlinked;
39 }
40 void f(List* baferi){
41     List_member* x = nullptr;
42
43     for(int i = 0; i < 10000; i++){
44         if(x == nullptr){
45             x = baferi->unlink();
46         }else{
47             baferi->link(x);
48             x = nullptr;
49         }
50     }
51 }
52

```

```

53     if(x != nullptr){
54         baferi->link(x);
55     }
56 }
57
58 int main(){
59     List l;
60     List_member* b1 = new List_member();
61     List_member* b2 = new List_member();
62     List_member* b3 = new List_member();
63     List_member* b4 = new List_member();
64     List_member* b5 = new List_member();
65
66     l.link(b1);
67     l.link(b2);
68     l.link(b3);
69     l.link(b4);
70     l.link(b5);
71     thread t1(f, &l);
72     thread t2(f, &l);
73
74     t1.join();
75     t2.join();
76     List_member* p;
77     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
78     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
79     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
80     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
81
82     assert((nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa");
83     assert((nullptr == l.unlink()) && "U klasi ima sufficit bafera posle testa");
84     delete b1;
85     delete b2;
86     delete b3;
87     delete b4;
88     delete b5;
89
90     return 0;
91 }
```

## Pobeda?

- Deluje kao da je sve OK, zar ne?
- Nije.
- Probajte slobodno da napravite treću nit i videćete da se mani-

- festuju problemi koje je teško opisati rečima
- Potpun haos, štetno preplitanje, užas.
- Što? ABA

## ABA

- ABA je noćna mora koja vas može proganjati kada radite atomske operacije
  - U pitanju je sekvenca promena koja ostavi onu promenljivu sa kojom atomski operišete u naizgled dobrom stanju koje maskira, u stvari, nekonzistentno stanje.
  - Hajde da to razumemo u kontekstu baš ove naše klase.
1. Nit 1 počne unlink operaciju i vidi na vrhu A i B ('vidimo' dve stvari jer imamo i next)
  2. Nit 2 počne unlink operaciju i završi je vrativši A
  3. Nit 2 počne link operaciju i ubaci u listu neko D.
  4. Nit 2 počne link operaciju i ubaci u listu ono isto A koje je izvadila.
  5. Nit 1 vidi na vrhu A i zaključi da nije došlo do desinhronizacije i završi svoj unlink što znači da smo čvor D preskočili.
- Naravno ABA može da uradi i druge probleme
  - U stvari, može da uradi praktično sve što može i štetno preplitanje
  - Ne javlja se uvek, naravno, ali bilo kada imamo stanje koje može da izgleda isto spolja a nije, u opasnosti smo

## Rešenje?

- Ubacimo polje za određivanje verzije
- Sada kada radimo `compare_and_swap` operaciju sve što treba da uradimo jeste da je uradimo nad dve vrednosti istovremeno.
- Jedna vrednost je pokazivač, druga je ista kao pokazivač u dimenzijama ali sadrži samo monotono rastuć broj verzije promene što garantuje detekciju ABA problema (osim ako nemamo overflow, ali na 64-bitnim arhitekturama to nije veliki rizik)
- Da li to uopšte može?

## DWCAS

- Proizvođači procesora su nam to omogućili
- Generički termin za ovo je Double Word Compare and Swap
- Intel to zove `cmpxchg16b` i dostupno je na novim procesorima (Pogledajte da li se u `/proc/cpuinfo` nalazi flag `cx16`)

- Onda nam samo treba da napravimo umesto pokazivača struct koji je pokazivač + verzija i da ubedimo kompjajler da emituje ovu korisnu instrukciju i sve je lako

## Lako?

- Nikako.
- Prvo, ima teškoća u implementaciji ovoga kako valja ali sa tim se, uz pažnju, da izboriti.
- Kompajjeri će vam prirediti mnogo gori problem: kako stvari stoje GCC, recimo, jednostavno *odbija* da emituje DWCAS instrukciju maltene šta god vi radili.
- Jedini kompjajler na kome sam uspeo da proizvedem korektan kod je clang a i on zahteva poseban tretman – struct koji koristite za DWCAS mora biti eksplisitno poravnan u memoriji na 16 bajtova što obično nije potrebno.

## Verzija sa DWCAS

```

1 #include <iostream>
2 #include <thread>
3 #include <atomic>
4 #include <cassert>
5 #include <cstdint>
6
7 using namespace std;
8
9 struct List_member {
10     List_member* next;
11     char buffer[512];
12 };
13
14 struct alignas(2 * sizeof(void*)) p_aba {
15
16     uintptr_t aba;
17     List_member* p;
18 };
19
20 class List {
21     atomic<p_aba> first;
22 public:
23
24     List () {
25         first.store({0, nullptr});

```

```

26     };
27     void link(List_member* member);
28     List_member* unlink();
29 };
30
31 void List::link(List_member* member){
32     p_aba found = first.load();
33     p_aba next;
34     do{
35         member->next = found.p;
36         next.aba = found.aba + 1;
37
38         next.p = member;
39     }while(!first.compare_exchange_weak(found, next));
40 }
41 List_member* List::unlink(){
42     p_aba found = first.load();
43
44     p_aba novi;
45     do{
46         if(found.p == nullptr) return nullptr;
47         novi.p = found.p->next;
48         novi.aba = found.aba + 1;
49     }while(!first.compare_exchange_weak(found, novi));
50     return found.p;
51 }
52 }
53
54 void f(List* baferi){
55     List_member* x = nullptr;
56     for(int i = 0; i < 10000; i++){
57         if(x == nullptr){
58             x = baferi->unlink();
59
60         }else{
61             baferi->link(x);
62             x = nullptr;
63         }
64     }
65
66     if(x != nullptr){
67
68         baferi->link(x);
69     }
70 }
71

```

```

72 int main(){
73     List l;
74     List_member* b1 = new List_member();
75
76     List_member* b2 = new List_member();
77     List_member* b3 = new List_member();
78     List_member* b4 = new List_member();
79     List_member* b5 = new List_member();
80     l.link(b1);
81     l.link(b2);
82     l.link(b3);
83
84     l.link(b4);
85     l.link(b5);
86     thread t1(f, &l);
87     thread t2(f, &l);
88     thread t3(f, &l);
89     t1.join();
90     t2.join();
91
92     t3.join();
93     List_member* p;
94     assert(nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa";
95     assert(nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa";
96     assert(nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa";
97     assert(nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa";
98     assert(nullptr != l.unlink()) && "U klasi nema dovoljno bafera posle testa";
99
100    assert(nullptr == l.unlink()) && "U klasi ima sufficit bafera posle testa";
101    delete b1;
102    delete b2;
103    delete b3;
104    delete b4;
105    delete b5;
106    return 0;
107 }
108

```

## Kako se ovo kompajlira?

```
clang++ -O3 -pthread -std=c++20 -mcpu=x64 -o lld lockless_double.cpp
```

## Da li stvarno emituje instrukciju?

```
objdump -d lld|grep cmpxchg16b
```

## Rezultat

40142a:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
40143a:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
40144a:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
40145a:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
40146b:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
40147b:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
40149b:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
4014ab:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
4014cb:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
4015e0:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
401600:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
401610:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
401630:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
401640:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
401660:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
401670:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
401690:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)
4016a0:	f0 48 0f c7 0c 24	lock cmpxchg16b (%rsp)

## Šta možemo da naučimo iz ovoga?

- Atomsko programiranje je opasno - kao što ime i sugeriše.
- Problemi su jako izazovni i kompleksni za razumevanje i zavise od hardvera koji nikako nije univerzalan
- Debagovanje je efektivno nemoguće
- Dokumentacija je vrlo štura i, recimo, primer dat na cppreference (inače vrlo pouzdan izvor) sadrži ABA problem ako se implementira.
- Treba se baviti ovim samo ako ste *veoma* sigurni da znate šta radite, i ako ste uvereni da će posledice biti vredne truda.
- Naročito morate se postarate, čak i pre nego počnete, da su sledeće stvari tačne:
  - Performanse su apsolutno kritične
  - Vreme koje je neophodno za lock-ovanje u nekoj strukturi je važan faktor usporenja
  - Greška u implementaciji neće imati pogubne posledice, tj. ono što programirate sme da se sruši a da se ne desi kataklizma.
- Čak i tada treba dvaput razmisliti: u zavisnosti od neverovatno suptilnih razlika u tome kako upravljate memorijom može se desiti da dobijete brži kod na x86\_64 platformi (dobra uređenost)

i kod koji je sporji nego da smo koristili mutex na ARM platformi zato što je ona vrlo sklona, arhitektonski govoreći, out-of-order egzekuciji.

- Dalje, ako je to moguće, valja koristiti gotove implementacije kao što su one u standardnoj biblioteci ili u bibliotekama kao što je Folly i Abseil
- Nemojte podceniti težinu rada na ovom nivou: pravljenje nečega tako prostog kao što je jednostruko linkovana lista koja je bez lock-ova a *korektna* je bio ozbiljan istraživački projekat.
- Ne samo to, ekserti i dalje znaju da se zbune: bag vezan za baš ove probleme je vrebao u Linux kernelu **deset godina**.

## Pitanja

### Pitanja

- Šta karakteriše semafor?
- Koje operacije su vezane za semafor?
- Kako semafor obezbeđuje sinhronizaciju međusobne isključivosti?
- Kako se obično implementira semafor?
- U čemu se semafori razlikuju od isključivih regiona?
- Koji semafori postoje?
- Šta karakteriše binarni semafor?
- Šta karakteriše raspodeljeni binarni semafor?
- Šta karakteriše generalni semafor?
- Šta omogućuje raspodeljeni binarni semafor?
- Šta omogućuje binarni semafor?
- Šta omogućuje generalni semafor?
- Koje su prednosti i mane semafora?
- Šta je lockless programiranje / programiranje bez zaključavanja?
- Šta je ABA problem?
- Šta je DWCAS i zašto je potreban?
-

# **Šta su mane a šta prednosti programiranja bez zaključavanja / lockless programiranja?**

title: Operativni Sistemi - Simulator operativnog sistema 1 author: Veljko Petrović date: Mart, 2024 —

## **Uvod**

### **Šta simuliramo**

- Ovo je simulator operativnog sistema koji radi pod određenim znatnim ograničenjima
- Simulira (isključivo) jednoprocesorni operativni sistem i to kroz konkurentnu biblioteku
- Šta znači da simulira kroz konkurentnu biblioteku? Da je interfejs prema ovome onaj isti C++ interfejs ka nitima, sa tom razlikom da umesto da koristimo POSIX niti, mi sami realizujemo prebacivanje između niti i to u obliku simulatora operativnog sistema.
- Radi (isključivo) na 32-bitnoj x86 arhitekturi koju ste koristili prošli semestar.
- Ništa od ovih stvari nisu 100% aktuelne, ali su odličan smanjeni skup funkcionalnosti koji nam može pomoći da razumemo osnovne principe.
- Gde je prigodno tokom kursa, mi skačemo u kod Linux kernela da vidimo apsolutno aktuelnu i ozbiljnu verziju ovoga
- Termin koji povremeno koristimo za simulator je i CppTss pošto implementiramo podskup za niti iz ranog 2011 standarda za C++ zato što je relativno jednostavan za razumevanje.

### **Šta ne simuliramo**

- Ništa od ovoga nije pravi operativni sistem u klasičnom smislu
- Ako vas zanima kako se operativni sistem startuje od nule na pravom hardveru: sačekajte. Pričaćemo i o tome.
- Za sada nas zanima da povežemo gradivo ovde sa gradivom iz AR na nečemu što je dovoljno pojednostavljeno da celo može da stane na predavanja.

# Osnove simulatora

## Atomski regioni

- Stvaranje atomskih regiona omogućuje klasa `Atomic_region`.
- Njen konstruktor onemogućuje prekide, a destruktur vraća prekide u stanje koje je prethodilo akciji konstruktora.

```
1  {
2      Atomic_region ar;
3      //kod
4 }
```

- Upotrebu atomskog regiona ilustruje primer rukovanja pozicijom kursora u kome nisu moguća prekide.
- Pošto se operacija `set()` poziva samo iz obrada prekida, njeno telo po definiciji obrazuje prekide.

## Primer sa kurzorom

```
1  class Position {
2      int x, y;
3  public:
4      Position();
5      void set(int new_x, int new_y);
6      void get(int* current_x, int* current_y);
7  };
8
9  Position::Position(){
10     x = 0;
11     y = 0;
12 }
13
14 void Position::set(int new_x, int new_y){
15     //ne mora atomski ako su prekidi prekida zabranjeni
16     x = new_x;
17     y = new_y;
18 }
19
20 void Position::get(int* current_x, int* current_y){
21     Atomic_region ar;
22     *current_x = x;
23     *current_y = y;
24 }
```

## Klase Driver

- Pisanje drajvera olakšava klasu `Driver`.

- Nju nasleđuju klase koje opisuju ponašanje drajvera.
- Drajvere karakteriše saradnja obrađivača prekida i pozadinskih niti.
- U okviru klase, koja opisuje ponašanje drajvera, obrađivač prekida se predstavlja u obliku funkcije bez povratne vrednosti i bez parametara.
- Ova funkcija mora biti `static`, da bi se mogla koristiti njena adresa.
- Takva moraju biti i sva polja klase kojima ona pristupa.
- Operacija `start_interrupt_handling()` klase `Driver` omogućuje smeštanje adrese obrađivača prekida u tabelu prekida.
- Prvi argument poziva ove operacije predstavlja broj vektora prekida, a drugi adresu obrađivača prekida.
- Saradnja obrađivača prekida i pozadinskih niti podrazumeva da pozadinske niti čekaju dešavanje spoljnih događaja, a da obrađivači prekida objavljuju dešavanje spoljnih događaja.

## Neki primeri iz stvarne prakse - Linux

```

int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev)

typedef irqreturn_t (*irq_handler_t)(int, void *);

static irqreturn_t intr_handler(int irq, void *dev)

```

### Event i Driver

- Klasa `Driver` ima kao povezani klasu `Event`
- `signal` metoda

- Poziva se iz atomskog regiona
- Nit prelazi u čekanje
- Preključivanje na spremnu nit
- expect metoda
  - Poziva se iz obrađivača prekida
  - Nit koja čeka najduže prelazi u stanje spremna
  - Poziva se na kraju obrade prekida samo jednom

## Drajver za rukovanje vremenom

- Ovaj drajver registruje proticanje vremena u računaru brojanjem otkucaja sata.
- Otkucaji, čiji period zavisi od takta procesora, nisu uvek podesni za određivanje vremena u danu, predstavljenog brojem sati, minuta i sekundi, jer sekunda ne može uvek da se izrazi celim brojem perioda ovakvih otkucanja.
- Zato je zgodno uvesti dodatni sat, čiji otkucaji (prekidi) imaju period od tačno jedne sekunde.
- Ova komponenta se, tradicionalno, zove 'Real Time Clock' u računarima i vi je imate na matičnoj ploči, tipično zakčenoj za SRAM modul i CR2032 bateriju.
- Rukovanje ovim satom, odnosno rukovanje vremenom opisuje klasa `Timer_driver`.
- Njena tri celobrojna polja: `hour`, `minute` i `second` sadrže broj proteklih sati, minuta i sekundi.
- Početni sadržaj ovih polja određuje konstruktor klase `Timer_driver`.
- Pored toga on u element tabele prekida, koga indeksira konstanta `TIMER` (broj vektora prekida dodatnog sata), smesti adresu njene operacije `interrupt_handler()`, koja je zadužena za periodičnu izmenu sadržaja polja `hour`, `minute` i `second`, sa periodom od jedne sekunde.
- Klasa `Timer_driver` sadrži i operacije `set()` i `get()` za zadavanje i preuzimanje sadržaja njenih polja.
- To su osetljive operacije, čije preplitanje sa obradom prekida sata je štetno.
- Na primer, ako se obrada prekida sata desi nakon preuzimanja sadržaja polja `hour`, a pre preuzimanja sadržaja polja `minute`, i ako je, uz to, broj minuta pre periodične izmene bio na granici od 59, tada preuzeto vreme kasni iza stvarnog za 60 minuta.

- Da bi se ovakva štetna preplitanja sprečila, preuzimanja i zadatavanja sadržaja njenih polja moraju da budu u atomskim regionima.

### Klasa Timer\_driver

```

1  class Timer_driver : public Driver {
2      static int hour;
3      static int minute;
4      static int second;
5      static void interrupt_handler();
6  public:
7      Timer_driver()
8      { start_interrupt_handling(TIMER,
9          interrupt_handler); };
10     void set(const int h, const int m,
11             const int s);
12     void get(int* h, int* m, int* s) const;
13 };
14
15 int Timer_driver::hour = 0;
16 int Timer_driver::minute = 0;
17 int Timer_driver::second = 0;
18
19 void Timer_driver::interrupt_handler() {
20     if(second < 59)
21         second++;
22     else {
23         second = 0;
24         if(minute < 59) minute++;
25         else {
26             minute = 0;
27             if(hour < 23) hour++;
28             else hour = 0;
29         }
30     }
31 }
32
33 void Timer_driver::set(const int h, const int m, const int s){
34     Atomic_region ar;
35     hour = h;
36     minute = m;
37     second = s;
38 }
39

```

```

40 void Timer_driver::get(int* h, int* m, int* s) const{
41     Atomic_region ar;
42     *h = hour;
43     *m = minute;
44     *s = second;
45 }

```

## Klasa Sleep\_driver

- Upotrebu klase Driver ilustruje i primer drajvera koji omogućuje uspavljivanje jedne niti dok ne protekne zadani broj otkucaja sata.
- Ovo uspavljanje može da se prikaže kao očekivanje dešavanja zadanog broja otkucaja sata.
- To opisuje klasu Sleep\_driver.
- Njena operacija simple\_sleep\_for() omogućuje jednoj niti da zaustavi svoju aktivnost dok se ne desi zadani broj otkucaja sata.
- Zadatak obrađivača prekida (operacije interrupt\_handler()) je da odbroji zadani broj otkucaja sata i da nakon toga signalizira da je moguć nastavak aktivnosti uspavane niti.

## Klasa Sleep\_driver

```

1 class Sleep_driver: public Driver {
2     static unsigned long countdown;
3     static Event alarm;
4     static void interrupt_handler();
5 public:
6     Sleep_driver(
7         { start_interrupt_handling(TIMER,
8             interrupt_handler); };
9     void simple_sleep_for(
10         unsigned long duration);
11 };
12
13 unsigned long Sleep_driver::countdown = 0;
14
15 void Sleep_driver::interrupt_handler(){
16     if((countdown > 0) && ((--countdown) == 0))
17         alarm.signal();
18 }
19
20 void Sleep_driver::simple_sleep_for(unsigned long duration){
21     Atomic_region ar;

```

```

22     if(duration > 0) {
23         countdown = duration;
24         alarm.expect();
25     };
26 }
27
28 # Ulagno/izlagni moduli

```

## Drajveri tastature i ekrana

- Klasa Display\_driver sadrži dajver ekrana koji upravlja kontrolerom ekrana.
- Kontroler ekrana (objekat display\_controller) sadrži registar stanja (display\_controller.status\_reg) i registar podataka (display\_controller.data\_reg).
- Prikaz znaka na ekranu je moguć ako registar stanja sadrži konstantu DISPLAY\_READY.
- U tom slučaju se kod znaka smešta u registar podataka, a u registar stanja se smešta konstanta DISPLAY\_BUSY.
- Ova konstanta ostaje u registru stanja dok traje prikaz znaka.
- Po prikazu znaka, kontroler ekrana smešta u registar stanja konstantu DISPLAY\_READY (podrazumeva se da se ova vrednost nalazi u registru stanja na početku rada kontrolera ekrana).
- Pokušaj niti da prikaže znak, dok je u registru stanja konstanta DISPLAY\_BUSY, zaustavlja aktivnost niti.
- Nastavak aktivnosti niti usledi nakon obrade prekida ekrana, koja objavljuje da je prikaz prethodnog znaka završen.
- Zaustavljanje i nastavak aktivnosti niti omogućuje polje displayed\_char klase Display\_driver.
- Opisano ponašanje dajvera ekrana ostvaruju operacije character\_put() i interrupt\_handler() klase Display\_driver.
- Broj vektora prekida ekrana određuje konstanta DISPLAY.

## Drajver Ekrana

```

1 class Display_driver : public Driver {
2     static Event displayed_char;
3     static void interrupt_handler();
4     Display_driver(const Display_driver&);
5     Display_driver& operator=(const Display_driver&);

```

```

6  public :
7      Display_driver()
8
9      { start_interrupt_handling(DISPLAY,
10          interrupt_handler); };
11      void character_put(const char c);
12  };
13
14 Display_driver::Event
15 Display_driver::displayed_char;
16
17 void Display_driver::interrupt_handler()
18 {
19     displayed_char.signal();
20 }
21
22 void Display_driver::character_put(const char c)
23 {
24     Atomic_region ar;
25     if(display_controller.status_reg == DISPLAY_BUSY)
26         displayed_char.expect();
27     display_controller.data_reg = c;
28     display_controller.status_reg = DISPLAY_BUSY;
29 }
30
31
32 static Display_driver display_driver;

```

## Drajver tastature

- Klasa Keyboard\_driver sadrži dajver tastature koji upravlja kontrolerom tastature.
- Kontroler tastature (objekat keyboard\_controller) sadrži registr podataka (keyboard\_controller.data\_reg).
- Podrazumeva se da pritisak dirke na tastaturi:
  - doveđe do smeštanja koda odgovarajućeg znaka u registar podataka.
  - izazove prekid tastature.
- Pomenuti kod znaka se preuzima iz registra podataka u obradi prekida tastature i smešta u cirkularni bafer, ako on nije pun.
- Cirkularnom baferu odgovara polje buffer klase Keyboard\_driver.
- Njeno polje count određuje popunjenošć ovog bafera.

- Indekse cirkularnog bafera sadrže polja `first_full` i `first_empty` klase `Keyboard_driver`.
- Pokušaj niti da preuzme znak, kada je cirkularni bafer prazan, zaustavlja njenu aktivnost.
- Nastavak aktivnosti niti usledi nakon obrade prekida tastature.
- Zaustavljanje i nastavak aktivnosti niti omogućuje polje `pressed` klase `Keyboard_driver`.
- Opisano ponašanje drajvera tastature ostvaruju operacije `character_get()` i `interrupt_handler()` klase `Keyboard_driver`.
- Broj vektora prekida tastature određuje konstanta `KEYBOARD`.

```

1 const unsigned
2 KEYBOARD_BUFFER_SIZE = 1024;
3
4 class Keyboard_driver : public Driver {
5     static Event pressed;
6     static char buffer[KEYBOARD_BUFFER_SIZE];
7     static unsigned count;
8
9     static unsigned first_full;
10    static unsigned first_empty;
11    static void interrupt_handler();
12    Keyboard_driver(const Keyboard_driver&);
13    Keyboard_driver& operator=(const Keyboard_driver&);
14 public:
15     Keyboard_driver()
16
17     { start_interrupt_handling(KEYBOARD,
18         interrupt_handler); };
19     char character_get();
20 };
21
22 Keyboard_driver::Event
23 Keyboard_driver::pressed;
24
25 char Keyboard_driver::buffer[KEYBOARD_BUFFER_SIZE];
26 unsigned Keyboard_driver::count = 0;
27 unsigned Keyboard_driver::first_full = 0;
28 unsigned Keyboard_driver::first_empty = 0;
29
30 void Keyboard_driver::interrupt_handler(){
31     if(count<KEYBOARD_BUFFER_SIZE) {
32
33         buffer[first_empty++] =

```

```

34     keyboard_controller.data_reg;
35     if(first_empty == KEYBOARD_BUFFER_SIZE)
36         first_empty = 0;
37     count++;
38     pressed.signal();
39 }
40 }
41
42 char Keyboard_driver::character_get(){
43     char c;
44     Atomic_region ar;
45     if(count==0)
46         pressed.expect();
47
48     c = buffer[first_full++];
49     if(first_full == KEYBOARD_BUFFER_SIZE)
50         first_full = 0;
51     count--;
52     return c;
53 }
54 static Keyboard_driver keyboard_driver;

```

## Znakovni ulaz-izlaz

- Prilikom ulaza-izlaza znakova moguća su štetna preplitanja. Sprečavanje štetnih preplitanja ulaznih i izlaznih operacija podrazumeva da su one međusobno isključive.
- Njihova međusobna isključivost se može ostvariti, ako se tastatura, odnosno ekran zaključa (zauzme) pre i otključa (oslobodi) nakon korišćenja, prilikom svakog izvršavanja odgovarajuće operacije.
- Neuspisan pokušaj zaključavanja uređaja dovodi do zastavljanja izvršavanja ovakve operacije, dok zaključavanje ne postane moguće.
- Zaključavanje tastature i ekrana, tokom izvršavanja ulaznih i izlaznih operacija, se zasniva na korišćenju propusnica.
- Posebne propusnice reprezentuju ekran i tastaturu.
- Zauzimanje propusnice odgovara zaključavanju njenog uređaja, a oslobođanje propusnice odgovara otključavanju dotičnog uređaja.
- Time se obezbeđuje međusobna isključivost pojedinačnih ulaznih i izlaznih operacija.

- Klasa Terminal\_out omogućuje znakovni izlaz, odnosno prikaz znaknova na ekranu.
- Ona sadrži operacije koje omogućuju formatiranje prikazivanog podatka (njegovo pretvaranje u niz znakova).
- Oznake %6d, %6u, %11d i %11u određuju broj cifara u decimalnom formatu u kome se prikazuju cifre celih označenih (d) i neoznačenih (u) brojeva, a oznaka %.3e određuje broj cifara iza decimalne tačke u decimalnom formatu u kome se prikazuju cifre razlomljenih brojeva.
- Za prikaz niza znakova (znakovni izlaz) zadužena je operacija string\_put() klase Terminal\_out, koja se brine i o zaključavanju ekrana.

## Klasa Terminal\_out

```

1  const char endl[] = "\n";
2
3  class Terminal_out : private mutex {
4      Terminal_out(const Terminal_out&);
5      Terminal_out& operator=(const Terminal_out&);
6      void string_put(const char* string);
7  public:
8      Terminal_out() {};
9      Terminal_out& operator<<(int number);
10     Terminal_out& operator<<(unsigned int number);
11     Terminal_out& operator<<(short number);
12     Terminal_out& operator<<(unsigned short number);
13     Terminal_out& operator<<(long number);
14     Terminal_out& operator<<(unsigned long number);
15
16     Terminal_out& operator<<(double number);
17     Terminal_out& operator<<(char character);
18     Terminal_out& operator<<(const char* string);
19     friend class Terminal_in;
20 };
21
22 static const char* SHORT_FORMAT = "%6d";
23
24 static const char* UNSIGNED_SHORT_FORMAT = "%6u";
25 static const char* INT_FORMAT = "%11d";
26 static const char* UNSIGNED_FORMAT = "%11u";
27 static const char* DOUBLE_FORMAT = "% .3e";
28 static const int SHORT_SIGNIFICANT FIGURES COUNT = 5;
29

```

```

30 static const int INT_SIGNIFICANT FIGURES COUNT = 10;
31 static const int DOUBLE_SIGNIFICANT FIGURES COUNT = 10;
32
33
34 void Terminal_out::string_put(const char* string)
35 {
36     lock();
37     while(*string != '\0')
38         display_driver.character_put(*string++);
39     unlock();
40
41 }
42
43 Terminal_out& Terminal_out::operator<<(int number)
44 {
45     char string[INT_SIGNIFICANT FIGURES COUNT + 2];
46     sprintf(string, INT_FORMAT, number);
47     string_put(string);
48
49     return *this;
50 }
51
52 Terminal_out& Terminal_out::operator<<(unsigned int number)
53 {
54     char string[INT_SIGNIFICANT FIGURES COUNT + 2];
55     sprintf(string, UNSIGNED_FORMAT, number);
56
57     string_put(string);
58     return *this;
59 }
60
61 Terminal_out& Terminal_out::operator<<(short number)
62 {
63     char string[SHORT_SIGNIFICANT FIGURES COUNT + 2];
64
65     sprintf(string, SHORT_FORMAT, number);
66     string_put(string);
67     return *this;
68 }
69
70 Terminal_out& Terminal_out::operator<<(unsigned short number)
71 {
72
73     char string[SHORT_SIGNIFICANT FIGURES COUNT + 2];
74     sprintf(string, UNSIGNED_SHORT_FORMAT, number);
75     string_put(string);

```

```

76     return *this;
77 }
78
79 Terminal_out& Terminal_out::operator<<(long number)
80
81 {
82     char string[INT_SIGNIFICANT FIGURES_COUNT + 2];
83     sprintf(string, INT_FORMAT, number);
84     string_put(string);
85     return *this;
86 }
87
88 Terminal_out& Terminal_out::operator<<(unsigned long number)
89 {
90     char string[INT_SIGNIFICANT FIGURES_COUNT + 2];
91     sprintf(string, UNSIGNED_FORMAT, number);
92     string_put(string);
93     return *this;
94 }
95
96 Terminal_out& Terminal_out::operator<<(double number)
97 {
98     char string[DOUBLE_SIGNIFICANT FIGURES_COUNT + 2];
99     sprintf(string, DOUBLE_FORMAT, number);
100    string_put(string);
101    return *this;
102 }
103
104 Terminal_out& Terminal_out::operator<<(char character)
105 {
106     char string[2];
107     string[0] = character;
108     string[1] = '\0';
109
110     string_put(string);
111     return *this;
112 }
113
114 Terminal_out& Terminal_out::operator<<(const char* string)
115 {
116     string_put(string);
117
118     return *this;
119 }
120
121 Terminal_out cout;

```

## Klasa Terminal\_in

- Klasa Terminal\_in omogućuje preuzimanje znakova (znakovni ulaz).
- Ona obezbeđuje zaključavanje tastature dok se izvršava njena operacija string\_get(), kao i zaključavanje ekrana, radi eha znakova, preuzetih u ovoj operaciji.
- Operacija string\_get() poziva operaciju edit() klase Terminal\_in koja je zadužena za echo znakova na ekranu i njihovo primitivno editiranje.
- Preostale operacije klase Terminal\_in koriste operaciju string\_get() za preuzimanje nizova znakova koji odgovaraju raznim tipovima podataka.

## Klasa Terminal\_in

```

1 const int INPUT_BUFFER_LENGTH = 128;
2
3 class Terminal_in : private mutex {
4     Terminal_in(const Terminal_in&);
5     Terminal_in& operator=(const Terminal_in&);
6     unsigned index;
7     char c;
8
9     bool pressed_enter;
10    char buff[INPUT_BUFFER_LENGTH];
11    inline void edit();
12    void string_get(unsigned figures_count);
13 public:
14     Terminal_in() {};
15     Terminal_in& operator>>(int &number);
16
17     Terminal_in& operator>>(short &number);
18     Terminal_in& operator>>(long &number);
19     Terminal_in& operator>>(double &number);
20     Terminal_in& operator>>(char &character);
21 };
22
23 static const int SHORT_SIGNIFICANT FIGURES_COUNT = 5;
24
25 static const int INT_SIGNIFICANT FIGURES_COUNT = 10;
26 static const int DOUBLE_SIGNIFICANT FIGURES_COUNT = 10;
27

```

```

28 #define CHAR_ESC (27)
29 #define CHAR_LF ('\n')
30 #define CHAR_BS1 ('\b')
31 #define CHAR_BS2 (127)
32
33
34 void Terminal_in::edit() //radi echo znakova preuzetih sa tastature na ekran
35 {
36     switch(c) {
37         case CHAR_ESC:
38             buff[index++]=c;
39             display_driver.character_put('^'); //esc caret karakter
40             break;
41         case CHAR_LF:
42             pressed_enter = true;
43             break;
44         case CHAR_BS1:
45         case CHAR_BS2:
46             if(index>0) {
47
48                 buff[--index]='\0';
49                 display_driver.character_put('\b'); //pomeranje kursora nazad
50                 display_driver.character_put(' '); //stavljanje space-a
51                 display_driver.character_put('\b'); //pomeranje kursora nazad
52             }
53             break;
54         default:
55
56             buff[index++]=c;
57             display_driver.character_put(c);
58             break;
59         }
60     buff[index]='\0';
61 }
62 }
63
64 void Terminal_in::string_get(unsigned figures_count)
65 {
66     lock(); //zakljucaj tastaturu
67     index = 0;
68     pressed_enter = false;
69     c = keyboard_driver.character_get();
70     cout.lock(); //zakljucaj terminal
71
72     edit(); //ispisi prvi karakter
73     while((index < (figures_count - 1)) &&

```

```

74             !pressed_enter) { //ispisuj do entera tj. do precizn.
75             c = keyboard_driver.character_get();
76             edit();
77         }
78         cout.unlock();           //otkljucaj terminal
79
80         unlock();               //otkljucaj tastaturu
81     }
82
83 Terminal_in& Terminal_in::operator>>(int& number)
84 {
85     string_get(INT_SIGNIFICANT FIGURES_COUNT);
86     number = (int)strtol(buff, 0, 10);
87
88     return *this;
89 }
90
91 Terminal_in& Terminal_in::operator>>(short& number)
92 {
93     string_get(SHORT_SIGNIFICANT FIGURES_COUNT);
94     number = (int)strtol(buff, 0, 10);
95
96     return *this;
97 }
98
99 Terminal_in& Terminal_in::operator>>(long& number)
100 {
101     string_get(INT_SIGNIFICANT FIGURES_COUNT);
102     number = (int)strtol(buff, 0, 10);
103
104     return *this;
105 }
106
107 Terminal_in& Terminal_in::operator>>(double& number)
108 {
109     string_get(DOUBLE_SIGNIFICANT FIGURES_COUNT);
110     number = strtod(buff, 0);
111
112     return *this;
113 }
114
115 Terminal_in& Terminal_in::operator>>(char& character)
116 {
117     string_get(1);
118     character = buff[0];
119

```

```

120     return *this;
121 }
122
123 Terminal_in cin;

```

## Klasa Disk\_driver

- Klasa Disk\_driver sadrži drajver diska koji upravlja DMA kontrolerom diska.
- DMA kontroler diska (objekat disk\_controller) sadrži:
  - registar bloka (disk\_controller.block\_reg)
  - registar bafera (disk\_controller.buffer\_reg)
  - registar smera prenosa (disk\_controller.operation\_reg)
  - registar stanja (disk\_controller.status\_reg).
- Podrazumeva se da nit pokreće prenos bloka tako što:
  - u registar bloka smesti broj prenošenog bloka
  - u registar bafera smesti adresu bafera koji učestvuje u prenosu
  - u registar smera prenosa smesti konstantu DISK\_READ ili DISK\_WRITE
  - u registar stanja konstantu DISK\_STARTED
- Nakon toga aktivnost niti se zaustavi dok traje prenos bloka.
- Kraj prenosa bloka objavi prekid diska.
- Zaustavljanje i nastavak aktivnosti omogućuje polje ready klase Disk\_driver. Opisano ponašanje drajvera diska ostvaruju operacije block\_transfer() i interrupt\_handler() klase Disk\_driver.
- Broj vektora prekida diska određuje konstanta DISK.

```

1 class Disk_driver : public Driver {
2     static Event ready;
3     static void interrupt_handler();
4     Disk_driver(const Disk_driver &);
5     Disk_driver& operator=(const Disk_driver &);
6 public:
7     Disk_driver(void) { start_interrupt_handling(DISK,
8
9                     interrupt_handler); }
10    inline int block_transfer(char* buffer, unsigned block,
11                               Disk_operations operation);
12 };
13 Disk_driver::Event Disk_driver::ready;
14
15 void Disk_driver::interrupt_handler()
16 {
17     ready.signal();

```

```

18 }
19
20 int Disk_driver::block_transfer(char* buffer,
21                               unsigned block, Disk_operations operation)
22
23 {
24     int r = -1;
25     if(block < DISK_BLOCKS) {
26         Atomic_region ar;
27         disk_controller.block_reg = block;
28         disk_controller.buffer_reg = buffer;
29         disk_controller.operation_reg = operation;
30
31         disk_controller.status_reg = DISK_STARTED;
32         ready.expect();
33         r = 0;
34     }
35     return r;
36 }
37
38 static Disk_driver disk_driver;
39

```

## Blokovski ulaz-izlaz

- Klasa Disk opisuje rukovanje virtuelnim (magneto-rotacionim) diskom.
- Ova klasa definiše operacije `block_get()` i `block_put()` koje omogućuju preuzimanje bloka sa diska i smeštanje bloka na disk.
- Prvi parametar ovih operacija pokazuje na niz od 512 bajta radne memorije koji učestvuje u prebacivanju bloka, a drugi parametar određuje broj bloka (u rasponu od 0 do 999).
- Obe operacije su blokirajuće.
- Pošto brzina pomeranja glave diska ograničava ukupnu brzinu diska, važno je skratiti put koji glava diska prelazi.
- Optimizacija kretanja glave diska je moguća kada se skupi, više zahteva za čitanjem ili pisanjem.
- Optimizacija se svodi na opsluživanje zahteva u redosledu staza na koje se oni odnose, a ne u hronološkom redosledu pojave zahteva.

- Na taj način se izbegava da glava diska osciluje između vanjskih i unutrašnjih staza diska.
- Da bi optimizacija kretanja glave diska bila moguća, neophodno je uticati na redosled zahteva za čitanjem ili pisanjem blokova.
- To postavlja specifične zahteve na implementaciju klase `condition_variable` (proširenje klase).
- Podrazumeva se da odabrani redosled deskriptora niti u listi uslova nastaje na osnovu privezaka koji se dodeljuju svakom deskriptoru prilikom njegovog uvezivanja u ovu listu.
- Privesci imaju oblik neoznačenih celih brojeva, a njihovo dodjivanje deskriptoru omogućuje dodatni, drugi parametar operacije `wait()`.
- Uticaj na redosled deskriptora niti u listi uslova omogućuju operacije `first()`, `next()` i `last()` klase `condition_variable`.
- Operacija `first()` omogućuje pozicioniranje pre prvog deskriptora u listi uslova.
- Operacija `next()` omogućuje pozicioniranje pre narednog ili iza poslednjeg deskriptora u listi uslova.
- Operacija `last()` omogućuje pozicioniranje iza poslednjeg deskriptora u listi uslova.
- Podrazumeva se da operacija `notify_one()` klase `condition_variable` uvek izvezuje deskriptor sa početka liste uslova.
- Optimizaciju kretanja glave diska omogućuje operacija `optimize()` klase `Disk`.
- U situaciji kada je disk sloboden (kada polje state klase `Disk` sadrži konstantu `FREE`), poziv operacije `optimize()` dovodi do poziva blokirajuće operacije `disk_driver.block_transfer()`.
- U toku njenog izvršavanja disk je zaposlen (polje state sadrži konstantu `BUSY`), a aktivnost pozivajuće niti je zaustavljena.
- Ako u ovoj situaciji više niti, jedna za drugom, pozove operaciju `optimize()`, njihova aktivnost se zaustavlja, a njihovi deskriptori se uvezuju u jednu od dve liste uslova, koje odgovaraju polju `q` klase `Disk`.
- Polje `index` ove klase indeksira ili listu uslova namenjenu za deskriptore niti koje čitaju blokove između trenutnog položaja glave diska i njegovog oboda ili listu uslova namenjenu za deskriptore niti koje čitaju blokove između centra rotacije ploče diska i trenutnog položaja njegove glave.

- Podatak o trenutnom položaju glave diska (odnosno, o bloku koji se upravo čita) sadrži polje `boundary` klase `Disk`.
- Polje `index` može imati vrednost 0 ili 1.
- U pomenutim listama uslova deskriptori su poređani u rastućem redosledu brojeva blokova koje niti čitaju (što je u skladu sa optimizacijom kretanja glave diska).
- Pomenuti brojevi blokova predstavljaju priveske deskriptora iz listi uslova.

## Klasa Disk

```

1  const int DISK_ERROR = -1;
2
3  class Disk {
4      mutex mx;
5      enum Optimized_disk_state { FREE, BUSY };
6      Optimized_disk_state state;
7      unsigned boundary;
8
9      condition_variable q[2];
10     int index;
11     Disk(const Disk&);
12     Disk& operator=(const Disk&);
13     int optimize(char* buffer, unsigned block,
14                  Disk_operations operation);
15 public:
16
17     Disk() : state(FREE), boundary(0), index(0) {};
18     int block_get(char* buffer, unsigned block);
19     int block_put(char* buffer, unsigned block);
20 };
21
22 int
23 Disk::optimize(char* buffer, unsigned block, Disk_operations operation)
24 {
25     unsigned tag;
26     int i;
27     int status;
28     {
29         unique_lock<mutex> lock(mx);
30         if(state == BUSY) {
31
32             i = index;

```

```

34         if(block < boundary)
35             i = ((i == 0) ? (1) : (0));
36         if(q[i].first(&tag))
37             do {
38                 if(block < tag)
39                     break;
40
41             } while(q[i].next(&tag));
42             q[i].wait(lock, block);
43         }
44         state = BUSY;
45         boundary = block;
46     }
47     status = disk_driver.block_transfer(buffer,
48
49             block, operation);
50     {
51         unique_lock<mutex> lock(mx);
52         state = FREE;
53         if (!q[index].first())
54             index = ((i == 0) ? (1) : (0));
55         q[index].notify_one();
56     }
57     return status;
58 }
59
60
61 int Disk::block_get(char* buffer, unsigned block)
62 {
63     int status;
64
65     status = optimize(buffer, block, DISK_READ);
66     return status;
67 }
68
69 int Disk::block_put(char* buffer, unsigned block)
70 {
71     int status;
72
73     status = optimize(buffer, block, DISK_WRITE);
74     return status;
75 }
76
77 Disk disk;
78
79 # Pitanja

```

## Pitanja

- Do čega dovodi pokušaj niti da preuzme znak kada je cirkularni bafer drajvera tastature prazan?
- Šta se desi kada se napuni cirkularni bafer drajvera tastature?
- 

## Šta se desi u obradi prekida diska?

title: Operativni Sistemi - Simulator operativnog sistema 2  
author: Veljko Petrović date: April, 2024 —

## Virtuelna Mašina Simulatora

### Upozorenje

- Termin 'virtuelna mašina' se koristi u više značenja u računarskim naukama
- Značenje kako ga koristimo ovde nije nijedno od onih na koje ste vi navikli
- Znači, u ovom slučaju, više *simulator* hardvera.

### VM

- Razvojna verzija konkurentne biblioteke CppTss sadrži virtuelnu mašinu koja za potrebe ostatka ove biblioteke:
  - emulira kontrolere tastature, ekrana i diska
  - emulira mehanizam prekida
  - podržava okončanje izvršavanja konkurentnog programa
  - podržava rukovanje pojedinim bitima memorijskih lokacija
  - podržava rukovanje numeričkim koprocesorom (Numeric Processor Extension – NPX)
  - podržava rukovanje stekom

### Neophodna zaglavlja

```
1 #include <stdlib.h>
2 #include <strings.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <termios.h>
6 #include <signal.h>
7 #include <sys/time.h>
```

```

8     #include <string.h>
9     #include <stdio.h>
```

## Emulacija mehanizma prekida

- Emulacija mehanizma prekida se zasniva na:
  - uvođenju (emulirane) tabele prekida
  - uvođenju (emuliranog) bita prekida
  - obezbeđenju nezavisnosti (asinhronosti) između prekida i izvršavanja konkurentnog programa
- Potrebe CppTss biblioteke su uzrokovale da (emulirana) tabela prekida sadrži pet elemenata.
- Prvi od njih je namenjen za vektor obradivača hardverskog izuzetka (FLOATING POINT EXCEPTION), a drugi je rezervisan za vektor obradivača prekida sata.
- Preostala tri su predviđena za vektore obradivača prekida tastature, ekrana i diska.

```

1 static const unsigned
2 INTERRUPT_TABLE_VECTOR_COUNT = 5;
3 enum Vector_numbers { FP_EXCEPTION, TIMER, KEYBOARD, DISPLAY, DISK };
4
5 - Svrha (emuliranog) bita prekida je da označi da li je ili nije omogućena obrada prekida. O
6 - Emulacija bita prekida je ostvarena pomoću promenljive `interrupts_enabled` i funkcija `ad_
7 - Promenljiva `interrupts_enabled` sadrži (emulirani) bit prekida. Njena vrednost određuje o
8 - Prva od njih, izmenom (emuliranog) bita prekida, onemogućuje (emulirane) prekide, a druga
9 - Kada operacija `ad__restore_interrupts()` utvrdi da je došlo do odlaganja obrade (emuliran
10 - Nakon toga se registruje da nema više odloženih obrada (emuliranih) prekida.
```

## Emulacija prekida

```

1 bool interrupts_enabled = true;
2 inline static bool ad__disable_interrupts(){
3     bool saved_interrupts_enabled = interrupts_enabled;
4     interrupts_enabled = false;
5     return saved_interrupts_enabled;
6 }
7
8 inline void ad__restore_interrupts(bool saved_interrupts_enabled){
9     if(saved_interrupts_enabled && interrupt.pending) {
10         interrupt.controller_emulator();
11         interrupt.pending = false;
12     }
13     interrupts_enabled = saved_interrupts_enabled;
```

## Emulacija mehanizma prekida

- Nezavisnost (emuliranih) prekida od izvršavanja konkurentnog programa se ostvaruje pomoću mehanizma signala Linux-a.
- Ovaj mehanizam omogućuje da se na pojavu signala reaguje izvršavanjem odabrane funkcije (user level exception handling).
- Ova funkcija se naziva obradivač signala.
- Signali su unapred definisani, a svaki od njih je pridružen jednoj vrsti događaja, kao što je isticanje zadanog vremenskog intervala (SIGVTALRM) ili pojava hardverskog izuzetka (SIGFPE).
- Kada se takav događaj desi mehanizam signala zaustavi izvršavanje programa (u toku koga se desio dotični događaj), radi pokretanja izvršavanja odgovarajućeg obradivača signala.
- Nakon obrade dotičnog signala moguć je nastavak zaustavljenog izvršavanja programa.
- Obradivač signala je funkcija koja opisuje korisničku reakciju na pojavu odabranog signala.
- Funkcija postaje obradivač signala kada se poveže sa odgovarajućim signalom.
- Pojava signala SIGVTALRM nije zavisna od izvršavanja konkurentnog programa.
- Zadatak obradivača signala SIGVTALRM je da pozove operaciju kontrolera i tako izazove obradu nekog od (emuliranih) prekida, a zadatak obradivača signala SIGFPE je da izazove obradu izuzetka.
- Za razliku od obrade izuzetaka, koja se uvek obavlja bez odlaganja, obrada (emuliranih) prekida se obavezno odlaže ako su (emulirani) prekidi onemogućeni.
- Do obrade prethodno onemogućenog (emuliranog) prekida dolazi tek nakon omogućenja (emuliranih) prekida.
- U slučaju konkurentnog programa, pojava signala podstakne mehanizam signala da zaustavi zatečenu aktivnost niti i pokrene odgovarajućeg obradivača signala.
- Ako u sklopu obrade (emuliranog) prekida, koju izazove ovaj obradivač signala, dođe do preključivanja na drugu nit, za-

početa obrada signala će biti završena tek nakon ponovnog preključivanja na prethodno zaustavljenu nit.

- Da bi se u međuvremenu mogli obraditi novi signali, neophodno je da se razna izvršavanja obrađivača signala mogu preklapati.
- Za takve obrađivače signala se kaže da su višeulazni (reentrant).
- Klasa `Linux_signals` opisuje reakciju na Linux signale.
- Njen konstruktor koristi njeno polje `sa` i sistemski poziv `sigaction()` da saopšti da njena operacija `signal_handler()` ima ulogu obrađivača signala `SIGVTALRM` i `SIGFPE`.
- Ovaj konstruktor koristi konstantu `SA_NODEFER` (koju upisuje u polje `sa.sa_flags`) da saopšti da nema odlaganja obrada signala (da je obrađivač signala višeulazni).
- Destruktor klase `Linux_signals` poništava akcije njenog konstruktora.
- Obradivač signala `Linux_signals::signal_handler()` u slučaju signala `SIGVTALRM` pozove emulaciju prekida (`Interrupt::emulation()`), a u slučaju signala `SIGFPE` pozove obrađivača izuzetka (`Interrupt::handler()`) koji opslužuje hardverski izuzetak.

## Emulacija prekida

```
1 class Linux_signals {
2     struct sigaction sa; // struktura za definisanje signala
3     static void signal_handler(int signal);
4     Linux_signals(const Linux_signals &);
5     Linux_signals &operator=(const Linux_signals &);
6
7 public:
8
9     Linux_signals();
10    ~Linux_signals();
11 };
12
13 void Linux_signals::signal_handler(int signal) // override signala
14 {
15     switch (signal) {
16
17     case SIGVTALRM:
18         interrupt.emulation();
19         break;
20     case SIGFPE:
21         interrupt.handler(FP_EXCEPTION);
```

```

22     break;
23 }
24 }
25 }
26
27 Linux_signals::Linux_signals() {
28     sa.sa_handler = signal_handler;
29     sigemptyset(&(sa.sa_mask));    // Ne blokiramo nijedan signal
30     sa.sa_flags = SA_NODEFER;      // obrada bez odlaganja
31     sigaction(SIGVTALRM, &sa, 0); // promena signala za
32
33     sigaction(SIGFPE, &sa, 0);    // SIGVTALRM i SIGFPE
34 }
35
36 Linux_signals::~Linux_signals() {
37     sa.sa_handler = SIG_DFL; // vracanje na default
38     sigaction(SIGVTALRM, &sa, 0);
39     sigaction(SIGFPE, &sa, 0);
40
41 }
42
43 Linux_signals linux_signals;
44
45 const int LINUX_TIMER_INTERVAL = 10;
46
47 class Linux_timer {
48
49     struct itimerval itimer;
50     Linux_timer(const Linux_timer &);
51     Linux_timer &operator=(const Linux_timer &);
52
53 public:
54     Linux_timer();
55     ~Linux_timer();
56
57 };
58
59 Linux_timer::Linux_timer() // usec vreme se meri u mikrosekundama
60 {
61     itimer.it_interval.tv_sec = 0; // tekuća vrednost
62     itimer.it_interval.tv_usec = LINUX_TIMER_INTERVAL * 1000;
63     itimer.it_value.tv_sec = 0; // vrednost kojom se resetuje
64
65     itimer.it_value.tv_usec = LINUX_TIMER_INTERVAL * 1000;
66     setitimer(ITIMER_VIRTUAL, &itimer, 0);
67 }
```

```

68
69 Linux_timer::~Linux_timer() {
70     itimer.it_value.tv_sec = 0;
71     itimer.it_value.tv_usec = 0;
72
73     setitimer(ITIMER_VIRTUAL, &itimer, 0); // user mode timer
74 }
75
76 static Linux_timer linux_timer;
77

```

## Emulacija mehanizma prekida

- Klasa Interrupt:
  - uvodi (emuliranu) tabelu prekida (sadržanu u nizu vector)
  - omogućuje registrovanje odlaganja obrade (emuliranog) prekida (polje pending)
  - reguliše redosled pozivanja obrađivača pojedinih (emuliranih) prekida (polja controller\_turn i timer\_turn)
- (Emulirana) tabela prekida se inicijalizuje tako da njeni elementi sadrže vektor podrazumevajućeg obrađivača prekida: default\_interrupt\_handler().
- Operacija handler() klase Interrupt posreduje u pozivu obrađivača (emuliranog) prekida.
- Operacija emulation() registruje odlaganje obrade prekida ili pokreće emulaciju kontrolera pozivom operacije controller\_emulator().
- U svakoj parnoj emulaciji kontrolera poziva se obrađivač (emuliranog) prekida sata (sa brojem vektora TIMER).
- U svakoj neparnoj emulaciji kontrolera obavlja se, u kružnom redosladu, emulacija samo jednog od kontrolera (display\_controller.output(), keyboard\_controller.input() ili disk\_controller.transfer()).
- Operacija ad\_set\_vector() omogućuje izmenu vektora prekida.

## Emulacija prekida

```

1 class Interrupt {
2     static void (*vector[INTERRUPT_TABLE_VECTOR_COUNT])();
3     bool pending;
4     int controller_turn;
5     bool timer_turn;
6     Interrupt(const Interrupt &);
7     Interrupt &operator=(const Interrupt &);
8
9 public:

```

```

10     Interrupt();
11     inline void handler(unsigned index);
12     inline void emulation();
13     inline void controller_emulator();
14     friend inline void ad_restore_interrupts(bool new_interrupt_status);
15
16     friend inline void ad_set_vector(int index, void (*handler)());
17 };
18
19 void default_interrupt_handler() {}
20
21 void (*Interrupt::vector[INTERRUPT_TABLE_VECTOR_COUNT])() = {
22     default_interrupt_handler};
23
24 Interrupt::Interrupt() : pending(false), controller_turn(0), timer_turn(true) {}
25
26 void Interrupt::handler(unsigned index) { (vector[index])(); }
27
28 void Interrupt::emulation() {
29     if (!interrupts_enabled)
30
31         interrupt.pending = true;
32     else {
33         interrupts_enabled = false;
34         controller_emulator();
35         interrupts_enabled = true;
36     }
37 }
38
39 void Interrupt::controller_emulator() {
40     bool interrupt_emulated;
41     int counter = 0;
42     if (timer_turn) {
43         timer_turn = false;
44         handler(TIMER);
45     } else {
46         timer_turn = true;
47         do {
48             switch (controller_turn) {
49                 case 0:
50                     interrupt_emulated = display_controller.output();
51                     controller_turn = 1;
52
53                     break;
54                 case 1:
```

```

56         interrupt_emulated = keyboard_controller.input();
57         controller_turn = 2;
58         break;
59     case 2:
60         interrupt_emulated = disk_controller.transfer();
61
62         controller_turn = 0;
63         break;
64     }
65 } while (!interrupt_emulated && ++counter < 3);
66 }
67 }
68
69 inline void ad__set_vector(int index, void (*handler)()) {
70     Interrupt::vector[index] = handler;
71 }
72
73 Interrupt interrupt;
74

```

## Emulacija kontrolera tastature

- Klasa Keyboard\_controller opisuje kontroler tastature.
- Njeno polje data\_reg predstavlja registar podataka kontrolera, a njena operacija input() opisuje ponašanje kontrolera tastature.
- Ako postoji znak za preuzimanje, on se preuzima u okviru operacije input() posredstvom odgovarajućeg Linux sistemskog poziva.
- Ujedno se poziva obrađivač prekida tastature posredstvom operacije handler():
  - interrupt.handler(KEYBOARD)
  - klase Interrupt koja emulira tabelu prekida.
- Operacija input() se periodično poziva u toku emulacije kontrolera.

```

1 class Keyboard_controller {
2     char data_reg;
3     bool input();
4     Keyboard_controller(const Keyboard_controller &);
5     Keyboard_controller &operator=(const Keyboard_controller &);
6
7 public:
8
9     Keyboard_controller() {};
10    friend class Interrupt;
11    friend class Keyboard_driver;

```

```

12    };
13
14    bool Keyboard_controller::input() {
15        bool interrupt_emulated = false;
16
17        unsigned read_count;
18        char c;
19        read_count = read(STDIN_FILENO, &c, 1); // sys call
20        if (read_count > 0) {
21            data_reg = c;
22            interrupt.handler(KEYBOARD);
23            interrupt_emulated = true;
24        }
25        return interrupt_emulated;
26    }
27
28    Keyboard_controller keyboard_controller;
29
30

```

## Emulacija kontrolera ekrana

- Klasa Display\_controller opisuje kontroler ekrana.
- Njena polja data\_reg i status\_reg predstavljaju registre podataka i stanja kontrolera, a njena operacija output() opisuje ponašanje kontrolera ekrana.
- Ako postoji znak za prikazivanje, on se prikazuje u okviru operacije output() posredstvom odgovarajućeg Linux sistemskog poziva.
- Ujedno se poziva obrađivač prekida ekrana posredstvom operacije handler() interrupt.handler(DISPLAY) klase Interrupt koja emulira tabelu prekida.
- Operacija output() se periodično poziva u toku emulacije kontrolera.

```

1 enum Display_status { DISPLAY_READY, DISPLAY_BUSY };
2
3 class Display_controller {
4     char data_reg;
5     Display_status status_reg;
6     bool output();
7     Display_controller(const Display_controller &);
8
9     Display_controller &operator=(const Display_controller &);
10
11 public:

```

```

12     Display_controller() : status_reg(DISPLAY_READY){};
13     friend class Interrupt;
14     friend class Display_driver;
15 };
16
17 bool Display_controller::output() {
18     bool interrupt_emulated = false;
19     if (status_reg == DISPLAY_BUSY) {
20         write(STDOUT_FILENO, &data_reg, 1); // sys call
21         status_reg = DISPLAY_READY;
22         interrupt.handler(DISPLAY);
23
24         interrupt_emulated = true;
25     }
26     return interrupt_emulated;
27 }
28
29 Display_controller display_controller;

```

## Emulacija kontrolera tastature i ekrana

- Prethodne dve klase koriste tastaturu i ekran Linux terminala.
- Za potrebe emulacije neophodno je isključiti echo (echo) Linux terminala, prevesti Linux terminal u režim rada bez linijskog editiranja (raw mode) i obezbediti da poziv čitanja znaka sa terminala bude neblokirajući.
- Sve prethodno obezbeđuje klasa Linux\_terminal koji koristi polje ots ove klase da sačuva zatečeni režim rada Linux terminala, a ts polje da zada njegov novi režim rada.
- Prethodno zatečeni režim rada Linux terminala ponovo uspostavlja destruktor klase Linux\_terminal.
- Ovaj destruktor se poziva na kraju aktivnosti procesa i radi provere da li postoje niti za koje nije regularno da kraj njihove aktivnosti nastupi kao posledica kraja aktivnosti procesa kome one pripadaju i radi obaveštenja o prevremenom završetku ovakvih niti.

## Rukovanje terminalom

```

1 class Linux_terminal {
2     struct termios ts;
3     struct termios ots;
4     Linux_terminal(const Linux_terminal &);


```

```

5     Linux_terminal &operator=(const Linux_terminal &);
6
7 public:
8
9     Linux_terminal();
10    ~Linux_terminal();
11 };
12
13 Linux_terminal::Linux_terminal() {
14     tcgetattr(STDIN_FILENO, &ts); // preuzmi parametre terminala
15     ots = ts;
16
17     ts.c_lflag &= ~ECHO; // zabrana eha na ekran
18     ts.c_lflag &= ~ICANON; // flag nekanonskog moda
19     ts.c_cc[VTIME] = 0; // timeout u decisekunda za nekanonski read
20     ts.c_cc[VMIN] = 0; // minimalni broj karaktera za nekanonski read
21     tcsetattr(STDIN_FILENO, TCSANOW, &ts); // postavi parametre term
22 }
23
24
25 Linux_terminal::~Linux_terminal() {
26     if (undetached_threads())
27         write(STDOUT_FILENO, &"\nERROR: DESTROYING UNDETACHED
28             THREADS!\n", 40);
29     else
30         write(STDOUT_FILENO, &"\n", 1);
31     tcsetattr(STDIN_FILENO, TCSANOW, &ots); //povrati parametre
32 }
33
34
35 static Linux_terminal linux_terminal;
36

```

## Emulacija kontrolera diska

- Klasa Disk\_controller opisuje ponašanje kontrolera diska.
- Njena polja operation\_reg, buffer\_reg, block\_reg i status\_reg odgovaraju registrima smera prenosa, bafera, bloka i stanja kontrolera, a njena operacija transfer() opisuje ponašanje kontrolera diska.
- Konstruktor klase Disk\_controller koristi sistemski poziv calloc() radi zauzimanja radne memorije, u kojoj se čuvaju blokovi emuliranog diska.
- Inercija diska se emulira brojanjem poziva operacije transfer().

- Kada broj poziva ove operacije postane jednak procenjenom broju vremenskih jedinica, potrebnom za prenos bloka, tada se obavi prenos bloka u okviru ove operacije i ujedno se pozove obradivač prekida diska posredstvom operacije handler() interrupt.handler(DISK) klase Interrupt koja emulira tabelu prekida.
- Operacija transfer() se periodično poziva u toku emulacije kontrolera.

## Emulacija diska

```

1 enum Disk_operations { DISK_READ, DISK_WRITE };
2 enum Disk_status { DISK_STARTED, DISK_ACTIVE, DISK_STOPPED };
3 const unsigned BLOCK_SIZE = 512;
4 const unsigned DISK_BLOCKS = 1000;
5 typedef char Disk_block[BLOCK_SIZE];
6
7 class Disk_controller {
8
9     Disk_block *disk_space;
10    unsigned accessed_last;
11    unsigned transfer_time;
12    Disk_operations operation_reg;
13    char *buffer_reg;
14    unsigned block_reg;
15    Disk_status status_reg;
16
17    bool transfer();
18    Disk_controller(const Disk_controller &);
19    Disk_controller &operator=(const Disk_controller &);
20
21 public:
22    Disk_controller();
23    ~Disk_controller();
24
25    friend class Interrupt;
26    friend class Disk_driver;
27 };
28
29 Disk_controller::Disk_controller()
30     : accessed_last(0), transfer_time(0), status_reg(DISK_STOPPED) {
31     disk_space = (Disk_block *)calloc(DISK_BLOCKS, sizeof(Disk_block));
32 }
33
34 }
```

```

35 Disk_controller::~Disk_controller() { free(disk_space); }
36
37 const int SECTORS_PER_TRACK = 10;
38 const int TRANSFER_TIME_AND_ROTATIONAL_DELAY = 2;
39
40
41 bool Disk_controller::transfer() {
42     bool interrupt_emulated = false;
43     Disk_block *block_pointer;
44     int block_distance;
45     if (status_reg == DISK_STARTED) { // deo simulacije inercije rotacije diska
46         status_reg = DISK_ACTIVE;
47         block_distance = accessed_last - block_reg;
48
49         if (block_distance < 0)
50             block_distance = -block_distance;
51         transfer_time = TRANSFER_TIME_AND_ROTATIONAL_DELAY;
52         transfer_time += block_distance / SECTORS_PER_TRACK; // vreme rotacije
53         accessed_last = block_reg;
54     }
55     if ((status_reg == DISK_ACTIVE) && (--transfer_time == 0)) {
56
57         block_pointer = disk_space + block_reg;
58         if (operation_reg == DISK_WRITE)
59             bcopy(buffer_reg, block_pointer, BLOCK_SIZE); // kopiraj bajte
60         else
61             bcopy(block_pointer, buffer_reg, BLOCK_SIZE); // kopiraj bajte
62         status_reg = DISK_STOPPED;
63         interrupt.handler(DISK);
64
65         interrupt_emulated = true;
66     }
67     return interrupt_emulated;
68 }
69
70 Disk_controller disk_controller;
71

```

## Okončanje izvršavanja konkurentnog programa

- Izvršavanje konkurentnog programa se okončava sistemskim pozivom `exit()`. To omogućuje funkcija `ad__report_and_finish()`

```

1 inline void ad__report_and_finish(const char* message_string){
2     int length = 0;
3     while(message_string[length] != 0)

```

```

4         length++;
5     write(STDERR_FILENO, message_string, length);
6     write(STDERR_FILENO, "\n", 1);
7     exit(1);
8 }

```

## ASM direktiva

- C/C++ standard podrazumeva postojanje ASM direktive koja omogućava da se u C/C++ kod umetne asemblerski kod date platforme.
- Standard ne definiše u detalje kako tačno ova funkcija treba da radi.
- Mi radimo sa GCC kompjlerom, te stoga koristimo sintaksu koju uvodi GCC.

## Vrste GCC ASM direktive

- U okviru GCC-a, postoje dve varijante ASM direktive:
  - Osnovna (basic) i
  - Proširena (extended)
- Osnovna služi da se samo navedu ASM komande, jedna za drugom, i ništa preko toga.
- Proširena, omogućava integraciju između ASM koda i C koda kroz ulazno/izlazne parametre.

## Osnovna ASM direktiva

```
asm asm-qualifiers ( AssemblerInstructions )
```

- Gde `asm-qualifiers` može biti:
  - `volatile` — kaže kompjleru da ne optimizuje, podrazumevano za osnovni ASM kod
  - `inline` — kaže kompjleru da minimizuje procenjenu veličinu ASM koda

## AssemblerInstructions

- Sastoje se od više linija od kojih je svaka u navodima i svaka se završava sa `\n\t`

```
asm ("movl %eax, %ebx\n\t"
     "movl $56, %esi\n\t"
     "movl %ecx, $label(%edx,%ebx,$4)\n\t"
     "movb %ah, (%ebx)");
```

## **ANSI standardan C**

- Ponekad, ako se želi držati strogog ANSI standarda, uzimajući u obzir nešto neobičnih osobina ASM direktive u GCC-u, može se mesto ‘asm’ koristiti ‘**asm**.’ Nama to može trebati ako želimo da koristimo -std opciju zbog C++11 opcija
- GCC tretira obe stvari absolutno identično.

## **Proširen ASM**

- Osnovni ASM nema jednostavan način da radi sa C kodom. Recimo, ako želite da u njemu modifikujete neku promenljivu iz C koda, to je nemoguće.
- Stoga postoji proširen ASM koji to dozvoljava i čija se upotreba preporučuje.
- Ograničenje u upotrebi ovakve ASM direktive jeste da se to mora činiti iz nekakve funkcije/metode.

## **Sintaksa proširenog ASM-a bez naredbe skoka**

```
asm asm-qualifiers (
    AssemblerTemplate
    : OutputOperands
    : InputOperands
    : Clobbers
)
```

## **Sintaksa proširenog ASM-a sa naredbama skoka**

```
asm asm-qualifiers (
    AssemblerTemplate
    :
    : InputOperands
    : Clobbers
    : GotoLabels
)
```

### **asm-qualifiers**

- **volatile** — isključuje stanovite optmizacije što je neophodno ako naš kod ima pobočne efekte, tj. ako radi nešto preko manipulacije ulaznih u izlazne vrednosti.
- **inline** — kao ranije

- goto — informišemo kompjajler da asm kod može skočiti na neku od labela koje smo specificirali u 'GotoLabels' parametru. Ako je to ikako moguće, ovo valja izbeći.

## AssemblerTemplate

- Ponaša se kao instrukcije kod osnovne ASM direktive uz dve ključne razlike:
  - Kada označavamo registre, umesto da kažemo %eax, recimo, moramo reći %%eax.
  - Možemo da mesto parametara asemblerских instrukcija da stavimo %n gde n nekakav broj i asm će umesto te oznake umetnuti vrednost koju pod tim brojem prosleđujemo iz C koda kroz specifikacije koje se nalaze u OutputOperands i InputOperands

## OutputOperands/InputOperands

- U oba slučaja su zarezima razdvojene liste koje smeju biti prazne.
- U oba slučaja, takođe, elementi liste su oblika "ograničenje" (izraz)
- Gde je ograničenje string sa raznim simbolima koji definišu kako koristimo dati operand, dok je izraz nekakav C izraz (gotovo uvek promenljiva) koju umećemo u naš ASM kod.

## Ograničenja

Karakter	Značenje
r	Ovaj operand ide u neki registar opšte namene, ali ne specificiram koji.
a	Ovaj operand ide u, u zavisnosti od bitaže, %eax, %ax, %al
b	Ovaj operand ide u, u zavisnosti od bitaže, %ebx, %bx, %bl
c	Ovaj operand ide u, u zavisnosti od bitaže, %ecx, %cx, %cl
d	Ovaj operand ide u, u zavisnosti od bitaže, %edx, %dx, %dl
s	Ovaj operand ide u, u zavisnosti od bitaže, %esi, %si
D	Ovaj operand ide u, u zavisnosti od bitaže, %edi, %di

Karakter	Značenje
m	Ovaj operand je isključivo u nekoj memorijskoj lokaciji, bilo kojoj
o	Ovaj opredeljenje je isključivo u memorijskoj lokaciji, i to nekoj koja je takva da dodavanje malog celog broja ravnoj širini tekuće reči u bajtovima i dalje proizvodi validnu adresu.
=	U ovaj operand samo pišemo, ne čitamo, uvek se stavlja za output operande.
broj	Ako stavimo broj kao ograničenje, onda to znači da istu promenljivu koristimo i za ulaz i za izlaz.
i	operand je konstantan celi broj čija se vrednost zna za vreme asembleriranja
E/F	operand je konstantan floating point broj čija se vrednost zna za vreme asembleriranja

## Clobbers

- Ovo je lista stringova u duplim navodnicima razdvojenih zarezima, koja sadrži sve registre koji se menjaju kao pobočni efekat operacija koje izvršavamo.
- To govori kompjajleru da ne očekuje da te vrednosti ostanu iste što utiče na optimizaciju.
- Osim što možemo staviti, npr, "eax" ili "ecx" ovde, može se navesti i "memory" što znači da se modifikuje sadržaj memorije na koji se ne referencira u output sekciji. Kod koji stavlja memory u clobber listu mora biti volatile.

## Rukovanje pojedinim bitima memorijskih lokacija

- Emulacija hardvera je namenjena za platforme zasnovane na i386 (i novijim) procesorima koji podržavaju asembleriske naredbe za:
  - dobijanje indeksa najznačajnijeg postavljenog bita u reči `bsr`
  - postavljanje datog bita reči `bts`
  - za njegovo čišćenje `btr`
- Funkcije `ad__get_index_of_most_significant_set_bit()`, `ad__set_bit()` i `ad__clear_bit()` posreduju u korišćenju ovih asembleriskih naredbi.

## Bitwise asembler

```

1 inline static int
2 ad__get_index_of_most_significant_set_bit(unsigned priority_bits) {
3     int index;           // poziv asm bit scan reverse
4     asm("bsr %1, %0"    ///%1 indeks msb, %0 ulazni biti

```

```

5      : "=r"(index)           // uvek ide prvo izlazni operand
6      : "r"(priority_bits)  // pa ulazni operand
7  );
8
9  return index;
10 }

11
12 inline static unsigned ad__set_bit(unsigned priority_bits,
13                               int index) { // poziva asm bit test and set
14     asm("bts %1, %0"
15           : "=r"(priority_bits)
16             : "r"(index), "0"(priority_bits)
17           );
18     return priority_bits;
19 }
20

21
22 inline static int ad__clear_bit(unsigned priority_bits,
23                               int index) { // poziva asm bit test and reset
24
25     asm("btr %1, %0" : "=r"(priority_bits) : "r"(index), "0"(priority_bits));
26     return priority_bits;
27 }
28

```

## Rukovanje numeričkim koprocesorom

- Preključivanje procesora sa jedne niti na drugu podrazumeva da se sačuva kontekst (sadržaj registara) prve niti i uspostavi kontekst druge niti.
- Kontekst se čuva na steku niti i obuhvata i registre numeričkog koprocesora.
- Klasa `I387_npx` omogućuje pripremu i preuzimanje inicijalnog sadržaja registara numeričkog koprocesora.
- Konstruktor klase `I387_npx` inicijalizuje registre numeričkog koprocesora i smešta njihov inicijalni sadržaj u polje `initial_context` ove klase pomoću asemblerских naredbi `fninit` i `fnsave`.
- Operacija `initial_context_get()` klase `I387_npx` omogućuje preuzimanje inicijalnog sadržaja registara numeričkog koprocesora.

## Rukovanje FPU mehanizmom

```
1 const unsigned i387_SAVE_REGION_SIZE = 0x6c; // velicina FPU steka
2
3 class I387_npx {
4     static char initial_context[i387_SAVE_REGION_SIZE];
5     I387_npx(const I387_npx &);
6     I387_npx &operator=(const I387_npx &);
7
8
9 public:
10    I387_npx();
11    inline void initial_context_get(Stack_item *stack_top);
12 };
13
14 char I387_npx::initial_context[i387_SAVE_REGION_SIZE] = {0};
15
16
17 I387_npx::I387_npx() {
18     asm volatile(           // volatile indikacija kompjleru da ne optimizuje
19         "fninit \n\t"      // inicializacija FPU, status, tag, IP, DP
20         "fnsavel %0 \n\t"  // sacuvaj FPU state u initial_context
21         :
22         : "m"(*initial_context));
23 }
24
25
26 void I387_npx::initial_context_get(Stack_item *stack_top) {
27     for (unsigned i = 0; i < i387_SAVE_REGION_SIZE; i++)
28         ((char *)stack_top)[i] = initial_context[i];
29 }
30
31 static I387_npx i387_npx;
32
```

## Rukovanje stekom

- Za uspeh preključivanja je neophodno da funkcija preključivanja na steku druge niti zatekne ispravan frejm (ako je nit već bila aktivna) ili ako ima spremljen inicijalni kontekst.
- To je obezbeđeno kada se procesor preključuje na nit koja je već bila aktivna. Ali, ako se procesor prvi put preključuje na neku nit, on na njenom steku mora zateći frejm sa ranije pripremljenim njenim inicijalnim kontekstom.
- Podrazumeva se da prvo preključivanje na neku nit dovodi do

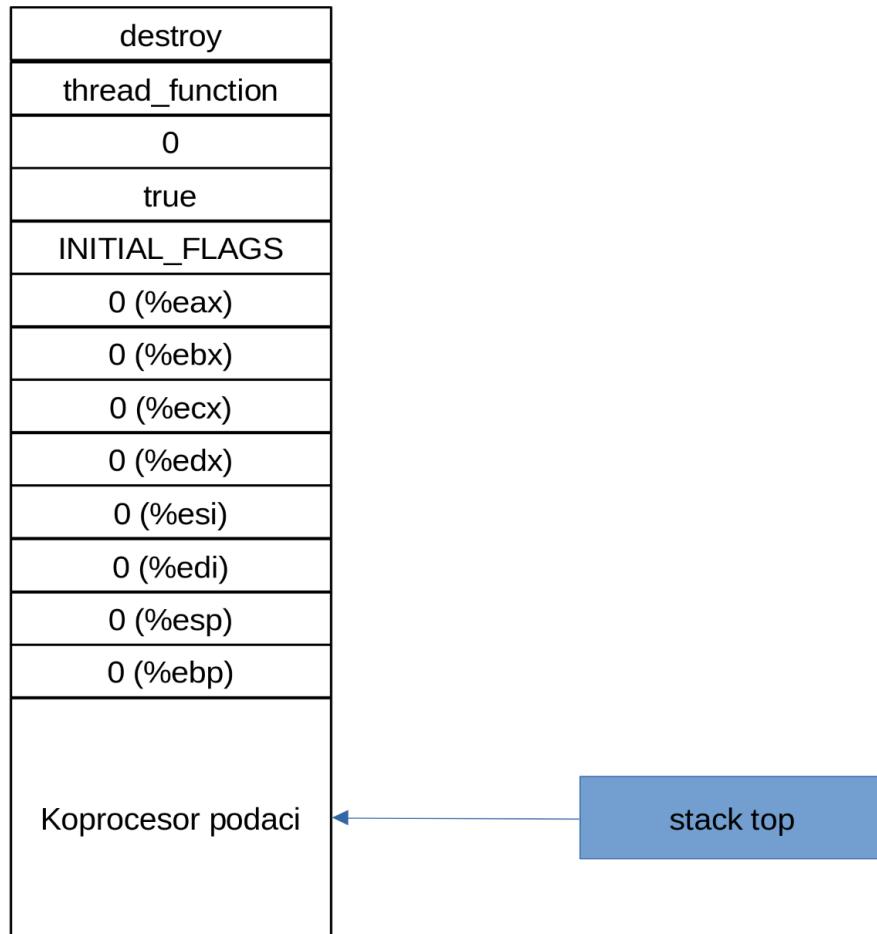
početka izvršavanja funkcije koja opisuje dotičnu nit.

- Da bi izvršavanje ove funkcije bilo moguće, neophodno je na steku niti pripremiti frejm njenog poziva sa odgovarajućom povratnom adresom.
- Kao povratna adresa služi adresa funkcije `destroy()`.
- Do izvršavanja funkcije koja opisuje nit dolazi nakon povratka iz funkcije preključivanja, ako se na steku niti pripremi i frejm poziva funkcije preključivanja u kome se kao povratna adresa koristi adresa funkcije koja opisuje nit.
- Pomenuta dva frejma (frejm poziva funkcije koja opisuje nit i frejm poziva funkcije preključivanja) na steku stvarane niti pripremi funkcija `ad__stack_init`

## Inicijalizacija steka

```
1 const int INITIAL_FLAGS = 0x0200;
2
3 static inline void ad__stack_init(Stack_item **stack_top,
4                                     unsigned thread_function) {
5     *((*(stack_top)) = (Stack_item)destroy;
6     *((*(stack_top)) = (Stack_item)thread_function;
7     *((*(stack_top)) = (Stack_item)0;
8
9     *((*(stack_top)) = (Stack_item) true;
10    *((*(stack_top)) = (Stack_item)INITIAL_FLAGS;
11    *((*(stack_top)) = (Stack_item)0;
12    *((*(stack_top)) = (Stack_item)0;
13    *((*(stack_top)) = (Stack_item)0;
14    *((*(stack_top)) = (Stack_item)0;
15    *((*(stack_top)) = (Stack_item)0;
16
17    *((*(stack_top)) = (Stack_item)0;
18    *((*(stack_top)) = (Stack_item)0;
19    *((*(stack_top)) = (Stack_item)0;
20    *stack_top = (Stack_item *)((size_t)(*stack_top)) - i387_SAVE_REGION_SIZE);
21    i387_npx.initial_context_get(*stack_top);
22 }
23
24
25 extern "C" void ad__stack_swap(Stack_item **const old_stack,
26                                 const Stack_item *new_stack);
```

## Šematski prikaz steka



## Rukovanje stekom

- Funkcija `ad__stack_init` na stek smesti:
  - Adresu funkcije `destroy()` kao povratnu adresu funkcije koja opisuje stvaranu nit
  - Adresu funkcije `thread_function()` kao povratnu adresu funkcije preključivanja
  - Kontekst niti na koju se procesor prvi put preključuje:
  - Pokazivač prethodnog frejma (0) - nema prethodnog frejma
  - Početno stanje emuliranog bita prekida (true)
  - Početno stanje status (flag) registra - INITIAL\_FLAGS
  - Početni sadržaj registara procesora (za registre opšte namene 0, za sadržaj koprocesora rezultat poziva operacije

initial\_context\_get() klase i387\_npx

## Swap

- Funkcija ad\_stack\_swap() ima ulogu funkcije preključivanja. Pošto je ona napisana asemblerским jezikom, radi provere ispravnosti njenih poziva uvedena je njena C deklaracija.

```
1 .text
2 .align 2
3 .globl ad_stack_swap
4
5 ad_stack_swap:
6     pushl %ebp //cuvanje zatecenog pokazivaca stek frejma
7     movl %esp,%ebp //postavljanje novog pokazivaca frejma
8
9     movl 8(%ebp),%edx//adresa vrha steka iz deskriptora niti sa
10        // koje se procesor preključuje
11
12    mov interrupts_enabled,%eax //na stek aktivne niti se smesta
13    push %eax      //zateceno stanje bita prekida
14    pushf          //smestanje flags registra na stek
15    pusha          //smestanje svih registara opste namene na stek
16
17    sub i387_SAVE_REGION_SIZE,%esp
18    fnsavel (%esp) //smestanje sadrzaja reg koprocvara na stek
19
20        mov %esp,(%edx)//adresa vrha steka se smesti u deskriptor
21        //do tada aktivne niti (lokacija u %edx)
22    mov 12(%ebp),%esp//u pokazivac steka se prebaci adresa vrha
23        //steka iz deskriptora novoaktivirane niti
24
25    frstorl (%esp) //sa novog steka se preuzmu novi sadrzaji
26        //registara numerickog koprocvara
27    add i387_SAVE_REGION_SIZE,%esp
28    popa          //sa novog steka se preuzima sadrzaj
29        //registara opste namene
30    popf          //sa novog steka se preuzima sadrzaj
31        //status registara flags
32
33    pop %eax
34    mov %eax,interrupts_enabled //sa novog steka se preuzme
35        //sadrzaj bita prekida i frejm
36        //pointer
37    popl %ebp
38    ret
```

```

39
40 # Izvršilac simulatora
41

```

## Delovi CppTss izvršioca

- Deo CppTss-a koji ima ulogu jezgra operativnog sistema se naziva izvršilac.
- Funkcionalna sličnost CppTss izvršioca sa operativnim sistemom ima za posledicu sličnost njihovih izvedbi.
- Struktura CppTss izvršioca se može predstaviti pomoću istih slojeva kao i struktura operativnog sistema.
- Ključna razlika je da CppTss izvršilac ne sadrži modul za rukovanje datotekama, jer CppTss ne podržava pojam datoteke.

Modul	Elementi u kodu
sistemske niti	thread_wake_up_deamon() thread_destroyer_deamon() thread_zero() klasa Delta
modul za rukovanje procesima	klasa thread klasa Thread_image
modul za rukovanje datotekama	-
modul za rukovanje random memorijom	klasa Memory_fragment

Modul	Elementi u kodu
modul za rukovanje kontrolerima	klasa Timer_driver klasa Exception_driver klasa Driver
modul za rukovanje procesorom	klasa condition_variable klasa unique_lock klasa mutex klasa Kernel klasa Atomic_region klasa Ready_list klasa Descriptor klasa Permit klasa List_link klasa Failure

## Klasa Failure

```

1 enum Failure_codes { MEMORY_SHORTAGE, NOTIFY_OUTSIDE_EXCLUSIVE_REGION };
2 class Failure {

```

```

3  protected:
4      const char *f_name;
5      Failure_codes f_code;
6
7  public:
8
9   Failure(const char *fname, const Failure_codes fcode)
10    : f_name(fname), f_code(fcode){};
11    inline const char *name() const { return f_name; };
12    inline Failure_codes code() const { return f_code; };
13 };
14 Failure failure_memory_shortage("MEMORY SHORTAGE!", MEMORY_SHORTAGE);
15 Failure
16
17     failure_notify_outside_exclusive_region("NOTIFY OUTSIDE EXCLUSIVE REGION!",
18                                         NOTIFY_OUTSIDE_EXCLUSIVE_REGION);
19

```

## Klasa List\_link

```

17 class List_link {
18     List_link *left;
19     List_link *right;
20     List_link(const List_link &);
21     List_link &operator=(const List_link &);
22
23 public:
24
25     List_link() {
26         right = this;
27         left = this;
28     };
29     List_link *left_get() const { return left; };
30     List_link *right_get() const { return right; };
31     void insert(List_link *const link);
32
33     List_link *extract();
34     bool empty() const { return (this == right); };
35     bool not_empty() const { return !empty(); };
36 };
37 void List_link::insert(List_link *const link) {
38     link->left = left;
39     link->right = this;
40
41     left->right = link;

```

```

42     left = link;
43 }
44 List_link *List_link::extract() {
45     List_link *p = right;
46     right->right->left = this;
47     right = right->right;
48
49     return p;
50 }
51
52
53
54 - Klasa List_link omogućuje obrazovanje dvosmerne cirkularne liste.
55 - Nju predstavlja objekat ove klase, koji tada istovremeno služi kao njen početak i kraj.
56 - Takođe se podrazumeva da se oni uvezuju na njen kraj i da se izvezuju sa njenog početka.
57 - Dvosmerna cirkularna lista se obrazuje pomoću polja left i right.
58 - Operacije klase List_link omogućuju preuzimanje vrednosti ovih polja: left_get(), right_get()
59 - uvezivanje novog elementa na kraj liste - insert()
60 - izvezivanje elementa sa početka liste - extract()
61 - proveru da li u listi ima uvezanih elemenata - not_empty()
62 - proveru da li je lista prazna - empty()
63
64
65
66 ! [] (./img/image.png)
67

```

## Klasa Permit

```

1 class Permit {
2     bool free;
3     Permit *previous;
4     List_link admission_list;
5     List_link fulfilled_list;
6
7 public:
8
9     Permit() {
10         free = true;
11         previous = 0;
12     };
13     inline bool not_free() const { return (free == false); };
14     inline void take() { free = false; };
15     inline void release() { free = true; };
16

```

```

17   inline void admission_insert(List_link *link) {
18     admission_list.insert(link);
19   };
20   inline void fulfilled_insert(List_link *link) {
21     fulfilled_list.insert(link);
22   };
23   inline List_link *admission_extract() { return admission_list.extract(); };
24
25   inline List_link *fulfilled_extract() { return fulfilled_list.extract(); };
26   inline bool admission_not_empty() { return admission_list.not_empty(); };
27   inline bool fulfilled_not_empty() { return fulfilled_list.not_empty(); };
28   friend class Descriptor;
29 };
30
31
32
33 - Klasa Permit opisuje rukovanje propusnicama. Polje free klase Permit čuva stanje propusnica
34 - Lista (polje previous) je vezana za nit koja je dobila pomenute propusnice. Propusnice se
35 - Oko polja admission_list klase Permit se obrazuje lista deskriptora niti koje čekaju na pristup
36
37
38
39 - Operacije klase Permit:
40 - not_free() - da li je propusnica zauzeta
41 - take() - zauzimanje propusnice
42 - release() - oslobođanje propusnice
43 - admission_insert(), admission_extract(), admission_not_empty(), fulfilled_insert(), fulfilled_
44
45
46
47 - Pozivi operacija not_free() i take() moraju biti u istom atomskom regionu , inače se može dovesti
48 - do problemata sa propusnicama
49

```

## Klasa Descriptor

```

1 typedef int Stack_item;
2 class Descriptor : private List_link { // nasledjuje se klasa List_link
3   // da bi bilo moguce deskriptore niti uvezivati u liste
4   protected:
5     Stack_item *stack_top; // pokazivac na vrh steka niti
6     int priority;          // prioritet niti
7     Permit *last;          // adresa poslednje dobijene propusnice
8
9     unsigned tag;           // privezak deskriptora niti

```

```

10 public:
11     Descriptor();
12     inline unsigned tag_get() const { return tag; };
13     inline void tag_set(unsigned t) { tag = t; };
14     inline void link_permit(Permit *const permit); // uvezivanje
15     // propusnice u listu propusnica prilikom ulaska niti u kriticnu sekciju
16
17     inline Permit *ulink_permit(); // izuezivanje propusnice iz
18     // liste propusnica niti prilikom njenog izlazka iz kriticne sekcije
19     friend class Ready_list;
20     friend class Kernel;
21     friend class thread;
22 };
23 Descriptor::Descriptor() {
24
25     stack_top = 0;
26     priority = 0;
27     last = 0;
28     tag = 0;
29 }
30 void Descriptor::link_permit(Permit *const permit) {
31     permit->previous = last;
32
33     last = permit;
34 }
35 Permit *Descriptor::ulink_permit() {
36     Permit *permit = last;
37     last = permit->previous;
38     return permit;
39 }
40
41
42
43     - Klasa Descriptor određuje deskriptor niti.
44     - Ona nasleđuje klasu List_link, da bi bilo moguće deskriptore niti uvezivati u liste.
45     - Polje stack_top klase Descriptor sadrži pokazivač (adresu) vrha steka niti.
46     - Prioritet niti je sadržan u polju priority ove klase.
47     - Polje last klase Descriptor sadrži adresu poslednje dobijene propusnice.
48     - Polje tag ove klase je namenjeno za smeštanje priveska deskriptora niti.
49
50
51
52     - Klasa Descriptor nudi operacije za pristup nekim od njenih polja: tag_get(), tag_set(), k
53

```

## Klasa Ready\_list

```
1 const unsigned PRIORITY_NUMBER = 32;
2 enum Priority {
3     TERMINAL = -1,
4     ZERO = 0,
5     PR01,
6     PR02,
7     PR03,
8
9     PR04,
10    PR05,
11    PR06,
12    PR07,
13    PR08,
14    PR09,
15    PR10,
16
17    PR11,
18    PR12,
19    PR13,
20    PR14,
21    PR15,
22    PR16,
23    PR17,
24
25    PR18,
26    PR19,
27    PR20,
28    PR21,
29    PR22,
30    PR23,
31    PR24,
32
33    PR25,
34    PR26,
35    PR27,
36    PR28,
37    PR29,
38    PR30,
39    SYSTEM = 31
40
41 };
42 class Ready_list {
43     unsigned priority_bits;
44     List_link ready[PRIORITY_NUMBER];
```

```

45     Ready_list(const Ready_list &);
46     Ready_list &operator=(const Ready_list &);

47
48
49 public:
50     Ready_list() : priority_bits(0){};
51     int highest() const;
52     void insert(Descriptor *d);
53     Descriptor *extract();
54     bool higher_than(Descriptor *d) const;
55 };
56
57 int Ready_list::highest() const {
58     int n = 0;
59     if (priority_bits != 0)
60         n = ad_get_index_of_most_significant_set_bit(priority_bits);
61     return n;
62 }
63 void Ready_list::insert(Descriptor *d) {
64
65     if (d->priority != TERMINAL) {
66         priority_bits = ad_set_bit(priority_bits, d->priority);
67         ready[d->priority].insert(d);
68     }
69 }
70 Descriptor *Ready_list::extract() {
71     Descriptor *d;
72
73     d = (Descriptor *) (ready[highest()].extract());
74     if (ready[d->priority].empty())
75         priority_bits = ad_clear_bit(priority_bits, d->priority);
76     return d;
77 }
78 bool Ready_list::higher_than(Descriptor *d) const {
79     return (highest() > d->priority);
80 }
81
82 static Ready_list ready;
83
84
85
86     - Klasa Ready_list omogućuje rukovanje spremnim nitima.
87     - Primer takvog rukovanja je brzo pronalaženje najprioritetnije niti među spremnim nitima,
88     - Radi toga, svakom od prioriteta niti se dodeljuje posebna lista spremnih niti i podrazumeva
89     - Takođe se podrazumeva da se deskriptor spremne niti uvek izvezuje sa početka odabrane liste
90     - Na ovaj način spremne niti istog prioriteta se uvek aktiviraju u redosledu u kome su postavljene

```

```

91
92
93
94 - Sve liste spremnih niti zajedno formiraju multi-listu (Ready::ready).
95 - Rukovanje ovom multi-listom obuhvata:
96 - Dobijanje prioriteta najprioritetnije neprazne liste spremnih niti: Ready::highest()
97 - Uvezivanje deskriptora niti na kraj odgovarajuće liste spremnih niti: Ready::insert()
98 - izvezivanje deskriptora niti sa početka najprioritetnije neprazne liste spremnih niti: Ready::remove()
99 - poređenje prioriteta najprioritetnije neprazne liste spremnih niti sa prioritetom zadane niti
100
101
102
103 - Pošto je multi-lista niz listi spremnih niti, deskriptor spremne niti se uvezuje na kraj liste.
104 - Međutim, za operaciju izvezivanja deskriptora iz najprioritetnije neprazne liste spremnih niti
105 - Brzo pronalaženje najprioritetnije neprazne liste spremnih niti se ostvaruje tako da se svaki
106
107
108
109 - Ove bite sadrži Ready::priority_bits, tako da se na značajnijim pozicijama nalaze biti prioriteta.
110 - Na najmanje značajnoj poziciji je bit nulte liste spremnih niti, sa prioritetom 0.
111 - U njoj se nalazi posebna nulta nit, sa prioritetom 0.
112 - Ona angažuje procesor kada nema drugih spremnih niti.
113 - Kada je nulta nit u stanju "spremna", tada je njen deskriptor uvezan u nultu listu spremnih niti.
114 - Ona u to stanje prelazi kada ne postoji neka druga nit koja može da zaposli procesor.
115
116
117
118 - Najniži prioritet spremnih niti 0 (ZERO) je rezervisan za nultu nit, a najviši prioritet 31 je rezervisan za poslednju.
119 - Između se nalaze prioriteti korisničkih niti (PR01, PR02, ..., PR30).
120 - Za uništavane niti, koje čekaju da budu uništene i koje više ne mogu biti spremne (a ni akcije)
121

```

## Klasa Atomic\_region

```

1 class Atomic_region {
2     bool flags;
3     Atomic_region(const Atomic_region &);
4     Atomic_region &operator=(const Atomic_region &);
5
6 public:
7     Atomic_region();
8
9     ~Atomic_region();
10 };
11 Atomic_region::Atomic_region() { flags = ad__disable_interrupts(); }

```

```
12     Atomic_region::~Atomic_region() { ad__restore_interrupts(flags); }
13
```

## Klasa Kernel

- Klasa Kernel omogućuje rukovanje procesorom. Rukovanje procesorom se svodi na preključivanje procesora sa jedne niti na drugu.
- Za preključivanje je neophodno imati adresu deskriptora aktivne niti sa koje se procesor preključuje (active), adresu deskriptora niti na koju se procesor preključuje (pretender), kao i adresu deskriptora niti sa koje se procesor preključio (former).
- Operaciju preključivanja poziva privatna operacija switch\_to(), koja postavlja pokazivač deskriptora aktivne niti active i pokazivač deskriptora niti sa koje se procesor preključio former.
- Preključivanje je nužno vezano za raspoređivanje (scheduling), odnosno za izbor niti na koju se procesor preključuje.
- Ciljevi raspoređivanja kod CppTss izvršioca su da uvek bude aktivna najprioritetnija spremna nit i da se ravnomerno deli vreme procesora između spremnih niti istog prioriteta. Do raspoređivanja dolazi:
  - kada se pojavi spremna nit sa višim prioritetom od aktivne niti (schedule())
  - na kraju kvantuma (periodic\_schedule())
- Klasa Kernel nasleđuje klasu Deskriptor da bi jedini objekat klase Kernel reprezentovao deskriptor main() niti.
- Konstruktor ove klase proglašava aktivnom main() niti. Pošto main() nit koristi stek konkurentnog programa kao svoj stek, za nju nije potrebno zauzeti stek.
- U nadležnosti klase Kernel nije samo preključivanje, nego i podrška viših slojeva iz hijerarhijske strukture CppTss izvršioca.
- Funkcije:
  - make\_ready() - omogućava aktivnost niti
  - expect() - omogućuje očekivanje dešavanja događaja
  - signal() - omogućuje objavu dešavanja događaja
  - exclusive\_in() - za ulazak u isključivi region
  - exclusive\_out() - za izlazak iz isključivog regiona

- wait() - očekivanje ispunjenja uslova
- notify\_one() - objava ispunjenja uslova
- U operacijama deljene promenljive kernel se koriste atomski regioni radi zaštite njene konzistentnosti, kao i konzistentnosti argumenta iz poziva ovih operacija.

```

1  class Kernel : public Descriptor {
2      Descriptor *active;
3      Descriptor *pretender;
4      Descriptor *former;
5      Kernel(const Kernel &);
6      Kernel &operator=(const Kernel &);
7      inline void switch_to(Descriptor *const d);
8
9      inline void schedule();
10     inline void periodic_schedule();
11
12 public:
13     Kernel() : active(0), pretender(0), former(0) {
14         priority = PR15;
15         active = this;
16     }
17     inline void make_ready(Descriptor *const d);
18     inline void expect(List_link *const waiting_list);
19     inline void signal(List_link *const waiting_list);
20     inline void exclusive_in(Permit *const permit);
21     inline void wait(const unsigned t, List_link *const waiting_list);
22     inline void notify_one(List_link *const waiting_list);
23
24     inline void exclusive_out();
25     inline Descriptor *active_get() const { return active; };
26     friend void yield();
27     friend class Timer_driver;
28 };
29 void Kernel::switch_to(Descriptor *const d) {
30     former = active;
31
32     active = d;
33     ad__stack_swap(&(former->stack_top), active->stack_top);
34 }
35 void Kernel::schedule() {
36     if (ready.higher_than(active)) {
37         ready.insert(active);
38         pretender = ready.extract();
39

```

```

40         switch_to(pretender);
41     }
42   }
43 }
44 void Kernel::periodic_schedule() {
45   ready.insert(active);
46   pretender = ready.extract();
47   if (active != pretender)
48     switch_to(pretender);
49   }
50 }
51 void Kernel::make_ready(Descriptor *const d) {
52   Atomic_region ar;
53   ready.insert(d);
54 }
55 void Kernel::expect(List_link *const waiting_list) {
56   waiting_list->insert(active);
57   pretender = ready.extract();
58   switch_to(pretender);
59 }
60 }
61 void Kernel::signal(List_link *const waiting_list) {
62   if (waiting_list->not_empty()) {
63     pretender = (Descriptor *)waiting_list->extract();
64     ready.insert(pretender);
65     schedule();
66   }
67 }
68 }
69 void Kernel::exclusive_in(Permit *const permit) {
70   Atomic_region ar;
71   if (permit->not_free()) {
72     permit->admission_insert(active);
73     pretender = ready.extract();
74     switch_to(pretender);
75   } else {
76     permit->take();
77     active->link_permit(permit);
78   }
79 }
80 }
81 }
82 void Kernel::wait(const unsigned t, List_link *const waiting_list) {
83   Atomic_region ar;
84   Permit *permit = active->ulink_permit();
85   active->tag = t;

```

```

86     if (permit->fulfilled_not_empty()) {
87         pretender = (Descriptor *)permit->fulfilled_extract();
88
89         pretender->link_permit(permit);
90         ready.insert(pretender);
91     } else if (permit->admission_not_empty()) {
92         pretender = (Descriptor *)permit->admission_extract();
93         pretender->link_permit(permit);
94         ready.insert(pretender);
95     } else
96
97         permit->release();
98
99     waiting_list->insert(active);
100    pretender = ready.extract();
101    switch_to(pretender);
102 }
103 void Kernel::notify_one(List_link *const waiting_list) {
104     Permit *permit = active->last;
105
106     if (permit == 0)
107         throw &failure_notify_outside_exclusive_region;
108     Atomic_region ar;
109     if (waiting_list->not_empty()) {
110         pretender = (Descriptor *)waiting_list->extract();
111         permit->fulfilled_insert(pretender);
112     }
113 }
114 void Kernel::exclusive_out() {
115     Atomic_region ar;
116     Permit *permit = active->ulink_permit();
117     if (permit->fulfilled_not_empty()) {
118         pretender = (Descriptor *)permit->fulfilled_extract();
119         pretender->link_permit(permit);
120
121         ready.insert(pretender);
122     } else if (permit->admission_not_empty()) {
123         pretender = (Descriptor *)permit->admission_extract();
124         pretender->link_permit(permit);
125         ready.insert(pretender);
126     } else
127         permit->release();
128
129     schedule();
130 }
131 static Kernel kernel;

```

```

132 void yield() {
133     Atomic_region set_up;
134     kernel.periodic_schedule();
135 }
136

```

## Klasa mutex

```

120 class mutex : private Permit {
121     mutex(const mutex &);
122     mutex &operator=(const mutex &);

123 public:
124     mutex() {};
125     void lock();
126
127     void unlock();
128 };
129 void mutex::lock() { kernel.exclusive_in(this); }
130 void mutex::unlock() { kernel.exclusive_out(); }
131
132

```

## Klasa unique\_lock

```

131 template <class MUXEX> class unique_lock {
132     unique_lock(const unique_lock &);
133     unique_lock &operator=(const unique_lock &);

134 public:
135     unique_lock(MUXEX &mx);
136     ~unique_lock();

137 };
138
139 template <class MUXEX> unique_lock<MUXEX>::unique_lock(MUXEX &mx) {
140     kernel.exclusive_in((Permit *)&mx);
141 }
142
143 template <class MUXEX> unique_lock<MUXEX>::~unique_lock() {
144     kernel.exclusive_out();
145 }
146

```

## Klasa condition\_variable

```

145 class condition_variable {
146     List_link list_head;

```

```

147     List_link *position;
148
149 public:
150     condition_variable() { position = &list_head; };
151     void wait(unique_lock<mutex> &lock, unsigned t = 0);
152
153     void notify_one();
154     bool first(unsigned *t = 0);
155     bool last();
156     bool next(unsigned *t = 0);
157     bool attach_tag(unsigned t);
158 };
159 void condition_variable::wait(unique_lock<mutex> &lock, unsigned t) {
160
161     kernel.wait(t, position);
162     position = &list_head;
163 }
164 void condition_variable::notify_one() {
165     kernel.notify_one(&list_head);
166     position = &list_head;
167 }
168
169 bool condition_variable::first(unsigned *t) {
170     bool r = false;
171     if (list_head.not_empty()) {
172         position = list_head.right_get();
173         if (t != 0)
174             *t = ((Descriptor *)position)->tag_get();
175         r = true;
176     }
177     return (r);
178 }
179 bool condition_variable::last() {
180     bool r = false;
181     if (list_head.not_empty()) {
182         position = &list_head;
183
184         r = true;
185     }
186     return (r);
187 }
188 bool condition_variable::next(unsigned *t) {
189     bool r = false;
190     if (position != &list_head) {
191
192

```

```

193     position = position->right_get();
194     if (position != &list_head) {
195         if (t != 0)
196             *t = ((Descriptor *)position)->tag_get();
197         r = true;
198     }
199 }
200
201     return (r);
202 }
203 bool condition_variable::attach_tag(unsigned t) {
204     bool r = false;
205     if (position != &list_head) {
206         ((Descriptor *)position)->tag_set(t);
207         r = true;
208     }
209     return (r);
210 }
211 }
212

```

## Klasa Driver

- Klasa Driver omogućuje smeštanje adrese obradivača prekida u tabelu prekida: Driver::start\_interrupt\_handling().
- Pored toga, ova klasa uvodi definiciju klase Event koja omogućuje zaustavljanje aktivnosti niti do dešavanja događaja i objavu dešavanja događaja.

```

1 class Driver {
2 protected:
3     void start_interrupt_handling(Vector_numbers vector_number,
4                                     void (*handler)());
5     class Event {
6         List_link list_head;
7
8
9     public:
10     void expect();
11     void signal();
12 };
13 };
14 void Driver::start_interrupt_handling(Vector_numbers vector_number,
15                                         void (*handler)()) {
16
17     Atomic_region ar;

```

```

18     ad__set_vector(vector_number, handler);
19 }
20 void Driver::Event::expect() { kernel.expect(&list_head); }
21 class Driver {
22 protected:
23     void start_interrupt_handling(Vector_numbers vector_number,
24                                     void (*handler)());
25     class Event {
26         List_link list_head;
27     public:
28         void expect();
29         void signal();
30     };
31 };
32 void Driver::start_interrupt_handling(Vector_numbers vector_number,
33                                       void (*handler)()) {
34     Atomic_region ar;
35     ad__set_vector(vector_number, handler);
36 }
37 void Driver::Event::expect() { kernel.expect(&list_head); }
38 void Driver::Event::signal() { kernel.signal(&list_head); }
39
40
41
42
43

```

## Klase Exception\_driver i Timer\_driver

- Iz klase Driver su izvedene klase Exception\_driver i Timer\_driver.
- Prva od njih omogućuje reakciju na pojavu hardverskih izuzetaka, radi izazivanja prevremenog kraja konkurentnog programa.
- Druga od ovih klasa omogućuje rukovanje vremenom.
- Klasa Exception\_driver uvodi operaciju interrupt\_handler(). Ova operacija zaustavlja izvršavanje konkurentnog programa.
- Rukovanje vremenom obuhvata:
  - brojanje otkucaja sata, radi praćenja proticanja sistemskog vremena
  - odbrojavanje otkucaja sata preostalih do kraja kvantuma aktivne niti
  - odbrojavanja otkucaja sata preostalih do budjenja uspavane niti

- Kada broj otkucaja, preostalih do isticanja kvantuma aktivne niti, padne na nulu, potrebno je pokrenuti periodično rasporedivanje.
- Takođe, kada broj otkucaja, preostalih do buđenja uspavane niti, padne na nulu, potrebno je probuditi sve niti za koje je nastupio trenutak buđenja.
- Svi prethodno pobrojani poslovi se nalaze u nadležnosti operacije interrupt\_handler() koju uvodi klasa Timer\_driver.
- Polje current\_ticks ove klase sadrži sistemsko vreme, polje countdown sadrži broj otkucaja do buđenja, a polje rest broj otkucaja do isticanja kvantuma.
- Funkcija now() vraća sadržaj polja current\_ticks, odnosno vraća sistemsko vreme.

```

1 const unsigned long QUANTUM = 2;
2 class Exception_driver : public Driver {
3     static void interrupt_handler();
4
5 public:
6     Exception_driver() {
7         start_interrupt_handling(FP_EXCEPTION, interrupt_handler);
8
9     };
10    };
11 void Exception_driver::interrupt_handler() {
12     ad__report_and_finish("\nHARDWARE EXCEPTION!\n");
13 }
14 static Exception_driver exception_driver;
15 class Timer_driver : public Driver {
16
17     static unsigned long current_ticks;
18     static unsigned long countdown;
19     static unsigned long rest;
20     static unsigned long quantum;
21     static Event alarm;
22     static void interrupt_handler();
23
24
25 public:
26     Timer_driver() { start_interrupt_handling(TIMER, interrupt_handler); };
27     friend unsigned long now();
28     friend class Delta;
29 };
30 unsigned long Timer_driver::current_ticks = 0;

```

```

31 unsigned long Timer_driver::countdown = 0;
32
33 unsigned long Timer_driver::rest = QUANTUM;
34 Timer_driver::Event Timer_driver::alarm;
35 void Timer_driver::interrupt_handler() {
36     current_ticks++;
37     if ((--rest) == 0)
38         rest = quantum;
39     if ((countdown > 0) && ((--countdown) == 0))
40
41         alarm.signal();
42     else if (rest == quantum)
43         kernel.periodic_schedule();
44 }
45 static Timer_driver timer_driver;
46 unsigned long now() {
47     Atomic_region ar;
48
49     return timer_driver.current_ticks;
50 }
51

```

## Klasa Memory\_fragment

- Klasa Memory\_fragment omogućuje rukovanje slobodnom radnom memorijom. Slobodnu radnu memoriju obrazuje celi broj jedinica sastavljenih od UNIT bajta.
- Rukovanje slobodnom radnom memorijom podrazumeva da se uvek zauzima, odnosno da se uvek oslobađa celi broj ovih jedinica.
- Zauzimanja ovakvih zona slobodne radne memorije, odnosno njihova oslobađanja uzrokuju iscepkanost slobodne radne memorije u odsečke.
- Odsečci se zato uvezuju u jednosmernu listu, uređenu u rastućem redosledu njihovih početnih adresa.
- Radi toga, početak svakog odsečka sadrži svoju veličinu, izraženu u pomenutim jedinicama od po UNIT bajta, i pokazivač narednog odsečka.
- Veličinu odsečka i pokazivač narednog odsečka sadrže polja size i next klase Memory\_fragment.
- Konstruktor klase Memory\_fragment opisuje obrazovanje liste odsečaka slobodne radne memorije, sastavljene od stalnog

odsečka čija veličina je 0 i od odsečka koji obuhvata raspoloživu slobodnu radnu memoriju.

- Stalnom (prvom) odsečku odgovara objekt memory klase Memory\_fragment, koji je jedini objekt ove klase. Dodavanju drugog odsečka prethodi provera da li je obezbeđeno dovoljno radne memorije za potrebe konkurentnog programa.
- Ako nije, izvršavanje konkurentnog programa se završava uz poruku INITIAL MEMORY SHORTAGE.
- Pošto je UNIT jednak veličini stranice, uvek se zauzima toliko radne memorije da u nju može da stane traženi broj stranica, a da početak raspoložive slobodne radne memorije bude postavljen na početak prve stranice.
- Zauzimanje slobodne radne memorije omogućuje operacija take() klase Memory\_fragment.
- Zauzima se jedna jedinica od UNIT bajta više nego što je traženo.
- Ona prethodi preostalim zauzetim jedinicama memorije i sadrži ukupnu veličinu zauzete memorije. Ova veličina se koristi prilikom kasnijeg oslobađanja zauzete memorije.
- Zauzimanju prethodi pretraživanje liste odsečaka, radi pronalaženja prvog dovoljno velikog odsečka.
- Pretraživanje uvek počinje od stalnog odsečka. Ako se pronađe dovoljno velik odsečak, traženi bajti se zauzimaju s njegovog kraja. Ako pronađeni odsečak obuhvata baš traženi broj bajta, tada se on isključuje iz liste i zauzimaju se svi njegovi bajti.
- Oslobađanje prethodno zauzete radne memorije omogućuje operacija free() klase Memory\_fragment.
- Za oslobađanje je neophodno u listi odsečaka pronaći odsečak iza koga će se oslobađani odsečak uvezati u ovu listu.
- Pre uvezivanja proverava se da li oslobađani odsečak može da se spoji u jedan odsečak sa svojim prethodnikom i sa svojim sledbenikom.
- Odsečak se uvezuje u pomenutu listu samo ako ovo spajanje nije moguće.
- Prethodno opisane operacije klase Memory\_fragment su namenjene za zauzimanje i oslobađanje radne memorije prilikom stvaranja i uništavanja objekata pojedinih klasa.
- Da bi se njihova namena ostvarila, neophodno je da ove operacije pozivaju globalni operatori new() i delete().

- Ali, tada razne niti mogu da pozivaju operacije klase Memory\_fragment posredstvom prethodna dva operatora i da tako ugroze konzistentnost liste odsečaka.
- Da bi se to sprečilo, ova klasa nasleđuje klasu mutex i tako omogućuje zaključavanje i otključavanje njenog jedinog objekta memory. To je obezbeđeno u definicijama funkcija operator new() i operator delete(), radi ostvarenja međusobne isključivosti različitih pristupanja listi odsečaka.

```

1 const size_t UNIT = PAGE_SIZE;
2 class Memory_fragment : public mutex {
3     size_t size;
4     Memory_fragment *next;
5     Memory_fragment(const Memory_fragment &);
6     Memory_fragment &operator=(const Memory_fragment &);
7
8
9 public:
10    Memory_fragment();
11    void *take(size_t size);
12    void free(void *address);
13 };
14 Memory_fragment::Memory_fragment() : size(0), next(this) {
15     size_t free_memory = 2000 * UNIT;
16
17     size_t beginning = (size_t)malloc(free_memory + UNIT - 1);
18     if (beginning == 0)
19         ad_report_and_finish("INITIAL MEMORY SHORTAGE");
20     else {
21         beginning = (beginning + UNIT - 1) & ~(UNIT - 1);
22         next = (Memory_fragment *)beginning;
23         next->size = free_memory;
24
25         next->next = this;
26     }
27 }
28 void *Memory_fragment::take(size_t size) {
29     size += 2 * UNIT - 1; // 1 UNIT za broj zauzetih i drugi ako
30     size -= size % UNIT; // size nije ceo broj unita
31     Memory_fragment *m = 0;
32
33     Memory_fragment *p = this;
34     while (p->next != this) {
35         if ((p->next->size) < size)
36             p = p->next;
37         else if (p->next->size == size) {

```

```

38         m = p->next;
39         p->next = p->next->next;
40
41         break;
42     } else {
43         p->next->size -= size;
44         m = (Memory_fragment *)((size_t)(p->next) + (p->next->size));
45         break;
46     }
47 }
48
49 if (m == 0)
50     throw &failure_memory_shortage;
51 m->size = size;
52 return (void *)((size_t)m + UNIT);
53 }
54 void Memory_fragment::free(void *address) {
55     if (address != 0) {
56
57         Memory_fragment *a = (Memory_fragment *)((size_t)address - UNIT);
58         Memory_fragment *p = this;
59         while (p->next != this)
60             if (a > p->next)
61                 p = p->next;
62             else
63                 break;
64
65         if (((size_t)p) + (p->size)) == ((size_t)a)) {
66             p->size += a->size;
67             if (((size_t)p) + (p->size)) == ((size_t)(p->next))) {
68                 p->size += p->next->size;
69                 p->next = p->next->next;
70             }
71         } else if (((size_t)a) + (a->size)) == ((size_t)(p->next))) {
72
73             a->size += p->next->size;
74             a->next = p->next->next;
75             p->next = a;
76         } else {
77             a->next = p->next;
78             p->next = a;
79         }
80     }
81 }
82 }
83 static Memory_fragment memory;

```

```

84 void *operator new(size_t size) {
85     void *memory_block;
86     memory.lock();
87     try {
88
89         memory_block = memory.take(size);
90     } catch (...) {
91         memory.unlock();
92         throw;
93     }
94     memory.unlock();
95     return memory_block;
96 }
97 void operator delete(void *address) {
98     memory.lock();
99     memory.free(address);
100    memory.unlock();
101 }
102 }
103

```

## Klase Thread\_image i thread

- Rukovanje nitima omogućuju klase Thread\_image i thread, kao i definicije funkcija thread\_destroyer\_deamon(), destroy() i undetached\_threads().
- Klasa Thread\_image opisuje sliku niti, sastavljenu od:
  - deskriptora niti
  - uslova (ended) koji omogućuje čekanje završetka aktivnosti niti
  - oznake da regularan kraj aktivnosti niti može da nastupi kao posledica kraja aktivnosti procesa kome dotična nit pripada (detached)
  - steka niti
- Klasa thread omogućuje:
  - međusobnu isključivost svojih operacija (mx)
  - brojanje niti za koje nije regularno da kraj njihove aktivnosti nastupi kao posledica kraja aktivnosti procesa kome dotične niti pripadaju (undetached\_threads\_number)
  - uništavanje niti (termination, terminating)
  - pristup slici niti (ti)

- Konstruktor klase `thread` omogućuje kreiranje niti. U toku kreiranja niti pripremi se njen stek za preključivanje, da bi automatski započelo izvršavanje funkcije koja opisuje ponašanje niti nakon prvog preključivanja na nit.
- Završetak aktivnosti niti se otkriva u okviru operacija `join()` i `detach()` na osnovu završnog prioriteta niti (`TERMINAL`).
- Na završetku aktivnosti niti, u toku izvršavanja funkcije `destroy()`, omogućuje se nastavak aktivnosti niti koja čeka dotični završetak i oslobođanje prostora koga zauzima slika niti, što je u nadležnosti sistemske niti `thread_destroyer_deamon()`.
- Funkcija `undetached_threads()` omogućuje proveru da li postoje niti za koje nije regularno da kraj njihove aktivnosti nastupi kao posledica kraja aktivnosti procesa kome dotične niti pripadaju, da bi se na kraju aktivnosti procesa ukazalo na prevremeni završetak ovakvih niti.

```

1 const unsigned DEFAULT_STACK_SIZE = 4096;
2 class Thread_image : public Descriptor {
3     condition_variable ended;
4     bool detached;
5     Stack_item stack[DEFAULT_STACK_SIZE];
6     Thread_image(const Thread_image &);
7     Thread_image &operator=(const Thread_image &);
8
9
10 public:
11     Thread_image(void (*thread_function)(), Priority p);
12     friend class thread;
13     friend void destroy();
14 };
15 Thread_image::Thread_image(void (*thread_function)(), Priority p)
16
17     : detached(false) {
18     stack_top = &(stack[DEFAULT_STACK_SIZE]);
19     ad__stack_init(&stack_top, (unsigned)thread_function);
20     priority = p;
21 }
22 class thread {
23     static mutex mx;
24
25     static unsigned undetached_threads_number;
26     static condition_variable termination;
27     static Thread_image *terminating;
28     Thread_image *ti;
29     thread(const thread &);


```

```

30     thread &operator=(const thread &);
31
32
33 public:
34     thread(void (*thread_function)(), Priority p = PR15);
35     void join();
36     void detach();
37     friend void thread_destroyer_deamon();
38     friend void destroy();
39     friend bool undetached_threads();
40
41 };
42 mutex thread::mx;
43 unsigned thread::undetached_threads_number = 0;
44 condition_variable thread::termination;
45 Thread_image *thread::terminating;
46 thread::thread(void (*thread_function)(), Priority p) {
47     ti = new Thread_image(thread_function, p);
48
49     unique_lock<mutex> lock(mx);
50     undetached_threads_number++;
51     kernel.make_ready(ti);
52 }
53 void thread::join() {
54     unique_lock<mutex> lock(mx);
55     if (ti->priority != TERMINAL)
56
57         ti->ended.wait(lock);
58 }
59 void thread::detach() {
60     unique_lock<mutex> lock(mx);
61     if ((ti->priority != TERMINAL) && (!ti->detached)) {
62         ti->detached = true;
63         undetached_threads_number--;
64     }
65 }
66
67 void thread_destroyer_deamon() {
68     for (;;) {
69         unique_lock<mutex> lock(thread::mx);
70         thread::termination.wait(lock);
71         delete thread::terminating;
72     }
73 }
74
75 void destroy() {

```

```

76     unique_lock<mutex> lock(thread::mx);
77     thread::terminating = (Thread_image *)kernel.active_get();
78     while (thread::terminating->ended.last())
79         thread::terminating->ended.notify_one();
80
81     if (!thread::terminating->detached)
82         thread::undetached_threads_number--;
83     thread::terminating->priority = TERMINAL;
84     thread::termination.notify_one();
85 }
86 bool undetached_threads() { return (thread::undetached_threads_number > 0); }
87

```

## Klasa Delta

- Klasa Delta i funkcije `thread_wake_up_daemon()`, `sleep_for()` i `sleep_until()` zajedno omogućuju uspavljivanje i buđenje niti, a funkcija `thread_zero()` opisuje aktivnost nulte (sistemske) niti.
- `sleep_for()` omogućuje uspavljivanje aktivne niti dok ne protekne zadani broj otkucaja sata
- `sleep_until()` omogućuje uspavljivanje aktivne niti dok ne nastupi zadani trenutak sistemskog vremena.
- Oko polja list klase Delta se formira lista deskriptora uspavanih niti.
- Da se za svaku uspavanu nit ne bi proveravalo, nakon svakog otkucaja, da li je nastupilo vreme njenog buđenja, deskriptori uspavanih niti se uvezuju u listu u hronološkom redosledu buđenja niti.
- Svakom od ovih deskriptora je dodeljen privezak koji pokazuje relativno vreme buđenja (relativni broj otkucaja do buđenja) u odnosu na prethodnika u listi.
- Ovakva lista se zove delta lista.
- Zahvaljujući delta listi, nakon svakog otkucaja potrebno je proveriti da li je nastupio trenutak buđenja samo za nit koja se najranije budi, odnosno samo za prvi deskriptor iz delta liste.
- Pošto može da bude više niti, čije buđenje je vezano za isti trenutak, unapred nije poznato koliko niti treba probuditi nakon otkucaja sata.
- Operaciju `awake()` klase Delta poziva sistemska nit `Wake_up_daemon()`. Vreme njenog buđenja je uvek jednako najranijem vremenu buđenja korisničkih niti.

- Nakon buđenja, sistemska nit budi sve korisničke niti sa početka delta liste, za koje je nastupio trenutak buđenja.
- Čekanje buđenja omogućuje poziv operacije Timer\_driver::alarm.expect().
- Dužinu čekanja određuje vrednost lokalne promenljive tag, kada ima uspavanih korisničkih niti (na čije prisustvo ukazuje vrednost lokalne promenljive sleeping).
- Dužina čekanja se skraćuje za vrednost lokalne promenljive passed\_ticks, koja registruje vreme proteklo na buđenju korisničkih niti.
- Operaciju sleep() klase Delta poziva, posredstvom funkcije sleep\_for(), korisnička nit, da bi se njen deskriptor uključio u delta listu, a njena aktivnost privremeno zaustavila.
- Konstruktor klase Delta omogućuje kreiranje sistemskih niti korišćenjem bezimenih (privremenih) objekata klase thread.
- Konzistentnost delta liste štite isključivi regioni u telima operacija awake() i sleep() klase Delta.

## Datoteke

### Svojstva datoteka

- Svaka datoteka poseduje ime koje bira korisnik. Poželjno je da ime datoteke ukazuje na:
  - njen konkretan sadržaj
  - na vrstu njenog sadržaja (radi klasifikacije datoteka po njihovom sadržaju)
  - Zato su imena datoteka dvodelna npr:
  - godina1.txt
- Može da predstavlja ime datoteke, koja sadrži podatke o studen-tima prve godine studija. Na to ukazuje prvi deo imena godina1, dok drugi deo imena txt ove datoteke govori da je datoteka tek-stualna, odnosno da sadrži samo vidljive ASCII znakove.
- Rukovanje datotekom obuhvata ne samo rukovanje njenim sadržajem, nego i rukovanje njenim imenom. Izmena imena datoteke moguća je kod:
  - Stvaranja datoteke
  - Editovanja

- Kompilacije
- Kopiranja

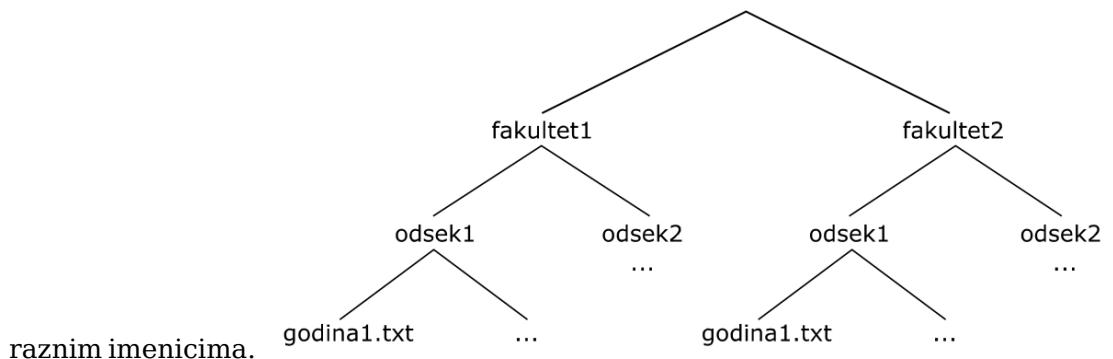
## **Organizacija datoteka**

- Datoteke se grupišu u skupove datoteka (datoteke sa podacima o studentima pojedinih godina studija istog odseka pripadaju istom skupu).
- Svaki skup datoteka predstavlja imenik (directory, folder) koji sadrži imena svih datoteka koje pripadaju datom skupu.
- Radi razlikovanja imenika, svaki od njih poseduje ime koje bira korisnik. Na primer:

  - odsek
  - može da predstavlja ime imenika, koji obuhvata datoteke sa podacima o studentima svih godina studija istog odseka.
  - Razvrstavanjem datoteka uz pomoć imenika nastaje hijerarhijska organizacija datoteka, u kojoj su na višem nivou hijerarhije imenici, a na nižem nivou se nalaze datoteke, čija imena su sadržana u ovim imenicima.
  - Hijerarhijsku oznaku ili putanju (path name) datoteke obrazuju ime imenika za koji je datoteka vezana i ime datoteke. Delove putanje obično razdvaja znak / (ili znak ):

    - odsek1/godina1.txt
    - predstavlja putanju datoteke, koja sadrži podatke o studentima prve godine studija sa prvog odseka.
    - Uobičajeno je da se ime imenika završava znakom /
    - Hijerarhijska organizacija datoteka ima više nivoa, kada jedan imenik sadrži, pored imena datoteka, i imena drugih imenika, odnosno obuhvata, pored datoteka, i druge imenike.
    - Obuhvaćeni imenici se nalaze na nižem nivou hijerarhije. Na primer, imenik fakultet obuhvata imenike pojedinih odseka.
    - Na vrhu hijerarhijske organizacije datoteka se nalazi korenski imenik (root) koji obično nema ime.
    - U slučaju više nivoa u hijerarhijskoj organizaciji datoteka, putanju datoteke obrazuju imena imenika sa svih nivoa hijerarhije (navедена u redosledu od najvišeg nivoa na dole) kao i ime datoteke.
    - Na primer:

- /fakultet/odsek1/godina1.txt
- predstavlja putanju datoteke godina1.txt, koja pripada imeniku odsek1. Ovaj imenik pripada imeniku fakultet, a on pripada korenskom imeniku /.
- Hjjerarhijska organizacija datoteka dozvoljava da postoje datoteke (imenici) sa istim imenima, pod uslovom da pripadaju raznim imenicima.



- U hijerarhijskoj organizaciji datoteka korenskom imeniku pripadaju imenici fakultet1 i fakultet2.
- Svaki od njih sadrži imenike odsek1 i odsek2. Pri tome, oba imenika sa imenom odsek1 sadrže datoteku godina1.txt.
- Putanje omogućuju razlikovanje istoimenih imenika, odnosno istoimenih datoteka. Tako, putanje:
  - /fakultet1/odsek1/ i /fakultet2/odsek1/
  - omogućuju razlikovanje imenika sa imenom
  - odsek1,
  - a putanje:
  - /fakultet1/odsek1/godina1.txt
  - /fakultet2/odsek1/godina1.txt
  - omogućuju razlikovanje datoteka sa imenom godina1.txt
  - Zahvaljujući hijerarhijskoj organizaciji datoteka, moguće je rukovanje skupovima datoteka.
  - Na primer, moguće je kopiranje celog imenika, odnosno kopiranje svih datoteka i imenika, koji mu pripadaju.
  - Navođenjeapsolutneputanje datoteke, sa svim prethodećim imenicima, je potrebno kad god je moguć nesporazum, zbog

datoteka sa istim imenima, odnosno, zbog imenika sa istim imenima.

- Ali, ako postoji mogućnost određivanja nekog imenika kao radnog (working directory), tada se njegova putanja može podrazumevati i ne mora se navoditi.
- Na primer, ako se podrazumeva da je:
  - /fakultet1/odsek1/
  - radni imenik, tada:
  - godina1.txt
  - jednoznačno označava datoteku, koja pripada imeniku odsek1 iz imenika fakultet1.
- Radni imenik omogućuje korišćenje relativnih putanja. Na primer, ako se podrazumeva da je:
  - /fakultet1/
  - radni imenik, tada:
  - odsek1/godina1.txt
  - jednoznačno označava datoteku, koja pripada imeniku fakultet1.
- Datoteke koje pripadaju istoj hijerarhijskoj organizaciji obrazuju sistem datoteka.

## Zaštita datoteka

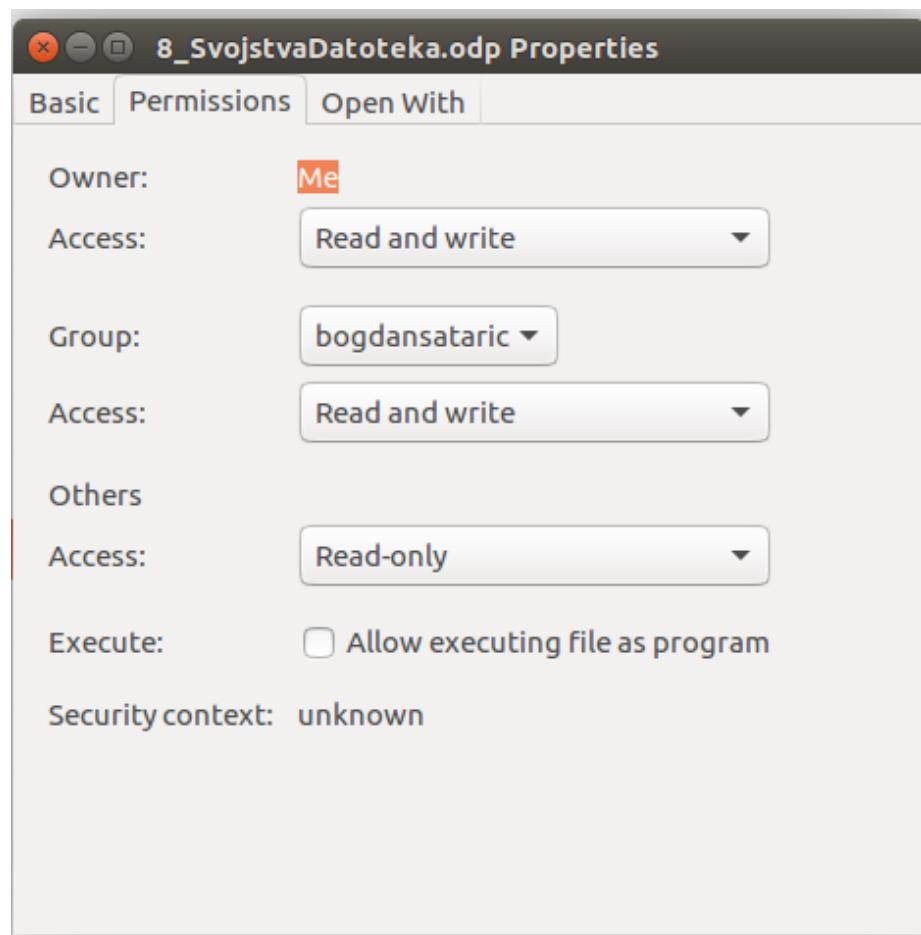
- Za uspešnu upotrebu podataka, trajno pohranjenih u datotekama, neophodna je zaštita datoteka.
- Ona obezbeđuje da podaci, sadržani u datoteci, neće biti izmenjeni bez znanja i saglasnosti njihovog vlasnika (zabрана prava pisanja).
- Takođe, ona obezbeđuje da podatke, sadržane u datoteci jednog korisnika, bez njegove dozvole drugi korisnici ne mogu da koriste (zabranu prava čitanja).
- Na ovaj način uvedeno pravo pisanja i pravo čitanja datoteke omogućuju da se za svakog korisnika jednostavno ustanovi koja vrsta rukovanja datotekom mu je dozvoljena, a koja ne.
- Tako, korisniku, koji ne poseduje pravo pisanja datoteke, nisu dozvoljena rukovanja datotekom, koja izazivaju izmenu njenog sadržaja.

- Ili, korisniku, koji ne poseduje pravo čitanja datoteke, nisu dozvoljena rukovanja datotekom, koja zahtevaju preuzimanje njenog sadržaja.
- Za izvršne datoteke uskraćivanje prava čitanja je prestrogo, jer sprečava ne samo neovlašćeno uzimanje tuđeg izvršnog programa, nego i njegovo izvršavanje (execution).
- Zato je uputno, radi izvršnih datoteka, uvesti posebno pravo izvršavanja programa, sadržanih u izvršnim datotekama.
- Zahvaljujući posedovanju ovog prava, korisnik može da pokrene izvršavanje programa, sadržanog u izvršnoj datoteci, i onda kada nema pravo njenog čitanja.
- Pravo čitanja, pravo pisanja i pravo izvršavanja datoteke predstavljaju tri prava pristupa datotekama (file access control), na osnovu kojih se za svakog korisnika utvrđuje koje vrste rukovanja datotekom su mu dopuštene.
- Da se za svaku datoteku ne bi evidentirala prava pristupa za svakog korisnika pojedinačno, uputno je sve korisnike razvrstati u kategorije i za svaku od njih vezati pomenuta prava pristupa.
- Iskustvo pokazuje da su dovoljne tri kategorije korisnika. Jednoj pripada vlasnik datoteke, drugoj njegovi saradnici, a trećoj ostali korisnici.

## Matrica zaštite

- Ima tri kolone (po jednu za svaku kategoriju korisnika) i onoliko redova koliko ima datoteka.
- U preseku svakog reda i svake kolone matrice zaštite navode se prava pristupa datoteci iz posmatranog reda za korisnike koji pripadaju kategoriji iz posmatrane kolone.

	vlasnik	saradnik	ostali
datoteka_1.bin	pisanje čitanje izvršavanje	- čitanje izvršavanje	- - izvršavanje
datoteka_2.bin	- čitanje izvršavanje	- izvršavanje	- - izvršavanje
datoteka_n.bin	... pisanje čitanje -	... - - -	... - - -



```
bogdansataric@bogdansataric-CELSIUS-M740:~/Desktop/Grive/Predmeti/OS$ ls -la
total 4744
drwxrwxr-x 13 bogdansataric bogdansataric 4096 апр 19 11:28 .
drwxrwxr-x  8 bogdansataric bogdansataric 4096 дец 25 11:12 ..
drwxrwxr-x  5 bogdansataric bogdansataric 4096 окт  6 2016 2012-2013
drwxrwxr-x  6 bogdansataric bogdansataric 4096 окт 17 2016 2013-2014
drwxrwxr-x  8 bogdansataric bogdansataric 4096 окт 17 2016 2014-2015
drwxrwxr-x 11 bogdansataric bogdansataric 4096 окт 12 2016 2015-2016
drwxrwxr-x 10 bogdansataric bogdansataric 4096 јан 29 16:39 2016-2017
drwxrwxr-x  9 bogdansataric bogdansataric 4096 апр 18 13:33 2017-2018
drwxrwxr-x  6 bogdansataric bogdansataric 4096 мај 14 2017 Autotest alati i vežbe
-rw-----  1 bogdansataric bogdansataric 124883 мај  8 10:20 c++11_podesavanje_kompajlera.odt
-rw-rw-r--  1 bogdansataric bogdansataric 351266 мај  8 10:20 c++11_podesavanje_kompajlera.pdf
drwx----- 11 bogdansataric bogdansataric 4096 дец 14 11:16 colibry_3.6.1-2
-rwxrwxrwx  1 bogdansataric bogdansataric 169 феб 19 12:28 cropar.sh
drwxrwxr-x  2 bogdansataric bogdansataric 4096 феб  7 2017 Knjige
-rw-rw-r--  1 bogdansataric bogdansataric 108 апр 19 11:28 .-lock.OS evidencija.docx#
-rw-rw-r--  1 bogdansataric bogdansataric 1959058 феб 19 12:17 0S2017-2x1.pdf
-rw-rw-r--  1 bogdansataric bogdansataric 1852907 феб 20 2017 0S2017.pdf
-rw-rw-r--  1 bogdansataric bogdansataric 75658 феб 19 12:29 05_cesta_pitanja_o_predmetu-2x1.pdf
-rw-rw-r--  1 bogdansataric bogdansataric 27627 феб 19 11:55 05_cesta_pitanja_o_predmetu.odt
-rw-----  1 bogdansataric bogdansataric 131872 феб 19 11:58 05_cesta_pitanja_o_predmetu.pdf
-rw-rw-r--  1 bogdansataric bogdansataric 11005 апр 19 11:28 05_evidencija.docx
-rw-rw-r--  1 bogdansataric bogdansataric 18197 мај 28 11:38 Pravila_za_studente.odt
-rw-rw-r--  1 bogdansataric bogdansataric 72508 мај 28 11:42 Pravila_za_studente.pdf
drwxrwxr-x  2 bogdansataric bogdansataric 4096 апр 19 11:31 Predavanja
-rw-rw-r--  1 bogdansataric bogdansataric 27306 мај 28 11:28 Uputstvo_za_asistente.odt
-rw-rw-r--  1 bogdansataric bogdansataric 89744 мај 28 11:42 Uputstvo_za_asistente.pdf
-rw-----  1 bogdansataric bogdansataric 15715 окт  6 2016 uputstvo_za_c++_help.odt
drwxrwxr-x  5 bogdansataric bogdansataric 4096 дец 28 2016 Zadaci_za_testove
bogdansataric@bogdansataric-CELSIUS-M740:~/Desktop/Grive/Predmeti/OS$ 
```

- Prava pristupa iz matrice zaštite se mogu vezati za:
- datoteke i čuvati u deskriptorima datoteka
- vezati za korisnike
- U prvom slučaju redovi matrice zaštite su raspoređeni po deskriptorima raznih datoteka, a u drugom slučaju elemente kolona matrice zaštite čuvaju pojedini korisnici.
- Za uspeh izloženog koncepta zaštite datoteka neophodno je onemogućiti neovlašćeno menjanje matrice zaštite.
- Jedino vlasnik datoteke sme da zadaje i menja prava pristupa sebi, svojim saradnicima i ostalim korisnicima. Zato je potrebno znati za svaku datoteku ko je njen vlasnik.
- To se postiže tako što svoju aktivnost svaki korisnik započinje svojim predstavljanjem (login).
- U toku predstavljanja korisnik predočava svoje ime i navodi dokaz da je on osoba za koju se predstavlja, za šta je, najčešće, dovoljna lozinka.
- Pređeno ime i navedena lozinka se porede sa spiskom imena i spiskom za njih vezanih lozinki registrovanih korisnika.
- Predstavljanje je uspešno, ako se u spiskovima imena i lozinki registrovanih korisnika pronađu pređeno ime i navedena lozinka.
- Predstavljanje korisnika se zasniva na pretpostavci da su njihova imena javna, ali da su im lozinke tajne.

- Zato je i spisak imena registrovanih korisnika javan, a spisak lozinki registrovanih korisnika tajan.
- Jedina dva slučaja, u kojima ima smisla dozvoliti korisnicima posredan pristup spisku lozinki, su:
  - radi njihovog predstavljanja
  - radi izmene njihove lozinke
- Za predstavljanje korisnika uvodi se posebna operacija, koja omogućuje samo proveru da li se zadani par (ime, lozinka) može pronaći u spiskovima imena i lozinki registrovanih korisnika.
- Slično, za izmenu lozinki uvodi se posebna operacija, koja omogućuje samo promenu lozinke onome ko zna postojeću lozinku.
- Sva druga rukovanja spiskovima imena i lozinki registrovanih korisnika, kao što su ubacivanje u ove spiskove novih parova (ime, lozinka), ili njihovo izbacivanje iz ovih spiskova, nalaze se u nadležnosti poverljive osobe, koja se naziva administrator (superuser).

## **UID i GID**

- Da bi se pojednostavila provera korisničkih prava pristupa, uputno je, umesto imena korisnika, uvesti njegovu numeričku oznaku.
- Radi klasifikacije korisnika zgodno je da ovu numeričku oznaku obrazuju dva redna broja.
- Prvi od njih označava korisnika, a drugi od njih označava grupu kojoj korisnik pripada.
- Podrazumeva se da su svi korisnici iz iste grupe međusobno saradnici.
- Prema tome, redni broj korisnika (UID, User IDentification) jednoznačno određuje vlasnika.
- Saradnici vlasnika su svi korisnici koji imaju isti redni broj grupe (GID, Group Identification) kao i vlasnik.
- Posebna grupa se rezerviše za administratore.

## **Matrica zaštite**

- Da se provera ne bi obavljala prilikom svakog pristupa datoteci, umesno je takvu proveru obaviti samo pre prvog pristupa.
- To je zadatak operacije otvaranja datoteke, koja prethodi operacijama, kao što su pisanje ili čitanje datoteke.

- Pomoću operacije otvaranja se saopštava i na koji način korisnik namerava da koristi datoteku.
- Ako je njegova namera u skladu sa njegovim pravima, otvaranje datoteke je uspešno, a pristup datoteci je dozvoljen, ali samo u granicama iskazanih namera.
- Iza operacija pisanja ili čitanja datoteke sledi operacija zatvaranja datoteke.
- Numerička oznaka vlasnika datoteke i prava pristupa korisnika iz pojedinih klasa predstavljaju atribute datoteke.
- Zaštita datoteka uvodi pojam sigurnost (security) koji se odnosi na uspešnost zaštite od neovlašćenog korišćenja ne samo datoteka, nego i ostalih delova računara kojima upravlja operativni sistem.
- Sigurnost se bavi načinima prepoznavanja ili identifikacije korisnika (authentication), kao i načinima provere njihovih prava pristupa (authorization).
- Operativni sistem nudi mehanizme sigurnosti pomoću kojih mogu da se ostvare različite politike sigurnosti.

## **ext2fs**

### **Blokovi**

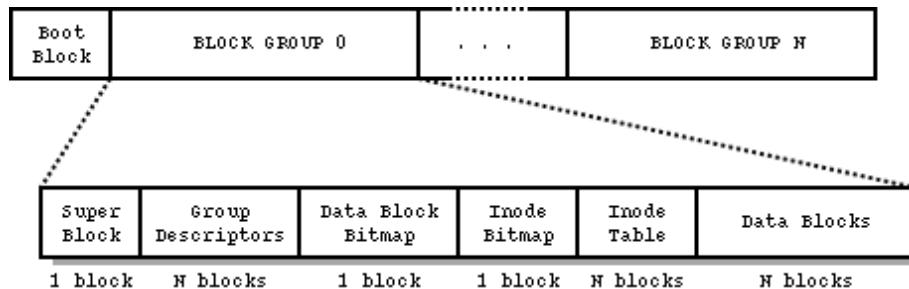
- ext2 smatra da se ono na čemu se zasniva sastoji od sektora (mi smo ih zvali blokovi na našem lažnom disku) koji se grupišu u blokove.
- Jedan blok u ex2fs je par Kb koji mora biti stepen broja 2. Na mom disku to je 4kb.
- Blokovi se grupišu u grupe ne bi li se smanjila fragmentacija. Maksimalna veličina grupe je  $8 \times$  veličina bloka u blokovima, tj. jedna grupa na mom disku je  $8 \times 4096 = 32\ 768$  blokova. Ovo je neophodno zato što bitmapa, koja čuva šta je slobodno a šta nije zauzima samo jedan blok.

### **Superblok**

- Superblok sadrži sve informacije o tome kako je podešen fajl sistem. Primarno, to se nalazi 1024 bajtova daleko od početka diska, ali zato što je apsolutno neophodno da se ovi podaci ne izgube (inače nema šanse da se podaci sa particije vrati) superblok se replicira.

- U starijim verzijama ext2fs stavlja se kopija na početak svake grupe blokova. To gubi dosta prostora, te se danas tipično samo stavlja tek u poneku grupu, tipično stepene 3, 5, ili 7.

## Grupa i superblok



## Inode

- Inoda (index node) je fundamentalan koncept za ext2 fajl sistem: sve o čemu fajl sistem vodi računa (fajl, direktorijum, link...) ima indeksni čvor, inodu. Taj čvor sadrži sve metapodatke o fajlu (tu se nalaze podaci o vlasniku, dozvolama, itd.) kao i, vrlo bitno, pokazivač ka blokovima sistema gde se fajlovi nalaze.
- Sve inode se nalaze u inode tabelama, sa po jednom takvom tabelom po grupi blokova.

## Fajl sistem pokazivači i inoda

- Postavlja se pitanje, kako da znamo gde se sve na diskusu nalazi naš fajl? ext2 ima jednostavan pristup. Prvo čuva 12 pokazivača (broja bloka) sa tkzv. direktnim podacima. Ako fajl stane tu, dobro. Ako ne, ima i pokazivač ka jednostruko indirektnom bloku.
- Taj pokazivač pokazuje na blok koji u sebi sadrži sekvencu pokazivača ka blokovima sa podacima.
- Ako i dalje nema mesta? Onda imamo pokazivač ka dvostruko indirektnom bloku koji u sebi sadrži pokazivače ka blokovima koji u sebi imaju pokazivače ka blokovima podataka.
- Ako i dalje nema mesta?
- Onda imamo trostruko indirektnu adresu
- To je pokazivač na blok koji sadrži listu pokazivača koji pokazuju na blokove koji sadrže liste pokazivača koje pokazuju na blokove

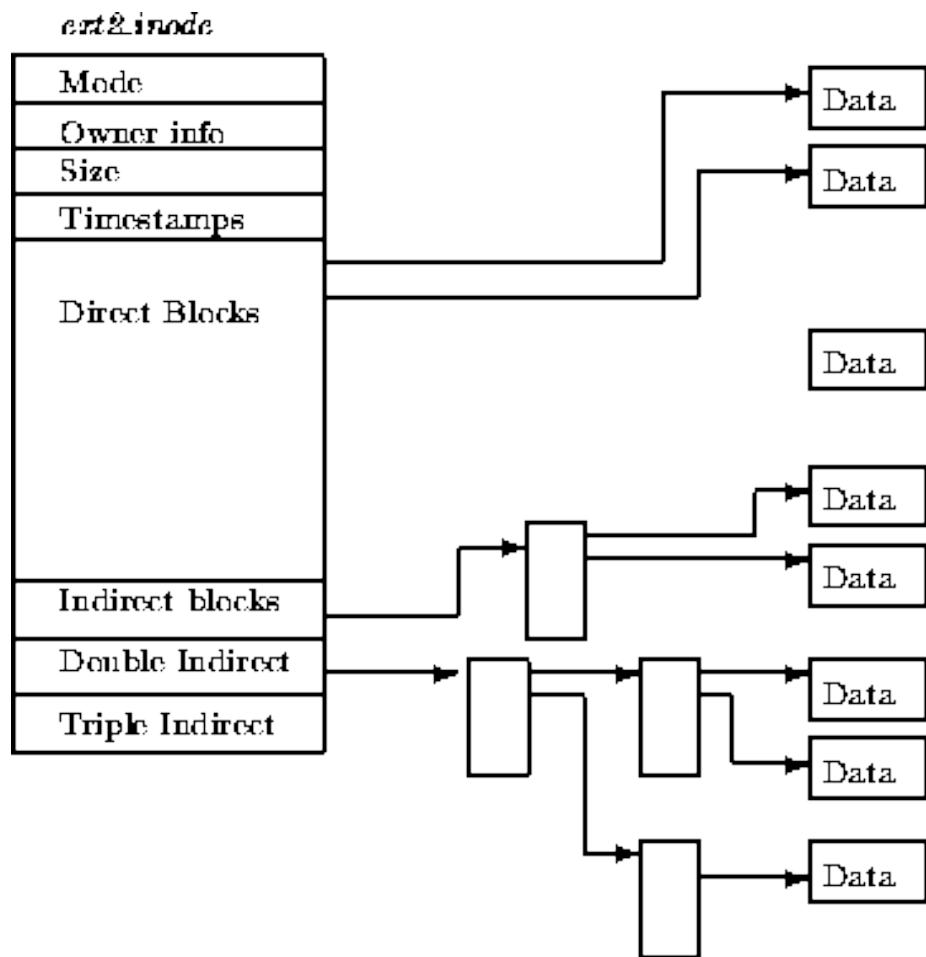
koje imaju u sebi podatke. Ovo znači da ima maksimalna veličina fajla a to je za sistem sa 32 - bitnim pokazivačima na blok

- $12 * \text{bs} + (\text{bs} / 4) * \text{bs} + (\text{bs} / 4) * (\text{bs} / 4) * \text{bs} + (\text{bs} / 4) * (\text{bs} / 4) * (\text{bs} / 4) * \text{bs}$  bajtova,
- gde je bs veličina bloka,
- tako da za 4kb blok maksimalna veličina bi bila oko  $4402345721856$  bajtova, odn. oko 4100GB.

### **Veličina fajla u praksi**

U praksi, fajl je ograničen on - disk veličinom bloka (koja se razlikuje od ext2 bloka i iznosi 512 bajtova) i time što je sistem 32 - bitni, te je maksimalan broj on - disk blokova u  $2^{32} - 1$ , što znači da je stvarni limit na veličinu fajla  $(2^{32} - 1) \cdot 512$ , odn. 2TB

## Inoda



## Direktorijumi i linkovi

- Što se tiče ext2 sve je fajl.
- Samo što određeni fajlovi imaju posebno interpretiran sadržaj, npr:
- Linkovi su 'pokazivač' ka drugom fajlu
- Ovo se odnosi na meke, simbolične linkove. Tvrdi linkovi su u stvari stavka u direktorijumu koja se odnosi na istu inod-u kao i neki drugi fajl.
- Direktorijumi su lista pokazivača ka fajlovima

## **Da li se ovo koristi?**

- Tehnički, koriste se modernije verzije, ext3 i ext4 koje su u velikoj meri kompatibilne sa tim kako ext2 radi.
- Ext je od 'extensible' tako da je oduvek bila ideja da fajl sistem može da se proširi.
- Glavna promena koja je ubaćena u ext3 je vođenje žurnala diska.

## **Žurnal fajl sistema—motivacija**

- Za određene operacije diska neželjen prekid (recimo gubljenjem struje) je spektakularno loša ideja. Ako brišemo fajl na disku očekujemo sledeće operacije u ext2:
  - Uklonimo zapis o fajlu iz sadžraja direktorijuma tog fajla.
  - Ako nema više tvrdih linkova na nju, oslobođimo inode tog fajla podešavanjem bitmape.
  - Oslobođimo sve blokove na koje pokazujemo tako što modifikujemo bitmape i metapodatke svih grupa gde se nalaze blokovi tog fajla.
  - Za veliki fajl ovo su potencijalno stotine upisa na disk, a fajl sistem je samo konzistentan sam sa sobom na početku svih tih upisa i na kraju. Prekid bilo gde izaziva, npr. prostor koji ne koristi nijedan fajl, a koji je tehnički i dalje zauzet, ili inode koji se vodi kao zauzet a kome ne možete da pridete ili hiljadu drugih problema.
  - To znači da gubitak struje dovodi do potencijalnog oštećenja diska što moramo da (pokušavamo) da otklonimo posebnim komandama kao što je fsck. Ovo nije idealno.

## **Žurnal fajl sistema**

- Rešenje ovog problema jeste da postoji deo fajl sistema (žurnal) alociran i organizovan kao cirkularni bafer koji sadrži zapis svih promena koje hoćemo da uradimo i koji se povremeno prazni tako što se logovane operacije izvrše na disku.
- Ako nestane struje u sred procesa, prilikom sledećeg aktiviranja fajl sistema (Unix to zove 'mount'-ovanje fajl sistema), se primeti da ima neizvršenih žurnal operacija, i postara se da su ili sve te operacije izvršene (te je sistem konzistentan) ili se od cele operacije odustalo.

## **Konzistentnost žurnala**

- Ovo nije savršeno rešenje. Prvi problem je: Šta ako nestane struje baš kada pišemo žurnal?
- Rešenje je da se izračuna i prvo upiše kontrola vrednost (checksum) cele transakcije, tako da ako pukne upis pre kraja, to se može preko kontrolne vrednosti ustvrditi i ta transakcija ignorisati u žurnalu.
- Drugi, mnogo veći problem, jeste interakcija između keširanja upisa na disk i žurnala. Žurnal se bazira na ideji da znamo redosled operacija upisa i da nećemo, recimo, početi da upisujemo podatke pre nego alociramo prostor kroz metapodatke.
- I fajl sistem i sami disk uređaji često menjaju redosled operacija upisa zbog performansi. I mi smo to radili!
- Ovo je rizik koji neki fajl sistemi, ext4, na primer, rešavaju uvođenjem barijera: tačaka gde se keš nasilno spusti na sam disk.

## **Fajl sistemi uopšte**

### **Sloj operativnog sistema za rukovanje datotekama**

- Zadatak: ozbezbediti punu slobodu rukovanja podacima u datotekama.
- Punu slobodu rukovanja podacima nudi predstava datoteke kao niza bajta.
- Niz se može, po potrebi, menjati i njemu se može pristupati u proizvolnjom redosledu, korišćenjem rednog broja bajta za njegovu identifikaciju.
- Ovakva predstava datoteke dozvoljava da se datoteka posmatra kao skup slogova iste ili različite dužine, koji se identificuju pomoću posebnih indeksa.

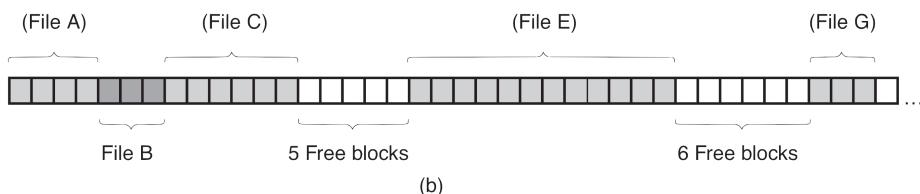
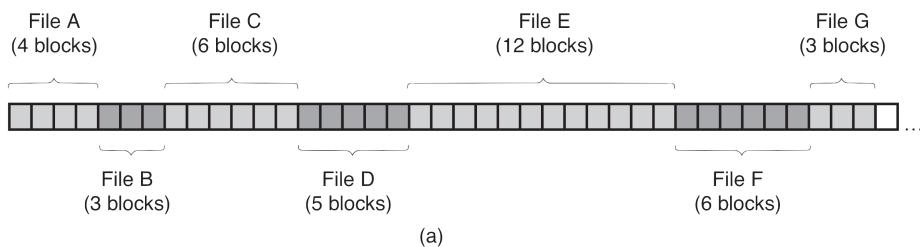
## **Kontinualne datoteke**

- Sadržaji datoteke se nalaze u blokovima masovne memorije. Za bilo kakvo rukovanje ovim sadržajem neophodno je da on dospe u radnu memoriju.
- Zato je rukovanje bajtima sadržaja datoteke neraskidivo povezano sa prebacivanjem blokova sa ovim bajtima između radne i masovne memorije.

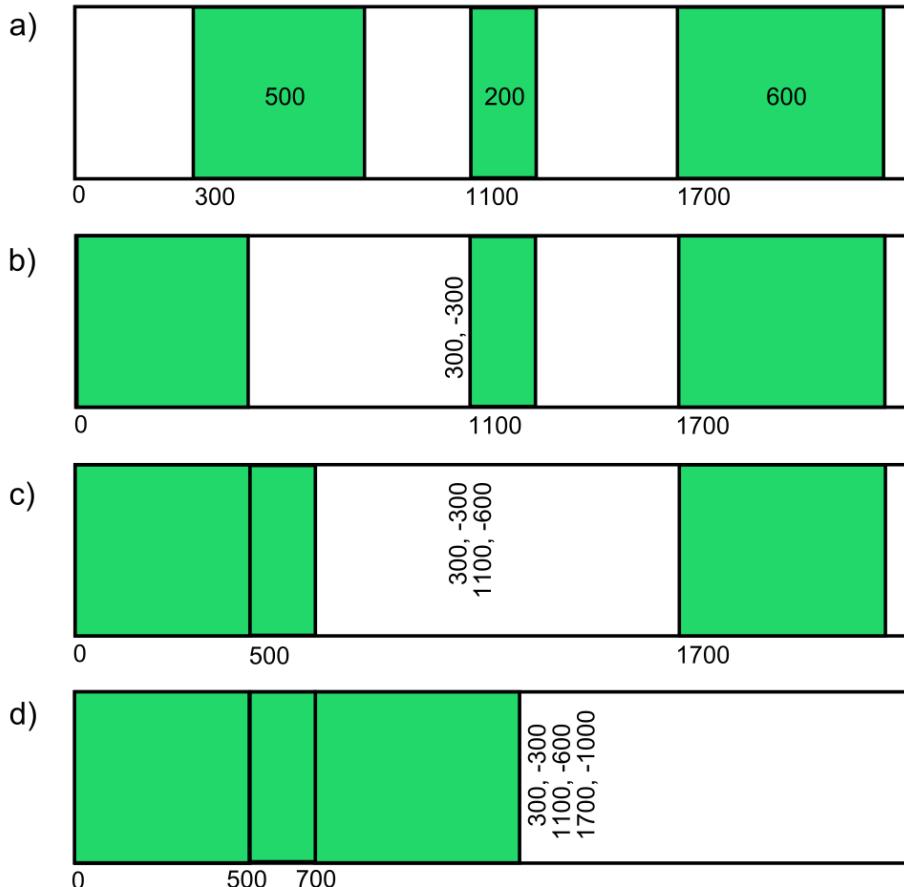
- Pri tome se bajti prebacuju iz radne u masovnu memoriju radi njihovog trajnog čuvanja, a iz masovne u radnu memoriju radi obrade.
- Da bi ovakvo prebacivanje bilo moguće, neophodno je da sloj za rukovanje datotekama uspostavi preslikavanje rednih brojeva bajta u redne brojeve njima odgovarajućih blokova.
- Ovakvo preslikavanje se najlakše uspostavlja, ako se sadržaj datoteke nalazi u susednim blokovima.
- Ovakve datoteke se nazivaju kontinualne (contiguous).
- Kod kontinualnih datoteka redni broj bloka sa traženim bajtom se određuje kao količnik rednog broja bajta i veličine bloka, izražene brojem bajta koje sadrži svaki blok.
- Pri tome, ostatak deljenja ukazuje na relativni položaj bajta u bloku.
- Kontinualne datoteke zahtevaju od sloja za rukovanje datotekama da za svaku datoteku vodi evidenciju o:
  - imenu datoteke
  - rednom broju početnog bloka
  - dužini datoteke
  - Ove podatke čuva deskriptor datoteke.
- Dužina datoteke može biti izražena brojem bajta, ali i brojem blokova, koga obavezno prati podatak o popunjenošći poslednjeg zauzetog bloka.
- Pojava da poslednji zauzeti blok datoteke nije popunjen do kraja se naziva interna fragmentacija (internal fragmentation).
- Ova pojava je važna, jer nepotpuni poslednji zauzeti blok datoteke predstavlja neupotrebljen deo masovne memorije.
- Sloj za rukovanje datotekama obavezno vodi i evidenciju slobodnih blokova masovne memorije.
- Za potrebe kontinualnih datoteka bitno je da ova evidencija olakša pronalaženje dovoljno dugačkih nizova susednih blokova.
- Zato je podesna evidencija u obliku niza bita (bit map), u kome svaki bit odgovara jednom bloku i pokazuje da li je on zauzet (0) ili slobodan (1).
- U slučaju ovakve evidencije, pronalaženje dovoljno dugačkih nizova susednih blokova se svodi na pronalaženje dovoljno

dugačkog niza jedinica u nizu bita koji odslikava zauzetost masovne memorije.

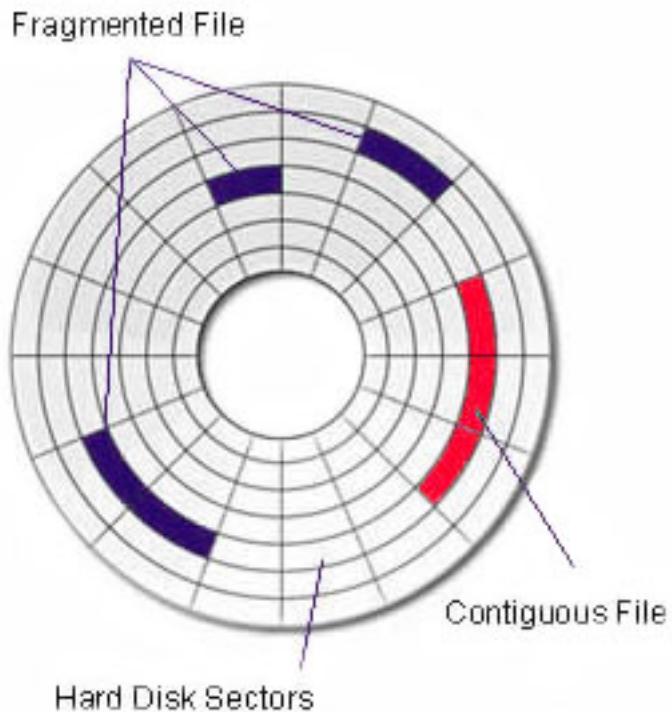
- Rukovanje slobodnim blokovima masovne memorije zahteva sinhronizaciju (međusobnu isključivost procesa), da bi se, na primer, izbeglo da više procesa, nezavisno jedan od drugog, zauzme iste slobodne blokove masovne memorije.
- Pojava iscepkanosti slobodnih blokova masovne memorije u kratke nizove susednih blokova otežava rukovanje kontinualnim datotekama.
- Ta pojava se zove eksterna fragmentacija (external fragmentation).
- Ona nastaje kao rezultat višestrukog stvaranja i uništenja datoteka u slučajnom redosledu, pa nakon uništavanja datoteka ostaju nizovi slobodnih susednih blokova, međusobno razdvojeni blokovima postojećih datoteka.
- Problem eksterne fragmentacije se povećava, kada se, prilikom traženja dovoljno dugačkog niza susednih blokova, pronađe niz duži od potrebnog, jer se tada zauzima (allocation) samo deo blokova u pronađenom nizu.
- To dovodi do daljeg drobljenja nizova slobodnih susednih blokova, jer preostaju sve kraći nizovi slobodnih susednih blokova.



- Eksterna fragmentacija je problematična, jer posredno izaziva neupotrebljivost slobodnih blokova masovne memorije.
- Eksterna fragmentacija onemogućuje stvaranje kontinualne datoteke, čija dužina je jednaka sumi slobodnih blokova, kada oni ne obrazuju niz susednih blokova.



- Problem eksterne fragmentacije se može rešiti sabijanjem (compaction) datoteka, tako da svi slobodni blokovi budu potisnuti iza datoteka i da tako obrazuju niz susednih blokova. Mana ovog postupka je njegova dugotrajnost.
- U opštem slučaju produženje kontinualne datoteke je komplikovano, jer zahteva stvaranje nove, veće kontinualne datoteke, prepisivanje sadržaja produžavane datoteke u novu datoteku i uništavanje produžavane datoteke.
- Problem produženja kontinualne datoteke se ublažava, ako se dozvoli da se kontinualna datoteka sastoji od više kontinualnih delova.



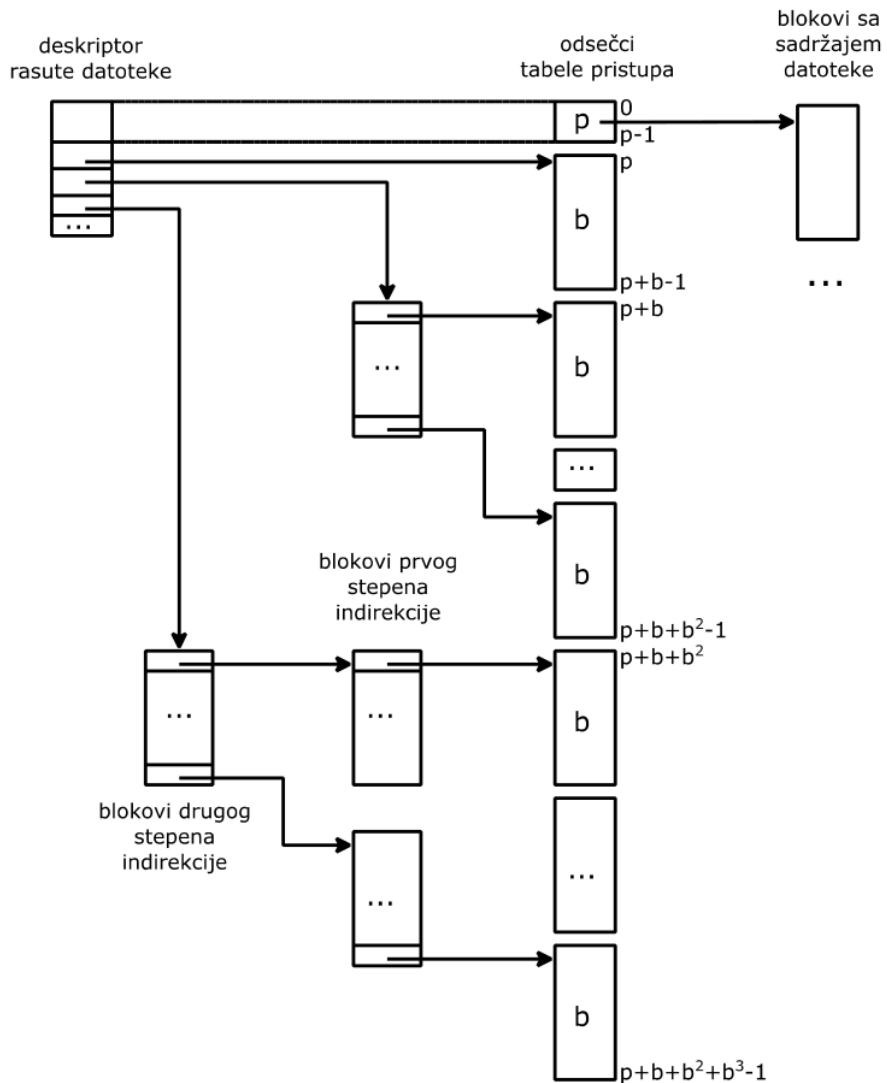
- Pri tome se za svaki od ovih delova u deskriptoru datoteke čuvaju podaci o rednom broju početnog bloka dotičnog dela i o njegovoj dužini (extent list).
- Ovakav pristup je zgodan za veoma dugačke datoteke (namenjene za čuvanje zvučnog ili video zapisa).

## Rasute datoteke

- Upotrebnu vrednost kontinualnih datoteka značajno smanjuju problemi:
  - eksterne fragmentacije
  - potreba da se unapred zna njihova veličina
  - teškoće sa njihovim produžavanjem.
- Zato se umesto kontinualnih koriste rasute (noncontiguous) datoteke, čiji sadržaj je smešten (rasut) u nesusednim blokovima masovne memorije.
- Kod rasutih datoteka redni brojevi bajta se preslikavaju u redne brojeve blokova pomoću posebne tabele pristupa (file allocation

table - FAT).

- Njeni elementi sadrže redne brojeve blokova. Indekse ovih elemenata određuje količnik rednog broja bajta i veličine bloka.
- Iz prethodnog sledi da dužinu rasutih datoteka ograničava veličina tabele pristupa.
- Zato se veličina tabele pristupa dimenzionira tako da zadovolji najveće praktične zahteve u pogledu dužine rasutih datoteka.
- Tabele pristupa se čuvaju u blokovima masovne memorije (kao, uostalom, i sadržaji datoteka).
- Radi manjeg zauzeća, važno je da se u blokovima masovne memorije ne čuva uvek cela tabela pristupa, nego samo njen neophodan (stvarno korišćen) deo.
- Zato se tabela pristupa deli u odsečke.
- Početni odsečak, sa  $p$  elemenata tabele pristupa je uvek prisutan. On nije veći od bloka masovne memorije. Dodatni odsečci su prisutni samo kad su neophodni.
- Svaki dodatni odsečak je jednak bloku masovne memorije i može da sadrži  $b$  elemenata tabele pristupa ( $b > p$ ).
- Prema tome, tabela pristupa svake rasute datoteke zauzima jedan blok, u kome se nalazi početni odsečak ove tabele sa  $p$  njenih elemenata.
- Za tabelu pristupa se, po potrebi, zauzima još jedan blok sa dodatnim odsečkom, u kome se nalazi narednih  $b$  njenih elemenata.
- Kada zatreba još dodatnih odsečaka, za tabelu pristupa se zauzima poseban blok prvog stepena indirekcije.
- On sadrži do  $b$  rednih brojeva blokova sa dodatnim odsečcima.
- U svakom od njih se nalazi  $b$  novih elemenata tabele pristupa.
- Na kraju, po potrebi, za ovu tabelu se zauzima poseban blok drugog stepena indirekcije.
- On sadrži do  $b$  rednih brojeva blokova prvog stepena indirekcije.
- Svaki od njih sadrži do  $b$  rednih brojeva blokova sa dodatnim odsečcima, a u svakom od njih se nalazi  $b$  novih elemenata tabele pristupa.
- Prema tome, ukupno ima  $1+b+b^2$  dodatnih odsečaka, svaki sa  $b$  elemenata tabele pristupa.



- Deskriptor rasute datoteke sadrži početni odsečak tabele pristupa, redni broj njenog prvog dodatnog odsečka, redni broj bloka prvog stepena indirekcije i redni broj bloka drugog stepena indirekcije.
- Pored toga, ovaj deskriptor sadrži i dužinu rasute datoteke, da bi se znalo koliko blokova je zauzeto sadržajem i koliko je popunjen poslednji zauzeti blok.
- Ideja, korišćena za organizaciju tabele pristupa, može da se upotrebi i za organizaciju evidencije slobodnih blokova masovne

memorije.

- U ovom slučaju ova evidencija ima oblik liste slobodnih blokova.
- Slobodni blokovi, uvezani u ovu listu, sadrže redne brojeve ostalih slobodnih blokova, pa podsećaju na blokove prvog stepena indirekcije.

## Konzistentnost sistema datoteka

- Iza rukovanja datotekama krije se rukovanje blokovima masovne memorije, u kojima se nalaze i sadržaj i deskriptori, eventualno, dodatni odsečci tabele pristupa svake rasute datoteke.
- Rukovanje ovim blokovima usložnjava činjenica da se međusobno zavisni podaci nalaze u raznim blokovima.
- Pošto se blokovi modifikuju u radnoj memoriji, a trajno čuvaju u masovnoj memoriji, prirodno je da u pojedinim trenucima postoji razlika između blokova u masovnoj memoriji i njihovih kopija u radnoj memoriji.
- Probleme izaziva gubitak kopija blokova u radnoj memoriji, na primer, zbog nestanka napajanja.
- Tako, na primer, produženje rasute datoteke zahteva:
  - izmenu evidencije slobodnih blokova, radi isključivanja pronađenog slobodnog bloka iz ove evidencije
  - izmenu tabele pristupa produžavane rasute datoteke, radi smeštanja rednog broja novog bloka u element ove tabele.
- Izmena evidencije slobodnih blokova dovodi do promene jedne od kopija njenih blokova u radnoj memoriji. Isti efekat ima i izmena tabele pristupa produžavane rasute datoteke.
- Ako obe izmenjene kopije budu prebačene u masovnu memoriju, tada je produženje rasute datoteke uspešno obavljen.
- Ako ni jedna od kopija ne dospe u masovnu memoriju, tada produženje rasute datoteke nije obavljen, jer nije registrovano u masovnoj memoriji.
- Ali, ako samo jedna od promenjenih kopija dospe u masovnu memoriju, tada se javljaju problemi konzistentnosti sistema datoteka.
- U jednom slučaju, kada samo promenjena kopija bloka evidencije slobodnih blokova dospe u masovnu memoriju, blok isključen iz ove evidencije postaje izgubljen, jer njegov redni

broj nije prisutan niti u ovoj evidenciji, a niti u tabeli pristupa neke od rasutih datoteka.

- U drugom slučaju, kada samo promenjena kopija bloka tabele pristupa dospe u masovnu memoriju, blok, pridružen ovoj tabeli, ostaje i dalje uključen u evidenciju slobodnih blokova.
- Prvi slučaj je bezazlen, jer se izgubljeni blokovi mogu pronaći.
- Pronalaženje izgubljenih blokova se zasniva na traženju blokova koji nisu prisutni ni u evidenciji slobodnih blokova, ni u tabelama pristupa rasutih datoteka.
- Za razliku od prvog slučaja, drugi slučaj je neprihvatljiv, jer može da izazove istovremeno uključivanje istog bloka u više rasutih datoteka, čime se narušava njihova konzistentnost.
- Zato je neophodno uvek prebacivati u masovnu memoriju prvo promenjenu kopiju bloka evidencije slobodnih blokova, pa tek iza toga i promenjenu kopiju bloka tabele pristupa.
- Znači, potrebno je paziti na redosled u kome se izmenjene kopije blokova prebacuju u masovnu memoriju.
- U opštem slučaju konzistentnost sistema datoteka može da se zasniva na vodenju pregleda izmena (journal).
- Pre bilo kakve izmene sistema datoteka, u pregledu izmena se registruje potpun opis nameravane izmene, na osnovu koga je moguće izvršiti oporavak sistema datoteka posle nedovršene izmene.
- Tek nakon toga se pristupa izmeni sistema datoteka. Po uspešno obavljenoj izmeni, u pregledu izmena se to i registruje.
- Ako u pregledu izmena nije registrovan potpun opis nameravane izmene, tada izmena nije ni započeta, pa je sistem datoteka u konzistentnom stanju.
- Kada je u pregledu izmena registrovan potpun opis nameravane izmene, ali nije registrovano njen uspešno obavljanje, tada je sistem datoteka moguće vratiti u konzistentno stanje.
- Ako su u pregledu izmena registrovani potpuni opis nameravane izmene i njen uspešno obavljanje, tada je sistem datoteka u konzistentnom stanju.
- Ideja pregleda izmena može da bude osnova za organizovanje celog sistema datoteka (log structured file system).
- U ovom pristupu izmena svake datoteke se registruje samo u posebnom pregledu izmena, čijom kasnijom analizom se, po potrebi, rekonstruiše aktuelni sadržaj datoteke.

- Nakon izmene kopije bloka u radnoj memoriji, važno je što pre izmenjenu kopiju prebaciti u masovnu memoriju, radi smanjenja mogućnosti da se ona izgubi (na primer, kao posledica nestanka napajanja).
- To je naročito važno, ako izmena nije rezultat automatske obrade, nego ljudskog rada (na primer, editiranja), jer se tada ne može automatski rekonstruisati.

## Baferski prostor

- Pristupi sadržaju datoteke mogu zahtevati prebacivanje više blokova u radnu memoriju:
- bloka sa deskriptorom datoteke
- jednog ili više dodatnih blokova tabele pristupa
- bloka sa traženim bajtima sadržaja
- Pošto je, sa stanovišta procesora, prenos blokova na relaciji radna i masovna memorija, spor (dugotrajan), dobra ideja je zauzeti u radnoj memoriji prostor za više bafera, namenjenih za čuvanje kopija korišćenih blokova (block cache, buffer cache).
- Pošto je radna memorija mnogo manja od masovne, njeni baferi mogu istovremeno da sadrže mali broj kopija blokova masovne memorije.
- Zato je važno da baferi sadrže kopije blokova, koje će biti korišćene u neposrednoj budućnosti, jer se samo tako značajno ubrzava obrada podataka.
- Problem se javlja kada su svi baferi napunjeni, a potrebno je pristupiti bloku masovne memorije, čija kopija nije prisutna u nekom od bafera.
- U tom slučaju neizbežno je oslobađanje nekog od bafera, da bi se u njega smestila kopija potrebnog bloka.
- Iskustvo pokazuje da je najbolji pristup osloboditi bafer za koga trenutak poslednjeg pristupa njegovom sadržaju prethodi trenucima poslednjeg pristupa sadržajima svih ostalih bafera (Least Recently Used - LRU).
- Za takav bafer se kaže da ima najstariju referencu.
- Pri oslobađanju bafera, njegov dotadašnji sadržaj se poništava, kada bafer sadrži neizmenjenu kopiju bloka, jer je ona identična bloku masovne memorije.

- U suprotnom slučaju, neophodno je sačuvati izmene, pa se kopija prebacuje u masovnu memoriju.
- U oslobođeni bafer se prebacuje kopija potrebnog bloka masovne memorije.
- Da bi se znalo koja od kopija ima najstariju referencu, baferi se uvezuju u listu.
- Na početak ovakve liste se uvek prebacuje bafer sa upravo korišćenom kopijom (sa najnovijom referencom), pa na njen kraj nužno dospeva bafer sa najstarijom referencom.
- Baferovanje izmenjenih kopija blokova u radnoj memoriji zahteva da se odredi trenutak u kome se izmenjena kopija prebacuje u masovnu memoriju.
- Ako se izmenjena kopija prebacuje u masovnu memoriju odmah nakon svake izmene, tada se usporava rad.
- Ako se izmenjena kopija prebacuje u masovnu memoriju tek pri oslobađanju bafera, tada se povećava mogućnost gubljenja izmene.
- Rešavanje ovoga problema se može posredno prepustiti korisniku, ako se uvede posebna sistemska operacija (`sync()`) za izazivanje prebacivanja sadržaja bafera u masovnu memoriju.
- U tom slučaju izmenjene kopije dospevaju u masovnu memoriju ili kada se baferi oslobađaju ili kada to zatraži korisnik.
- Potreba da se kopija bloka što brže nakon izmene prebaci u masovnu memoriju je u suprotnosti sa nastojanjem da se blokovi što ređe prenose između radne i masovne memorije.
- Na brzinu prebacivanja podataka između radne i masovne memorije važan uticaj ima i veličina bloka.
- Što je blok veći, u proseku se potroši manje vremena na prebacivanje jednog bajta između radne i masovne memorije.
- Međutim, što je blok veći, veća je i interna fragmentacija.
- Ta dva oprečna zahteva utiču na izbor veličine bloka, koja se kreće od 512 bajta do 8192 bajta.

## **Deskriptor datoteke**

- Deskriptor datoteke, pored atributa koji omogućuju preslikavanje rednih brojeva bajta u redne brojeve blokova, sadrži i:
- numeričku oznaku vlasnika datoteke

- prava pristupa datoteci za njenog vlasnika, za njegove saradnike i za ostale korisnike
- podatak da li je datoteka zaključana ili ne
- SUID podatak da li numerička oznaka vlasnika datoteke postaje numerička oznaka vlasnika procesa stvorenog na osnovu sadržaja datoteke (važi samo za izvršne datoteke)
- datum poslednje izmene datoteke
- Činjenica, da deskriptor datoteke sadrži prava pristupa datoteci, podrazumeva da je sadržaj masovne memorije fizički zaštićen.
- To podrazumeva da se centralni delovi računara nalaze u zaštićenoj (sigurnoj) prostoriji, a da su samo periferni delovi računara direktno na raspolaganju korisnicima.
- Kada to nije moguće, alternativa je da sadržaj masovne memorije bude kriptovan.
- Podatak da li je datoteka zaključana ili ne se čuva u kopiji deskriptora u radnoj memoriji.
- Ova kopija nastane prilikom otvaranja datoteke.
- Podatak da li je datoteka zaključana ili ne je uveden radi ostvarenja međusobne isključivosti procesa u toku pristupa datoteci.
- Pri tome se podrazumeva da su aktivnosti ovih procesa međusobno isključive i u toku obavljanja operacije zaključavanja datoteke.
- U ovoj operaciji se proverava da li je datoteka zaključana i eventualno obavi njen zaključavanje.
- Sinhronizacija procesa u toku obavljanja ove operacije je neophodna, da bi se spričilo da dva ili više procesa istovremeno zaključe da je ista datoteka otključana i da, nezavisno jedan od drugog, istovremeno zaključaju pomenutu datoteku.
- Pomenuta sinhronizacija obezbeđuje da uvek najviše jedan proces zaključa datoteku, jer samo on pronalazi otključanu datoteku, dok svi preostali istovremeno aktivni procesi pronalaze zaključanu datoteku.
- Ako je za nastavak aktivnosti ovih preostalih procesa neophodno da pristupe datoteci, tada se njihova aktivnost zaustavlja do otključavanja datoteke.

- Njeno izvršavanje omogućuje nastavak aktivnosti samo jednog od procesa, čija aktivnost je zaustavljena do otključavanja datoteke.
- Ako takav proces postoji, datoteka se i ne otključava, nego se samo prepušta novom procesu. Inače, datoteka se otključava.
- I operacija otključavanja datoteke zahteva sinhronizaciju procesa.
- U slučaju zaključavanja datoteke, moguće je da proces nastavi svoju aktivnost i nakon neuspelnog pokušaja zaključavanja datoteke.
- Jasno, tada se podrazumeva da on neće pristupati pomenutoj datoteci.
- Prema tome, operacija zaključavanja datoteke je blokirajuća, kada, radi uslovne sinhronizacije, u toku njenog obavljanja dolazi do zaustavljanja aktivnosti procesa, dok se ne stvore uslovi za međusobno isključive pristupe zaključanoj datoteci.
- Ova operacija je neblokirajuća, kada njena povratna vrednost ukazuje na neuspeli pokušaj zaključavanja datoteke i na nemogućnost pristupa datoteci, koju je zaključao neki drugi proces.
- Sinhronizaciju procesa moraju da obezbede ne samo operacije zaključavanja i otključavanja datoteke, nego i sve druge operacije za rukovanje deskriptorima datoteka.
- Jedino tako se može obezrediti očuvanje konzistentnosti deskriptora.

## **Imenici**

- Ime datoteke je prirodno vezano za njen deskriptor.
- Pošto se ime datoteke nalazi u imeniku, uz njega bi, u imenik, mogao da bude smešten i deskriptor datoteke.
- Međutim, tipičan način korišćenja imenika je njihovo pretraživanje, radi pronalaženja imena imenika ili imena datoteke, navedene u dатој putanji.
- Ovakvo pretraživanje prirodno prethodi pristupu sadržaju datoteke, odnosno sadržaju imenika.
- Brzina tog pretraživanja je veća što su imenici kraći, jer se tada manje podataka prebacuje između radne i masovne memorije.

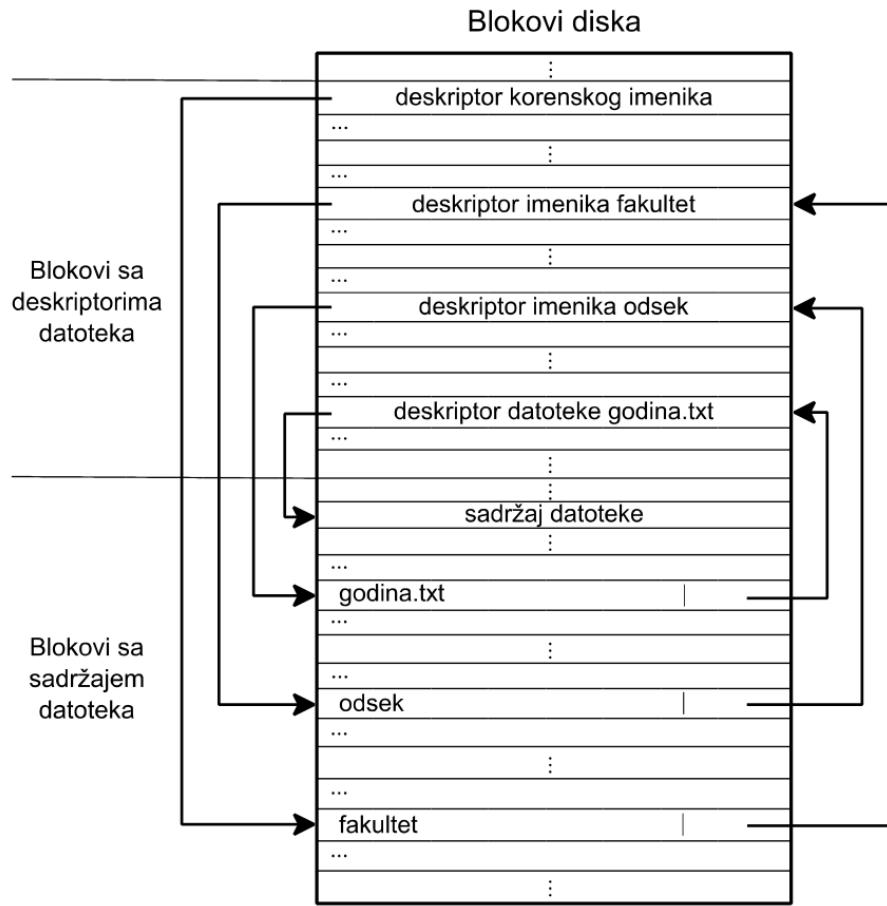
- Znači uputno je da imenici sadrže samo imena datoteka i imenika, a ne i njihove deskriptore, pogotovo ako su deskriptori veliki.
- Zato se deskriptori (inodes) čuvaju na disku van imenika.
- Da bi se uspostavila veza između imena datoteka, odnosno imena imenika sa jedne strane i njihovih deskriptora sa druge strane, u imenicima se, uz imena datoteka, odnosno uz imena imenika, navode i redni brojevi njihovih deskriptora, koji jednoznačno određuju ove deskriptore.
- Prema tome, imenik je datoteka koja sadrži tabelu u čijim elementima su imena datoteka (odnosno, imena imenika) i redni brojevi njihovih deskriptora

Imena datoteka (imenika)	Redni brojevi deskriptora datoteka (imenika)
.	.
.	.
.	.

- Iz rednog broja deskriptora se može odrediti redni broj bloka masovne memorije, u kome se deskriptor nalazi, ako se izvestan broj susednih blokova rezerviše samo za smeštanje deskriptora.
- Pod pretpostavkom da blok sadrži celi broj deskriptora, količnik rednog broja deskriptora i ukupnog broja deskriptora u bloku određuje redni broj bloka sa deskriptorom.
- Pri tome se podrazumeva da je deskriptor sa rednim brojem 0 rezervisan za korenski imenik.
- Zahvaljujući ovoj pretpostavci, moguće je uvek pronaći deskrip-

tor korenskog imenika i od njega započeti pretraživanje imenika, što obavezno prethodi pristupu sadržaju datoteke.

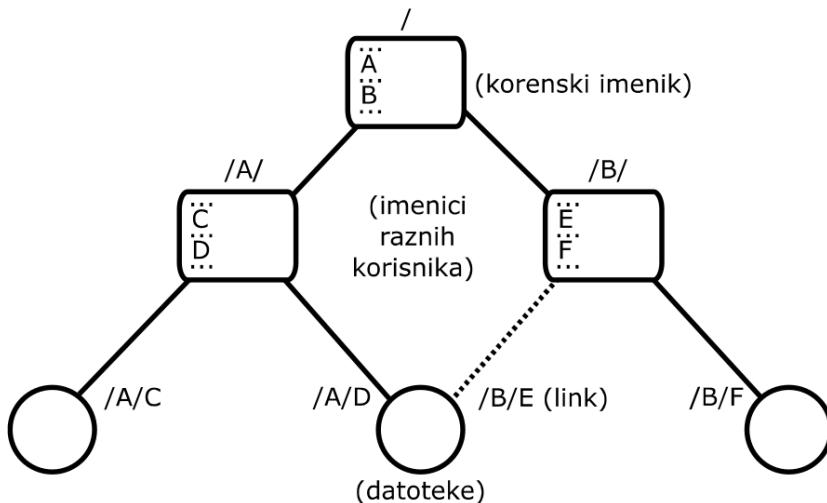
- Tako, na primer, za pristup sadržaju datoteke sa putanjom:
  - /fakultet/odsek/godina.txt
  - potrebno je prebaciti u radnu memoriju blok sa deskriptorom korenskog imenika, koji je poznat, zahvaljujući činjenici da je redni broj (0) deskriptora korenskog imenika unapred zadan.
  - U deskriptoru korenskog imenika se nalaze redni brojevi blokova sa sadržajem korenskog imenika.
  - Nakon prebacivanja ovih blokova u radnu memoriju, moguće je pretražiti sadržaj korenskog imenika, da bi se ustanovilo da li on sadrži ime fakultet.
  - Ako sadrži, uz ovo ime je i redni broj deskriptora odgovarajućeg imenika, iz koga se može odrediti redni broj bloka u kome se nalazi ovaj deskriptor.
  - Po prebacivanju ovog bloka u radnu memoriju, u pomenutom deskriptoru se pronalaze redni brojevi blokova sa sadržajem imenika fakultet.
  - Nakon prebacivanja ovih blokova u radnu memoriju, moguće je pretražiti sadržaj i ovog imenika, da bi se ustanovilo da li on sadrži ime odsek.
  - Ako sadrži, uz ovo ime je i redni broj deskriptora odgovarajućeg imenika, iz koga se može odrediti redni broj bloka u kome se nalazi ovaj deskriptor.
  - Po prebacivanju ovog bloka u radnu memoriju, u pomenutom deskriptoru se pronalaze redni brojevi blokova sa sadržajem imenika odsek.
  - Nakon prebacivanja ovih blokova u radnu memoriju moguće je pretražiti sadržaj i ovog imenika, da bi se ustanovilo da li on sadrži ime datoteke godina.txt.
  - Ako sadrži, uz nju je i redni broj deskriptora odgovarajuće datoteke, iz koga se može odrediti redni broj bloka u kome se nalazi ovaj deskriptor.
  - Po prebacivanju ovog bloka u radnu memoriju, u pomenutom deskriptoru se pronalaze redni brojevi blokova sa sadržajem datoteke godina.txt.
  - Tek tada je moguć pristup ovom sadržaju.



- U toku pristupa imenicima, neophodno je njihovo zaključavanje, radi obezbeđenja međusobne isključivosti pristupa raznih procesa istom imeniku.
- Za imenike je važno pitanje da li ista datoteka može istovremeno biti registrovana u dva ili više imenika.
- Ako se to dozvoli, tada razne putanje mogu voditi do istog sadržaja.
- To je efikasan način da vlasnik, ali i više drugih korisnika istu datoteku mogu naći svaki u svom imeniku, a da ne moraju praviti sopstvenu kopiju datoteke.
- Pri tome, svaki od drugih korisnika može dotičnoj datoteci dati novo ime, koje se naziva link (link).
- Slika sadrži prikaz tri imenika (predstavljeni kvadratima) i tri

datoteke (predstavljene krugovima).

- Do srednje datoteke vode dve putanje (link je predstavljen isprekidanom linijom).



- U imeniku uz link može biti naveden redni broj deskriptora odgovarajuće datoteke (hard link), ali može biti navedena putanja datoteke njenog vlasnika (soft link).
- U prvom slučaju, deskriptor datoteke mora da sadrži broj linkova.
- Ako se dozvoli da i imenici imaju linkove, tada postaju mogući ciklusi u hijerarhijskoj organizaciji datoteka (jer imenik može sadržati svoj link ili link imenika sa višeg nivoa hijerarhije).

## Sistemske operacije sloja za rukovanje datotekama

- Prethodno opisano pretraživanje imenika se dešava u okviru izvršavanja sistemske operacije otvaranja datoteke (`open()`).
- Zato je putanja datoteke obavezni argument poziva ove operacije.
- Njeno izvršavanje prebacuje kopiju deskriptora datoteke u radnu memoriju u tabelu deskriptora datoteka.
- Ova kopija se ne uključuje u deskriptor procesa, u toku čije aktivnosti je inicirano njeno prebacivanje, jer ista datoteka može istovremeno biti otvorena u toku aktivnosti više procesa.
- Da bi svaki od njih koristio istu kopiju deskriptora datoteke, u deskriptoru svakog procesa postoji tabela otvorenih datoteka.

- Svaki njen element sadrži adresu kopije deskriptora odgovarajuće datoteke iz tabele deskriptora datoteka.
- Indeks elementa tabele otvorenih datoteka (u kome je adresa kopije deskriptora otvorene datoteke) predstavlja povratnu vrednost poziva sistemske operacije otvaranja datoteke.
- Ovaj indeks otvorene datoteke se koristi kao argument u pozivima drugih sistemskih operacija sloja za rukovanje datotekama.
- On određuje datoteku na koju se pomenuti poziv odnosi.
- Od veličine tabele otvorenih datoteka zavisi najveći mogući broj istovremeno otvorenih datoteka nekog procesa.
- Kao dodatni argument poziva sistemske operacije otvaranja datoteke može se javiti oznaka nameravane vrste pristupa otvaranoj datoteci, koja pokazuje da li se datoteka otvara samo za čitanje, ili za pisanje/čitanje.
- U toku izvršavanja sistemske operacije otvaranja datoteke proverava se da li proces, koji poziva ovu operaciju, poseduje pravo nameravanog pristupa otvaranoj datoteci.
- Za ovo se koristi, sa jedne strane, numerička oznaka vlasnika procesa iz deskriptora procesa, a sa druge strane, numerička oznaka vlasnika otvarane datoteke, kao i prava pristupa otvaranoj datoteci za njenog vlasnika, za njegove saradnike i za ostale korisnike, što sadrži deskriptor otvarane datoteke.
- Otvaranje datoteke je uspešno samo ako proces poseduje pravo nameravanog pristupa datoteci.
- Tada poziv sistemske operacije otvaranja datoteke vraća indeks otvorene datoteke. Inače, on vraća kod greške.
- Oznaka vrste nameravanog pristupa datoteci se čuva u posebnom polju odgovarajućeg elementa tabele otvorenih datoteka, radi naknadne provere ispravnosti pristupa datoteci.
- Pokušaj otvaranja nepostojeće datoteke dovodi do njenog stvaranja, ako se tako navede u argumentima poziva sistemske operacije otvaranja datoteke.
- Alternativa je da postoji posebna operacija (creat()) za stvaranje datoteke (argument poziva ove operacije bi bila putanja stvarane datoteke).
- Korišćenje datoteke se završava pozivom sistemske operacije zatvaranja datoteke (close()).

- Ona čisti odgovarajući element tabele otvorenih datoteka procesa, koji je pozvao ovu sistemsku operaciju, i prebacuje, eventualno, kopiju deskriptora i baferovane kopije blokova sadržaja zatvarane datoteke u masovnu memoriju (što je neophodno samo ako su ove kopije izmenjene).
- Kopija deskriptora zatvarane datoteke ostaje u radnoj memoriji, ako, pored procesa koji zatvara datoteku, postoje i drugi procesi koji joj pristupaju.
- Zato kopija deskriptora datoteke sadrži broj procesa koji pristupaju datoteci.
- Obavezni argument poziva sistemske operacije zatvaranja datoteke je indeks otvorene datoteke.
- Važno je uočiti da u periodu dok je datoteka otvorena, znači, između uzastopnih poziva sistemskih operacija otvaranja i zatvaranja datoteke, proces nije pod uticajem izmena prava pristupa otvorenoj datoteci, jer izmena prava pristupa postaje delovna tek pri narednom otvaranju datoteke, pošto se tek tada ova prava ponovo proveravaju.
- Korisna praksa je da se, na kraju aktivnosti procesa, automatski zatvore sve otvorene datoteke.
- Otvorena datoteka se zaključava kada je neophodno ostvariti međusobnu isključivost u toku pristupa njenom sadržaju.
- Datoteka se otključava kada prestane potreba za međusobnom isključivošću u toku rukovanja njenim sadržajem.
- Zaključavanje i otključavanje datoteke se nalazi u nadležnosti posebne sistemske operacije (`flock()`) koja rukuje kopijom deskriptora zaključavane/ otključavane datoteke.
- U opštem slučaju, operacija zaključavanja datoteke može biti blokirajuća ili neblokirajuća.
- U prvom slučaju aktivnost procesa se zaustavlja tokom pokušaja zaključavanja datoteke, dok zaključavanje ne postane moguće.
- U drugom slučaju, ako zaključavanje datoteke nije moguće, poziv sistemske operacije zaključavanja datoteke vraća kod greške koji objašnjava razlog neuspeha u zaključavanju datoteke.
- Obavezni argument poziva sistemske operacije zaključavanja datoteke je indeks otvorene datoteke.
- Nakon otvaranja, sadržaj datoteke se može čitati pozivom sistemske operacije čitanja datoteke (`read()`) i pisati pozivom sis-

temske operacije pisanja datoteke (write()), ako je to u skladu sa namerama, izraženim u otvaranju datoteke.

- Obavezni argumenti ovih poziva su indeks otvorene datoteke i broj bajta (koji se čitaju ili pišu).
- Pored toga, poziv sistemske operacije čitanja sadrži, kao argument, adresu zone radne memorije u koju se smeštaju pročitani bajti, a poziv sistemske operacije pisanja sadrži, kao argument, adresu zone radne memorije iz koje se preuzimaju bajti za pisanje.
- Oba poziva vraćaju vrednost, koja ukazuje na uspešan poziv ili na grešku.
- Podrazumeva se da sistemske operacije pisanja i čitanja nude sekvensijalan pristup datotekama.
- To znači, da, ako jedan poziv sistemske operacije čitanja (pisanja) pročita (upiše) prvi bajt datoteke, naredni takav poziv datoteke će da pročita (upiše) drugi bajt datoteke.
- Radi podrške sekvensijalnom pristupu, svaki element tabele otvorenih datoteka sadrži i posebno polje pozicije u datoteci sa rednim brojem bajta od koga se primenjuje naredno čitanje ili pisanje.
- Svako čitanje ili pisanje pomera poziciju na prvi naredni nepročitani (neupisani) bajt.
- Da bi bili mogući direktni pristupi bajtima datoteke (u proizvoljnem redosledu), postoji sistemska operacija izmene pozicije u datoteci (seek()).
- Obavezni argumenti njenog poziva su indeks otvorene datoteke i podatak o novoj poziciji, dok povratna vrednost ovog poziva ukazuje da li je poziv bio uspešan ili ne.
- Tako, na primer, ako se želi pisati na kraj datoteke, neophodno je prvo pozvati sistemsku operaciju izmene pozicije u datoteci, radi pozicioniranja iza poslednjeg bajta datoteke, i zatim pozvati sistemsku operaciju pisanja.
- Prvi poziv izmeni poziciju u datoteci u odgovarajućem elementu tabele otvorenih datoteka.
- Drugi poziv, na osnovu ove pozicije i kopije deskriptora datoteke, odredi redni broj bloka, ako on postoji, i prebací njegovu kopiju u radnu memoriju, ako ona već nije bila prisutna u radnoj memoriji.

- U ovu kopiju se smeste dopisivani bajti i ona se prebac i (odmah ili kasnije, zavisno od strategije baferovanja) u masovnu memoriju.
- Ako blok ne postoji, ili ako se upisivanje proteže na više blokova, upisivanju bajta prethodi zauzimanje blokova.
- Radi toga se menja (proširuje) tabela pristupa datoteke, što može da dovede i do izmene njenog deskriptora.
- Nakon toga se u radnoj memoriji oblikuje novi sadržaj blokova i oni se prebacuju u masovnu memoriju.
- Sloj za rukovanje datotekama nudi posebne sistemske operacije za izmenu atributa datoteke, sadržanih u njenom deskriptoru (kao što su numerička oznaka vlasnika datoteke, prava pristupa datoteci za njenog vlasnika, za njegove saradnike i za ostale korisnike, ili SUID podatak da li numerička oznaka vlasnika datoteke postaje numerička oznaka vlasnika procesa, stvorenog na osnovu sadržaja datoteke).
- Obavezni argumenti poziva ovih sistemskih operacija su putanja datoteke i nova vrednost menjanog atributa datoteke.
- Njihova povratna vrednost ukazuje na uspešnost poziva.
- U okviru izvršavanja ovih sistemskih operacija pretražuju se imenici, radi prebacivanja u radnu memoriju kopije deskriptora datoteke.
- Zatim se proverava da li je ove sistemske operacije pozvao vlasnik datoteke, jer jedino on ima pravo da menja attribute datoteke.
- Ako jeste, tada se menja zadani atribut i blok sa izmenjenom kopijom deskriptora vraća u masovnu memoriju.
- U sistemske operacije za izmenu atributa datoteke spada i sistema operacija za izmenu imena datoteke.
- Iako se u okviru ove operacije ne menja deskriptor datoteke, nego njen imenik, ovde je, pored pristupa deskriptoru njenog imenika, neophodan i pristup deskriptoru dotočne datoteke, radi provere da li je ovu operaciju pozvao vlasnik datoteke.
- Za stvaranje linka (dodatnog imena datoteke) potrebna je posebna sistema operacija (link()).
- Obavezani argumenti njenog poziva su putanja sa imenom datoteke i putanja sa linkom. U okviru ove operacije se pretražuju imenici radi ubacivanja linka.

- Za uništenje datoteke neophodna je posebna sistemska operacija (unlink()).
- Obavezni argument njenog poziva je putanja uništavane datoteke.
- U okviru sistemske operacije uništenja datoteke, pretražuju se imenici, radi prebacivanja u radnu memoriju kopije deskriptora uništavane datoteke.
- Zatim se proverava da li je broj linkova za ovu datoteku nula i da li je ovu sistemsku operaciju pozvao vlasnik datoteke, pa ako jeste, oslobođaju se blokovi datoteke i njen deskriptor.
- Na kraju se poništava ime datoteke u odgovarajućem imeniku, što predstavlja izmenu njegovog sadržaja.
- Povratna vrednost ove operacije ukazuje da li je ona uspešno obavljena.
- Na primer, uništenje nepostojeće datoteke ne može biti uspešno obavljeno.
- Prethodno opisane sistemske operacije omogućuju pristup svim datotekama, znači i imenicima.
- Ali za pristupe imenicima je potrebno poznavati detalje njihove organizacije, kao što su:
  - broj bajta koji je rezervisan za imena datoteka i imenika
  - broj bajta koji je rezervisan za redne brojeve njihovih deskriptora
  - oznaku koja označava prethodni imenik u hijerarhiji i slično
- Potreba za poznavanjem organizacije imenika se izbegava, ako se ponude posebne sistemske operacije za rukovanje imenicima, kao što su operacije za stvaranje imenika, za pregledanje i za izmene njegovog sadržaja, ili za uništenje imenika.
- Među sistemske operacije za rukovanje imenicima spada posebna operacija (mount()) koja omogućuje spajanje dva sistema datoteka, tako što korenski imenik jednog od sistema datoteka postane (zameni) običan imenik drugog sistema datoteka.
- Postoji i suprotna operacija (umount()) koja razdvaja dva prethodno spojena sistema datoteka.

## **Specijalne datoteke**

- Važno svojstvo pojma datoteke je da je on primenljiv i za opisivanje ulaznih i izlaznih uređaja.
- Tako, ulazni uređaj, kao što je tastatura, odgovara datoteci, čiji sadržaj se može samo čitati, a sastoji se od bajta, koji stižu sa tastature.
- Slično, izlazni uređaj, kao što je ekran, odgovara datoteci, čiji sadržaj se može samo pisati, a sastoji se od bajta, koji se upućuju na ekran.
- Takođe, ulazno izlazni uređaj, kao što je disk, odgovara datoteci, čiji sadržaj se može i pisati i čitati, a sastoji se od bajta iz pojedinih blokova diska.
- Datoteke, koje predstavljaju pojedine ulazne ili izlazne uređaje, se nazivaju specijalne datoteke (special file).
- Specijalne datoteke se dele na znakovne, koje odgovaraju uređajima kao što su tastatura, ekran ili štampač, i na blokovske, koje odgovaraju, na primer, diskovima.
- Znakovne specijalne datoteke podržavaju sekvencijalno čitanje ili pisanje znakova (koji dolaze sa odgovarajućeg uređaja, ili odlaže na odgovarajući uređaj).
- Za ove datoteke izmena pozicije nema smisla.
- Blokovske specijalne datoteke podržavaju čitanje ili pisanje blokova.
- Za njih izmena pozicije omogućuje određivanje rednog broja bloka na koga se primenjuje naredna operacija.
- Blokovske specijalne datoteke omogućuju direktnе pristupe blokovima diska, što je važno, na primer, kod pronalaženja izgubljenih blokova, kod sabiranja datoteka, ili kod pripremanja disk jedinica za korišćenje.
- U poslednjem slučaju se određuje namena pojedinih blokova diska:

<b>blok 0</b>	<b>prvi (<i>boot</i>) blok</b>
<b>blok 1</b>	<b>drugi (<i>super</i>) blok</b>
<b>blok 2</b>	<b>blokovi namenjeni za deskriptore</b>
...	
<b>blon n</b>	<b>blokovi namenjeni za sadržaj datoteka i za tabele pristupa</b>
...	

- Prvi (boot) blok je rezervisan za podatke, koji su potrebni za pokretanje računara
- (operativnog sistema), a drugi (super) blok sadrži:
  - podatke o nizu susednih blokova, koji su namenjeni za smeštanje deskriptora datoteka (ovi podaci obuhvataju redni broj prvog bloka iz ovog niza, kao ukupan broj blokova u ovom nizu)
  - podatke o slobodnim mestima za deskriptore datoteka
  - podatke potrebne za evidenciju (preostalih) slobodnih blokova (u ovu evidenciju se uključuju svi blokovi, koji nisu upotrebljeni za smeštanje sadržaja datoteka ili za tabele pristupa).
- Blokovska specijalna datoteka nije nužno vezana za jedan celi disk.
- Ona se može odnositi na deo diska, koji se naziva particija (partition) i koji tada predstavlja logičku disk jedinicu (logical volume).
- Da bi to bilo moguće, neophodno je raspolažati sredstvima za rukovanje particijama.
- Rukovanje particijama dozvoljava i formiranje logičkih disk jedinica koje obuhvataju više particija na raznim fizičkim diskovima.
  - To omogućava:
    - brži pristup blokovima logičke disk jedinice, jer istovremeno mogu biti prebacivani blokovi iz raznih particija sa raznih fizičkih diskova (striping, RAID0),
    - veću pouzdanost logičke disk jedinice, jer svaki blok može biti repliciran tako da svaka particija sa raznih fizičkih diskova sadrži potpunu kopiju logičke disk jedinice (mirroring, RAID1)
    - oboje (RAID5).

- Pre korišćenja specijalnih datoteka, neophodno je njihovo otvaranje (radi provere prava pristupa uređaju, koga datoteka predstavlja), i eventualno njihovo zaključavanje (radi ostvarenja međusobne isključivosti pristupa uređaju, koga datoteka predstavlja).
- Nakon korišćenja, sledi eventualno otključavanje i zatvaranje specijalne datoteke.
- Za podršku otvaranja i zatvaranja, odnosno zaključavanja i otključavanja specijalnih datoteka, neophodno je da one poseduju svoje deskriptore.
- Deskriptori specijalnih datoteka obuhvataju atributе, као што су, на primer numeričка ознака власника датотеке, права приступа датотeci за њеног власника, за његове сараднике и за остale кориснике, или податак да ли је датотека заклjučана или не.
- Међутим, за операције чitanja ili pisanja specijalnih datoteka nije потребно preslikavanje rednih brojeva bajta u redne brojeve blokova masovne memorije.
- Umesto тога за njih je потребно pozivanje одговарајућих операција драјвера uređaja, које ове датотеке представљају.
- Зато се у deskriptorima specijalnih датотека не налазе подаци о табели приступа, него подаци о одговарајућем драјверу и примерку uređaja, кога он опслужује.
- За jednoznačno идентификовање драјвера уводи се redni broj драјвера (major number), који služi као indeks за poseбну табелу драјвера.
- Полja indeksiranog елемента ове табеле садрže адресе операција дотичног драјвера.
- Prema tome, ако се redni број драјвера чува у deskriptoru specijalне датотеке, на основу њега је могуће прonaći адресе операција овог драјвера (posredstvom одговарајућег елемента табеле драјвера).
- У поменутом deskriptoru се чува и redni број uređaja (minor number) koga predstavlja specijalna datoteka, da bi se na njega mogla usmeriti odabrana operacija драјвера.
- За измену atributa specijalnih датотека применљиве су sistemske операције, које су уведене с том нamerом за обичне датотеке.
- Isto važi i za sistemske операције за измену имена датотеке i за njeno uništenje.

- Pri tome, kod uništenja specijalne datoteke, nema oslobođanja blokova, nego se oslobođa samo njen deskriptor.
- Sistemske operacije prave razliku između običnih i specijalnih datoteka na osnovu posebne oznake, navedene u deskriptoru datoteke.

## **Standardni ulaz i standardni izlaz**

- Pojmovi datoteke i procesa su čvrsto povezani, jer je aktivnost procesa posvećena obradi podataka, sadržanih u datotekama.
- Pri tome je tipično da obrađivani podaci stižu u proces iz jedne, ulazne datoteke, a da obrađeni podaci napuštaju proces, završavajući u drugoj, izlaznoj datoteci.
- Ovakav model obrade podataka je dovoljno čest, da opravda uvođenje naziva standardni ulaz (standard input) za ulaznu datoteku i standardni izlaz (standard output) za izlaznu datoteku.
- Pri tome se podrazumeva da se u toku stvaranja procesa otvore i njegov standardni ulaz i njegov standardni izlaz. Zahvaljujući tome, bez posebnog otvaranja se može čitati standardni ulaz i pisati standardni izlaz.
- Kao podrazumevajući standardni ulaz služi specijalna datoteka, koja predstavlja tastaturu, a kao podrazumevajući standardni izlaz služi specijalna datoteka, koja predstavlja ekran.
- U slučaju da proces stvaralač zaustavlja svoju aktivnost, dok traje aktivnost stvorenog procesa, tada je prirodno da stvoren proces nasledi standardni ulaz i standardni izlaz od procesa stvaraoca i da tako, preuzimajući opsluživanje terminala, nastavi interakciju sa korisnikom.
- Kao indeks otvorene datoteke, koja odgovara standardnom ulazu, može da služi vrednost 0, a kao indeks otvorene datoteke, koja odgovara standardnom izlazu, može da služi vrednost 1.

## **Spašavanje datoteka**

- U nadležnosti sloja za rukovanje datotekama nalazi se i podrška spašavanju (backup) datoteka, čiji cilj je da se redovno prave kopije postojećih (svih, ili samo u međuvremenu izmenjenih, odnosno stvorenih) datoteka.
- Na osnovu ovakvih kopija moguće je rekonstruisati (restore) sadržaj oštećenih datoteka.

- Do oštećenja datoteka dolazi na razne načine, kao što je, na primer, pojava loših (neispravnih) blokova.
- Kada se otkriju, loši blokovi se izbacuju iz upotrebe. Jedan način da se to postigne je da se, na primer, formira datoteka loših blokova.

## **Osnova sloja za rukovanje datotekama**

- Sloj za rukovanje datotekama se oslanja na operacije drajvera iz sloja za rukovanje kontrolerima.
- On može da koristi i operacije sloja za rukovanje radnom memorijom, radi zauzimanja bafera, namenjenih za smeštanje kopija blokova, ili kopija deskriptora datoteka, na primer.

## **Pitanja**

### **Pitanja**

- Na šta ukazuje ime datoteke?
- Od koliko delova se sastoji ime datoteke?
- Od koliko delova se sastoji ime imenika?
- Šta obuhvata rukovanje datotekom?
- Šta karakteriše hijerarhijsku organizaciju datoteka?
- Šta važi za apsolutnu putanju?
- Šta važi za relativnu putanju?
- Koje datoteke obrazuju sistem datoteka?
- Koja su prava pristupa datotekama?
- Koje kolone ima matrica zaštite?
- Čemu je jednak broj redova matrice zaštite?
- Gde se mogu čuvati prava pristupa iz matrice zaštite?
- Šta je potrebno za sprečavanje neovlašćenog menjanja matrice zaštite?
- Kada korisnici mogu posredno pristupiti spisku lozinki?
- Koju dužnost imaju administratori?
- Šta sadrži numerička oznaka korisnika?
- Kakvu numeričku oznaku imaju saradnici vlasnika datoteke?

- Kakvu numeričku oznaku imaju ostali korisnici?
- Kada se obavlja provera prava pristupa datoteci?
- Čime se bavi sigurnost?
- Kako se predstavlja sadržaj datoteke?
- Gde se javlja interna fragmentacija?
- Šta karakteriše kontinualne datoteke?
- Koji oblik evidencije slobodnih blokova masovne memorije je podesan za kontinualne datoteke?
- Šta je eksterna fragmentacija?
- Šta karakteriše rasute datoteke?
- Šta karakteriše tabelu pristupa?
- Šta ulazi u sastav tabele pristupa?
- Kada rasuta datoteka ne zauzima više prostora na disku od kontinualne datoteke?
- Koji oblik evidencije slobodnih blokova masovne memorije je podesan za rasute datoteke?
- Kada dolazi do gubitka blokova prilikom produženja rasute datoteke?
- Kada dolazi do višestrukog nezavisnog korišćenje istog bloka prilikom produženja rasute datoteke?
- Kada pregled izmena ukazuje da je sistem datoteka u konzistentnom stanju?
- Kako se ubrzava pristup datoteci?
- Od čega zavisi veličina bloka?
- Šta sadrži deskriptor kontinualne datoteke?
- Kako se rešava problem eksterne fragmentacije?
- Kako se ublažava problem produženja kontinualne datoteke?
- Šta sadrži deskriptor rasute datoteke?
- Šta je imenik?
- Šta karakteriše specijalne datoteke?
- Šta sadrži deskriptor specijalne datoteke?
- Šta omogućuju blokovske specijalne datoteke?

- Šta omogućuje rukovanje particijama?— title: Memorija author: Veljko Petrović date: 2023-05-11 —

## **Memorija**

### **Zadatak sloja za rukovanje radnom memorijom**

- Sloj za rukovanje radnom memorijom rukuje fizičkom radnom memorijom.
- Fizičkoj radnoj memoriji se pristupa posredstvom fizičkog adresnog prostora (koga koristi operativni sistem), ali i preko logičkih adresnih prostora (po jedan za svaki korisnički proces).
- Logički adresni prostori izoluju procese (međusobno i od operativnog sistema) tako što ograničavaju pristup procesa samo na deo fizičke radne memorije.
- To se postiže preslikavanjem logičkog adresnog prostora svakog od procesa na samo jedan deo fizičkog adresnog prostora, koji odgovara delu fizičke memorije, dodeljene dotičnom procesu.
- Preslikavanje obavlja MMU (Memory Management Unit) tako što logičke adrese pretvara (translira) u odgovarajuće fizičke adrese.
- Funkcionisanje i organizacija MMU zavisi od karaktera logičkog adresnog prostora. Logički adresni prostor može biti:
  - Kontinualan
  - Sastavljen od segmenata raznih veličina
  - Sastavljen od stranica iste veličine
  - Sastavljen od segmenata raznih veličina koji se sastoje od stranica iste veličine

### **Kontinualni logički adresni prostor**

- Kontinualni logički adresni prostor se sastoји od jedног низа узастопних логичких адреса, који почиње од логичке адресе 0.
- Величина континуалног логичког адресног простора надмаšује потребе прозећног процеса.
- Зато, да би се могао детектовати покушај излaska процеса из неговог логичког адресног простора, неophodno je poznavati највишу исправну логичку адресу процеса.

- Ona se zove granična adresa procesa. Logičke adrese procesa ne smeju biti veće od njegove granične adrese.
- Za kontinualni logički adresni prostor se podrazumeva da se niz uzastopnih logičkih adresa preslikava u niz uzastopnih fizičkih adresa.
- Ova dva niza se razlikuju po tome što niz uzastopnih fizičkih adresa ne počinje od 0 nego od neke adresu koja se zove bazna adresa.
- Znači, dodavanjem bazne adrese logičkoj adresi nastaje fizička adresa.
- U toku translacije logičke adrese u fizičku, MMU poredi logičku i graničnu adresu. Ako je logička adresa veća, MMU izaziva izuzetak, inače sabira baznu adresu sa logičkom adresom da bi dobio fizičku adresu.

### **Segmentirani logički adresni prostor**

- Segmentirani logički adresni prostor (segmentation) se sastoji od više nizova uzastopnih logičkih adresa (za svaki segment po jedan niz).
- Da bi se znalo kom segmentu pripada logička adresa, njeni značajniji biti sadrže adresu njenog segmenta, a manje značajni biti sadrže unutrašnju adresu u njenom segmentu.
- Pošto je svaki segment kontinualan, on ima svojstva kontinualnog logičkog adresnog prostora. Zato svaki segment karakterišu njegova granična i bazna adresa.
- U ovom slučaju za translaciju je potrebna tabela segmenata.
- Broj njenih elemenata određuje najveći broj segmenata u segmentiranom logičkom adresnom prostoru.
- Svaki element ove tabele sadrži graničnu i baznu adresu odgovarajućeg segmenta.
- Adresa segmenta indeksira element tabele segmenata, a njegove graničnu i baznu adresu koristi MMU za translaciju unutrašnje adrese segmenta u fizičku adresu.

### **Stranični logički adresni prostor**

- Stranični logički adresni prostor (paging) se sastoji od jednog niza uzastopnih logičkih adresa (podeljenog u stranice iste veličine).

- Da bi se znalo kojoj stranici pripada logička adresa, njeni značajniji biti sadrže adresu njene stranice, a manje značajni biti sadrže unutrašnju adresu u njenoj stranici.
- Pošto je svaka stranica kontinualna, ona ima svojstva kontinualnog logičkog adresnog prostora.
- Razlika je da je veličina stranice unapred poznata i jednaka stepenu broja dva.
- Zato svaku stranicu karakteriše samo njena bazna adresa (granična adresa nije potrebna, jer je unutrašnja adresa ograničena veličinom stranice).
- U ovom slučaju za translaciju je potrebna tabela stranica.
- Broj njenih elemenata jednak je najvećem broju stranica u straničnom logičkom adresnom prostoru.
- Svaki element ove tabele sadrži baznu adresu odgovarajuće stranice. Adresa stranice indeksira element tabele stranica, a njegovu baznu adresu koristi MMU za translaciju logičke adrese stranice u fizičku adresu.

### **Odnos procesa i logičkog adresnog prostora**

- Svaki proces mora da ima neophodne translacione podatke koji obuhvataju ili graničnu i baznu adresu, ili tabelu segmenata, ili tabelu stranica, ili tabelu segmenata sa pripadnim tabelama stranica.
- Translacione podatke pripremi operativni sistem, prilikom stvaranja procesa. Da bi mogao da obavlja translaciju, MMU mora da raspolaže sa odgovarajućim translacionim podacima.
- Translacioni podaci se menjaju prilikom preključivanja, pa utiču na trajanje preključivanja.
- Veličina slike procesa određuje veličinu njegovog logičkog adresnog prostora.

### **Upotreba kontinualnog logičkog adresnog prostora**

- Kontinualni logički adresni prostor se koristi kada je logički adresni prostor svakog procesa manji od raspoloživog fizičkog adresnog prostora.
- Da bi translacija logičkih adresa u fizičke bila moguća, neophodno je da se u lokacije fizičke radne memorije smesti slika procesa.

- U ovom slučaju, translacija logičkih adresa u fizičke adrese odgovara dinamičkoj relokaciji i olakšava zamenu slika procesa (swapping).
- Pokušaj procesa da izade iz svog adresnog prostora izaziva izuzetak (SEGFAULT) na koji reaguje operativni sistem, tako što uništi dotični proces.

## **Upotreba segmentiranog logičkog adresnog prostora**

- Segmentacija (segmentirani logički adresni prostor) se koristi kada je važno racionalno korišćenje fizičke radne memorije.
- I ovde se prepostavlja da je logički adresni prostor procesa manji od raspoloživog fizičkog adresnog prostora, kao i da smeštanje slike procesa u lokacije fizičke radne memorije prethodi translacijsi logičkih adresa u fizičke.
- U ovom slučaju, segmenti logičkog adresnog prostora procesa sadrže delove njegove slike.
- Na primer, postoje segment za mašinske naredbe, segment za promenljive i segment za stek.
- Ako istovremeno postoji više procesa, nastalih na osnovu istog programa, tada oni mogu da dele iste mašinske naredbe.
- U ovom slučaju, tabele segmenata takvih procesa sadrže isti par granične i bazne adrese deljenog segmenta.
- Ovaj par omogućuje preslikavanje unutrašnjih adresa segmenta naredbi raznih procesa na iste fizičke adrese lokacija fizičke radne memorije sa deljenim mašinskim naredbama.
- Izdvajanje promenljivih i steka procesa u posebne segmente je korisno i zbog mogućnosti naknadnog proširenja ovih segmenata (kada to postane potrebno u toku aktivnosti procesa).
- Prednost, segmentacije je da ona ubrzava zamenu slika procesa (swapping), jer se segment naredbi ne mora izbacivati, ako je ranije već izbačen u masovnu memoriju, niti ubacivati, ako već postoji u fizičkoj radnoj memoriji.
- Jasno, rukovanje segmentima mora voditi posebnu evidenciju o segmentima, prisutnim u fizičkoj radnoj memoriji, da bi bilo moguće otkriti kada se isti segment može iskoristiti u slikama raznih procesa.
- Ovakva evidencija obuhvata jednoznačnu oznaku segmenta, podatak o broju korisnika segmenta, dužinu segmenta, odnosno,

njegovu graničnu i baznu adresu.

- Radi očuvanja konzistentnosti evidencije segmenata, operacije za rukovanje ovom evidencijom moraju obezbediti sinhronizaciju procesa.
- Prethodno opisana (osnovna) segmentacija se razvija u punu segmentaciju, ako se dozvoli da svakom potprogramu ili promenljivoj odgovara poseban segment.
- To, na primer, dozvoljava deljenje potprograma između raznih procesa, ako se segment istog potprograma uključi u slike više procesa.
- U tom slučaju, stvaraju se uslovi za dinamičko linkovanje (povezivanje) potprograma za program.
- Ono se ne dešava u toku pravljenja izvršne datoteke, nego u trenutku prvog poziva potprograma, kada se u fizičkoj radnoj memoriji pronalazi njegov segment ili se stvara ako ne postoji.
- Dinamičko linkovanje doprinosi racionalnom korišćenju masovne memorije, jer omogućava da se često korišćeni potprogrami ne multipliciraju u raznim izvršnim datotekama.
- Puna segmentacija dozvoljava i deljenje promenljivih između raznih procesa, ako se segment iste promenljive uključi u slike više procesa.
- Jasno, ovakav način ostvarenja saradnje procesa zahteva njihovu sinhronizaciju, radi očuvanja konzistentnosti deljenih promenljivih.
- Za segmente sa deljenim promenljivima su važna i prava pristupa segmentu, jer nekim od procesa treba dozvoliti samo da čitaju deljenu promenljivu, a drugima treba dozvoliti i da pišu u deljenu promenljivu.
- Zato se elementi tabele segmenata proširuju oznakom prava pristupa segmentu.
- U pogledu prava pristupa, segmenti se ne razlikuju od datoteka.
- Prema tome, prava pristupa segmentu obuhvataju pravo čitanja, pravo pisanja i pravo izvršavanja segmenta.
- Prva dva prava se primenjuju na segmente, koji sadrže promenljive, a treće pravo se primenjuje na segmente sa mašinskim naredbama programa ili potprograma.
- Na pokušaj narušavanja prava pristupa segmentu reaguje MMU, generisanjem prekida (izuzetka), što dovodi do uništenja aktivnog procesa (SEGFAULT).

- Rukovanje segmentima se obavlja posredstvom sistemskih programa, kao što su kompjaler ili linker.

## **Upotreba straničnog logičkog adresnog prostora**

- Stranični logički adresni prostor se koristi kada je logički adresni prostor tipičnog procesa veći od raspoloživog fizičkog adresnog prostora, pa slika procesa ne može da stane u fizičku radnu memoriju.
- Stranični logički adresni prostor se naziva i virtuelni adresni prostor.
- On pripada virtuelnoj memoriji i sadrži virtuelne adrese.
- Stranice virtuelnog adresnog prostora sa slikom procesa se nalaze na masovnoj memoriji.
- Postoji posebna evidencija o tome gde se one tačno nalaze na masovnoj memoriji.
- Pošto je za izvršavanje mašinske naredbe neophodno da u fizičkoj radnoj memoriji budu samo bajti njenog mašinskog formata, kao i bajti njenih operanada, u fizičkoj radnoj memoriji moraju da se nalaze samo kopije virtuelnih stranica koje sadrže pomenute bajte.
- Podrazumeva se da se kopije neophodnih virtuelnih stranica, kada zatreba, automatski prebacuju iz masovne memorije u fizičku radnu memoriju i obrnuto.
- Do prebacivanja u obrnutom smeru dolazi, kada je potrebno oslobođiti lokacije fizičke radne memorije sa kopijama u međuvremenu izmenjenih virtuelnih stranica.
- Ovo prebacivanje se obavlja, da bi se oslobođila fizička radna memorija i, istovremeno, obezbedilo da masovna memorija uvek sadrži ažurnu sliku procesa.
- Da bi se znalo koja kopija virtuelne stranice je izmenjena, neophodno je automatski registrovati svaku izmenu svake od kopija virtuelnih stranica.
- Fizička radna memorija sadrži kopije pojedinih virtuelnih stranica, pa je prirodno da i ona bude izdeljena u fizičke stranice u kojima se nalaze kopije virtuelnih stranica.
- To znači da se fizička adresa sastoji od adrese fizičke stranice (u značajnijim bitima fizičke adrese) i od unutrašnje adrese (u manje značajnim bitima fizičke adrese).

- Pošto su virtuelne i fizičke stranice iste veličine, unutrašnja adresa (manje značajni biti) virtuelne adrese se poklapa sa unutrašnjom adresom (manje značajnim bitima) fizičke adrese.
- Zato je za translaciju virtuelne adrese u fizičku potrebno samo zameniti adresu virtuelne stranice adresom fizičke stranice.
- Zbog toga se u elementu tabele stranica (koga indeksira adresa virtuelne stranice) kao bazna adresa nalazi adresa fizičke stranice (koja sadrži kopiju dotične virtuelne stranice).
- Ako virtuelna stranica nije kopirana u neku fizičku stranicu, tada translacija njenih virtuelnih adresa u fizičke nije moguća.
- U tom slučaju, to mora biti registrovano u odgovarajućem elementu tabele stranica. Takvu ulogu ima bit prisustva.
- Uz bit prisustva, elementi tabele stranica sadrže i bit referenciranja (koji pokazuje da li je bilo pristupanja kopiji virtuelne stranice) i bit izmene (koji pokazuje da li je bilo izmena kopije virtuelne stranice).
- Poslednja dva bita se automatski postavljaju.
- Kada pokušaj translacije virtuelne adrese bude neuspešan (jer bit prisustva ukaže da se kopija odgovarajuće virtuelne stranice ne nalazi u fizičkoj radnoj memoriji), MMU izaziva stranični prekid (page fault).
- U obradi straničnog prekida se prebaci kopija potrebne virtuelne stanice u neku fizičku stranicu.
- Adresa ove fizičke stranice postaje bazna adresa i smesti se u odgovarajući element tabele stranica.
- U ovom elementu se istovremeno postavi bit prisustva, a očiste se bit referenciranja i bit izmene.
- Nakon toga, ponavlja se neuspešna translacija virtuelne adrese.
- Važno je uočiti da se kopije virtuelnih stranica prebacuju na zahtev (demand paging), a ne unapred (prepaging), jer u opštem slučaju ne postoji način da se predviđi redosled korišćenja virtuelnih stranica.
- Praktična upotrebljivost virtuelne memorije se temelji na svojstvu lokalnosti izvršavanja programa.
- Zahvaljujući ovome svojstvu, za izvršavanje programa je dovoljno da u fizičkoj radnoj memoriji uvek bude samo deo progama.

## **Upotreba stranično segmentiranog logičkog adresnog prostora**

- Stranična segmentacija (stranično segmentirani logički adresni prostor) se koristi kada su segmenti potrebni radi racionalnog korišćenja fizičke radne memorije, a njihova veličina nadmašuje veličinu raspoložive fizičke radne memorije.
- U tom slučaju, svaki segment uvodi sopstveni virtuelni adresni prostor.
- Zahvaljujući tome, stranice virtuelnog adresnog prostora segmenta se nalaze u masovnoj memoriji, a u fizičkim stranicama se nalaze samo kopije neophodnih virtuelnih stranica segmenta.
- Prednost stranične segmentacije je da ona omogućava dinamičko proširenje segmenata (dodavanjem novih stranica), što je važno za segmente promenljivih i steka.
- Stranična segmentacija može otvorenim datotekama dodeljivati posebne segmente i na taj način ponuditi koncept memorijski preslikane datoteke (memory mapped file).
- Pristup ovakvoj datoteci ne zahteva sistemske operacije za čitanje, pisanje ili pozicioniranje, jer se direktno pristupa lokacijama sa odgovarajućim sadržajem datoteke.
- Mana koncepta memorijski preslikane datoteke je da se veličina datoteke izražava celim brojem stranica, jer nema načina da operativni sistem odredi koliko je popunjeno bajta iz poslednje stranice.
- Takođe, problem je i što virtuelni adresni prostor segmenta može biti suviše mali za pojedine datoteke.

## **Zadaci sloja za rukovanje fizičkom radnom memorijom**

- Zadatak sloja za rukovanje fizičkom radnom memorijom je da omogući zauzimanje zona susednih lokacija (sa uzastopnim adresama) slobodne fizičke radne memorije, kao i da omogući oslobođanje prethodno zauzetih zona fizičke radne memorije.
- Radi toga ovaj sloj nudi operacije zauzimanja i oslobođanja (fizičke radne memorije).
- Argument poziva operacije zauzimanja je dužina zauzimane zone (broj njenih lokacija), a povratna vrednost ovog poziva je adresa zauzete zone (adresa prve od njenih lokacija), ili indikacija da je operacija zauzimanja završena neuspešno.

- Argumenti poziva operacije oslobađanja su adresa oslobađane zone (adresa prve od njenih lokacija) i njena dužina (broj njenih lokacija).
- Uspešno obavljanje zadatka sloja za rukovanje fizičkom radnom memorijom se temelji na vođenju evidencije o slobodnoj fizičkoj radnoj memoriji.
- Kada podržava virtuelnu memoriju, ovaj sloj mora svakom procesu dodeliti dovoljan broj fizičkih stranica za kopije potrebnih virtuelnih stranica.

### **Raspodela fizičke radne memorije**

- Fizička radna memorija se deli na lokacije koje stalno zauzima operativni sistem i na preostale slobodne lokacije koje su na raspolaganju za stvaranje procesa i za druge potrebe.
- Operativni sistem obično zauzima lokacije sa početka i, eventualno, sa kraja fizičkog adresnog prostora, a između njih se nalaze lokacije slobodne fizičke radne memorije.
- Na početku fizičkog adresnog prostora su, najčešće, lokacije, namenjene za tabelu prekida.
- Iza njih slede lokacije sa naredbama i promenljivim operativnog sistema (koje obuhvataju i prostor za smeštanje deskriptora i sistemskih stekova procesa).
- Na kraju fizičkog adresnog prostora su, najčešće, lokacije, koje odgovaraju registrima kontrolera.
- U slučaju da procesor podržava virtuelnu memoriju, praksa je da se samo donja polovina (sa nižim adresama) virtuelnog adresnog prostora stavi na raspolaganje svakom procesu, a da gornja polovina (sa višim adresama) virtuelnog adresnog prostora bude rezervisana za operativni sistem.
- Gornja polovina virtuelnog adresnog prostora se zato može nazvati sistemski virtuelni adresni prostor, a donja polovina virtuelnog adresnog prostora se može nazvati korisnički virtuelni adresni prostor.
- Za virtuelne adrese iz prve polovine sistemskog virtuelnog adresnog prostora se ne vrši translacija (na primer, tako što se deo manje značajnih bita virtuelne adrese koristi kao fizička adresa) i podrazumeva se da se tu nalazi rezidentni deo operativnog sistema, koji je stalno prisutan u fizičkoj radnoj memoriji.

- Virtulene adrese iz druge polovine sistemskog virtuelnog adresnog prostora se transliraju na uobičajeni način i podrazumeva se da se one odnose na nerezidentni deo operativnog sistema, koji se po potrebi prebacuje u fizičku radnu memoriju.

## Evidencija slobodne fizičke radne memorije

- Sloj za rukovanje fizičkom radnom memorijom obavezno vodi evidenciju o slobodnoj fizičkoj radnoj memoriji.
- Za potrebe kontinualnog ili segmentiranog logičkog adresnog prostora ova evidencija može da bude u obliku niza bita (bit map), u kome svaki bit odgovara grupi susednih lokacija. Podrazumeva se da je broj lokacija u ovakvoj grupi unapred zadan.
- On ujedno predstavlja jedinicu u kojoj se izražava dužina zauzimane i oslobađane zone fizičke radne memorije.
- Ako je grupa lokacija slobodna, njoj odgovarajući bit sadrži 1. Inače, on sadrži 0.
- Mana ovakve evidencije je da su i operacija zauzimanja i operacija oslobađanja dugotrajne, jer prva pretražuje evidenciju, radi pronalaženja dovoljno dugačkog niza jedinica, a druga postavlja takav niz jedinica u evidenciju.
- Zato se češće evidencija slobodne fizičke radne memorije pravi u obliku liste slobodnih odsečaka fizičke radne memorije.
- Na početku rada operativnog sistema ovakva lista sadrži jedan odsečak, koji obuhvata celu fizičku slobodnu radnu memoriju.
- Ovakav odsečak se drobi u više kraćih odsečaka, kao rezultat višestrukog stvaranja i uništavanja procesa u slučajnom redosledu.
- Novonastali kraći odsečci se nalaze na mestu slika uništenih procesa, a između njih su slike postojećih procesa.
- Na početku svakog odsečka su njegova dužina i adresa narednog odsečka.
- Broj lokacija, potrebnih za smeštanje dužine dotičnog odsečka i adrese narednog odsečka, određuje najmanju dužinu odsečka i može da predstavlja jedinicu u kojoj se izražavaju dužine odsečaka (odnosno, dužine zauzimanih i oslobađanih zona fizičke radne memorije).

- Odsečci su uređeni u rastućem redosledu adresa njihovih početnih lokacija. To, prilikom oslobađanja zone fizičke radne memorije, olakšava operaciju oslobađanja:
- da pronađe dva susedna odsečka, koji mogu da se spoje u jedan, kada se između njih ubaci oslobađana zona, ili
- da pronađe odsečak, kome može da se doda (spreda ili straga) oslobađana zona ili
- da pronađe mesto u listi u koje će oslobađana zona biti uključena kao poseban odsečak.
- Listu slobodnih odsečaka pretražuje i operacija zauzimanja, radi pronalaženja dovoljno dugačkog odsečka.
- Pri tome se zauzima samo deo odsečka, koji je jednak zauzimanju zoni, dok preostali deo odsečka ostaje u listi kao novi odsečak.
- Na ovaj način se odsečci dalje usitnjavaju. To dovodi do eksterne fragmentacije.
- Ona posredno uzrokuje neupotrebljivost odsečaka, jer onemogućuje zauzimanje zone fizičke radne memorije, čija dužina je veća od dužine svakog od postojećih odsečaka, bez obzira na činjenicu da je suma dužina postojećih odsečaka veća od dužine zauzimane zone.
- Iskustvo pokazuje da se eksterna fragmentacija povećava, ako se, umesto traženja prvog dovoljno dugačkog odsečka (first fit), pokušava naći najmanji dovoljno dugačak odsečak (best fit), ili najveći dovoljno dugačak odsečak (worst fit).
- Poboljšanje ne nudi ni ideja da lista odsečaka bude ciklična i da se pretražuje ne od početka, nego od tačke u kojoj je zaustavljeno poslednje pretraživanje (next fit).
- Ideja da dužina zauzimanih zona bude uvek jednaka stepenu broja 2 (quick fit) i da postoji posebna lista odsečaka za svaku od mogućih dužina takođe ima manu, jer, pored eksterne, uvode i internu fragmentaciju, pošto se na ovaj način u proseku zauzimaju duže zone od stvarno potrebnih, čime nastaje neupotrebljiva radna memorija.
- Problem eksterne fragmentacije se može rešiti sabijanjem (compaction) slika procesa, čime se sve slike procesa pomeraju na jedan kraj fizičke radne memorije, tako da na drugom kraju bude slobodna fizička radna memorija.
- U toku sabijanja, moraju se menjati bazne adrese (segmenata) pojedinih procesa.

- Mana sabijanja je njegova dugotrajnost (sporost).
- Za potrebe virtuelnog adresnog prostora, evidencija slobodne fizičke radne memorije može da bude u obliku niza bita, u kome svaki bit odgovara slobodnoj fizičkoj stranici.
- Alternativa je da se slobodne fizičke stranice vežu u listu.
- Međutim, atraktivna je i evidencija o slobodnim odsečcima veličine jednake multiplu fizičke stranice (quick fit, buddy system), jer se u virtuelnom adresnom prostoru uvek zauzima celi broj stranica.
- Važno je uočiti da rukovanje evidencijom sistemske slobodne radne memorije zahteva sinhronizaciju procesa, u toku čije aktivnosti se istovremeno obavljaju operacije zauzimanja i oslobadanja.
- Sinhronizacija treba da obezbedi međusobnu isključivost obavljanja ovih operacija, radi očuvanja konzistentnosti pomenute evidencije.
- Za virtualnu memoriju je važno pitanje dužine stranice. Za dugačke stranice postaje izražen problem interne fragmentacije, jer sve stranice nisu uvek potpuno iskorišćene, pa se u njima javljaju neupotrebljive lokacije.
- Za kratke stranice postaje izražen problem veličine tabele stranica, jer tada virtualni adresni prostor ima više stranica, pa zato i tabela stranica ima više elemenata.
- Praksa je veličinu stranice smestila između 512 i 8192 bajta.

## **Dodela fizičkih stranica procesima**

- Za aktivnost procesa je potrebno da procesoru na raspolaganju budu kopije svih virtualnih stranica koje su neophodne za izvršavanje pojedinih mašinskih naredbi.
- Za čuvanje kopija tih stranica procesu mora biti dodeljeno dovoljno fizičkih stranica koje obrazuju minimalan skup.
- Na primer, za procesor, čije naredbe imaju najviše dva operanda, minimalni skup sadrži šest fizičkih stranica, jer se, u ekstremnom slučaju, i bajti mašinske naredbe, kao i bajti oba njena operanda, mogu nalaziti u različitim susednim fizičkim stranicama.
- Pošto se naredba može izvršiti samo kada su u fizičkoj radnoj memoriji prisutni svi bajti njenog mašinskog formata i svi bajti

njenih operanada, prethodno pomenuti ekstremni slučaj uslojava da je pridruživanje minimalnog skupa procesu preduslov bilo kakve njegove aktivnosti.

- Kada se, u toku aktivnosti procesa, desi stranični prekid, koji zahteva prebacivanje kopije nove virtuelne stranice u radnu memoriju, pre zahtevanog prebacivanja neophodno je razrešiti dilemu da li uvećati skup fizičkih stranica procesa novom fizičkom stranicom i u nju smestiti kopiju nove virtuelne stranice, ili u postojećem skupu fizičkih stranica procesa zameniti sadržaj neke od njih kopijom nove virtuelne stranice.
  - Uvećanje skupa fizičkih stranica procesa ima smisla samo ako to dovodi do smanjivanja učestanosti straničnih prekida (njihovog prosečnog broja u jedinici vremena).
  - Znači, kada je, u toku aktivnosti procesa, učestanost straničnih prekida iznad neke (iskustveno određene) gornje granice, tada ima smisla uvećanje skupa fizičkih stranica procesa, da bi se učestanost straničnih prekida svela na prihvatljiv nivo.
  - To je važno, jer obrada svakog prekida troši procesorsko vreme, pa veliki broj obrada straničnih prekida može u potpunosti da angažuje procesor i da tako vrlo uspori, ili potpuno spreči njegovu bilo kakvu korisnu aktivnost (trashing).
  - Međutim, ako je učestanost straničnih prekida ispod neke (iskustveno određene) donje granice, tada ima smisla smanjenje skupa fizičkih stranica procesa, jer i sa manjim skupom fizičkih stranica učestanost straničnih prekida ostaje u prihvatljivom rasponu.
  - U opštem slučaju učestanost straničnih prekida ne pada na nulu, ako sve kopije virtuelnih stranica procesa ne mogu stati u radnu memoriju.
  - Smanjenje skupa fizičkih stranica procesa je važno, jer se tako omogućuje neophodni rast skupova stranica drugih procesa.
  - U slučaju da je učestanost straničnih prekida između pomenute dve granice, tada nema potrebe za izmenom broja fizičkih stranica u skupu fizičkih stranica aktivnog procesa.
  - U ovom slučaju skup fizičkih stranica (odnosno, njima odgovarajući skup virtuelnih stranica) obrazuje radni skup (working set).
- Odnos učestanosti straničnih prekida i broja stranica u skupu fizičkih - stranica procesa
- Radni skup procesa nije statičan.

- U proseku, on se sporo menja u toku aktivnosti procesa (iako su, povremeno, moguće značajne kratkotrajne varijacije radnog skupa).
- Važno je uočiti da se u toku aktivnosti procesa obavezno javlja trashing, kada njegov radni skup ne može da stane u radnu memoriju.
- U ovom slučaju pomaže izbacivanje (swapping) procesa, dok se ne oslobodi dovoljan broj fizičkih stranica.
- Ovaj pristup ima smisla samo ako je radni skup procesa manji od ukupne slobodne fizičke radne memorije.
- Stepen multiprogramiranja kod virtuelne memorije zavisi od broja radnih skupova, koji se istovremeno mogu smestiti u raspoloživu fizičku radnu memoriju.
- Za uspeh koncepta virtuelne memorije važno je da se stalno prate radni skupovi istovremeno postojećih procesa i da se povremeno izbacuju procesi, čim fizička radna memorija postane pretesna za sve radne skupove (load control).
- Na ovaj način se oslobođaju fizičke stranice za preostale procese, neophodne za smeštanje njihovih radnih skupova.
- Prilikom kasnijeg ubacivanja procesa, uputno je ubacivati kopije svih virtuelnih stranica, koje obrazuju njegov radni skup.

## **Oslobađanje fizičkih stranica procesa**

- Pad učestanosti straničnih prekida ispod donje granice ukazuje na mogućnost smanjenja radnog skupa.
- U ovoj situaciji je potrebno odlučiti koju virtuelnu stranicu izbaciti iz radnog skupa, odnosno osloboditi.
- Za oslobađanje kao kandidat se nameće virtuelna stranica, koja neće biti referencirana do kraja aktivnosti procesa, ili će biti referencirana iza svih ostalih virtuelnih stranica iz radnog skupa (pod referenciranjem se podrazumeva pristup bilo kojoj lokaciji stranice, radi preuzimanja ili izmene njenog sadržaja).
- Povećanje učestanosti straničnih prekida preko gornje granice ukazuje na potrebu proširenja radnog skupa.
- U ovom slučaju, potrebno je procesu pridružiti novu fizičku stranicu, da bi se u nju smestila kopija potrebne virtuelne stranice.

- Ako nema slobodnih fizičkih stranica, tada se oslobađa, kada je to moguće, fizička stranica, koja je pridružena nekom drugom procesu.
- Time se radni skup ovog drugog procesa smanjuje. U oslobođenoj fizičkoj stranici zatečenu kopiju zamenjuje (replacement) kopija potrebne virtuelne stranice.
- Do zamene kopija virtuelnih stranica dolazi i kada se javi potreba za ubacivanjem kopije nove stranice, a učestanost straničnih prekida je između gornje i donje granice, pa se radni skup aktivnog procesa ne proširuje, nego se oslobađa jedna od fizičkih stranica iz njegovog radnog skupa.
- U oba prethodna slučaja kandidat za zamenu je fizička stranica, koja sadrži kopiju virtuelne stranice sa najstarijom referencom.
- I u slučaju pada učestanosti straničnih prekida ispod donje granice i u slučaju povećanja učestanosti straničnih prekida iznad gornje granice, kao i u slučaju kada je učestanost straničnih prekida između gornje i donje granice, javlja se potreba za oslobođanjem fizičkih stranica.
- Izbor fizičke stranice za oslobođanje se može vršiti na razne načine, po raznim algoritmima.
- Ovakvi algoritmi se nazivaju algoritmi zamene stranica (page replacement algorithms), jer se zatečeni sadržaj oslobođane fizičke stranice zamenjuje novim sadržajem.
- Oslobođanje fizičkih stranica može biti zasnovano samo na upotrebi bita referenciranja i bita izmene.
- U ovom pristupu se pronalazi fizička stranica koja nije korišćena u nekom prethodnom periodu (Not Recently Used - NRU) tako što se posmatra dvobitni broj, čiji značajniji bit odgovara bitu referenciranja, a manje značajan bit odgovara bitu izmene (MMU ih postavlja).
- Takođe se podrazumeva da se oba bita čiste i prilikom oslobođanja, odnosno prilikom zauzimanja fizičke stranice (prilikom zamene njenog sadržaja).
- Prema tome, pomenuti dvobitni broj sadrži 00, kada kopija virtuelne stranice nije korišćena nakon nekog od poslednjih vremenskih prekida (znači, niti referencirana niti izmenjena).
- Pomenuti dvobitni broj sadrži 01, kada kopija virtuelne stranice nije referencirana nakon nekog od poslednjih vremenskih prekida (ali je prethodno izmenjena).

- Ovaj broj sadrži 10, kada kopija virtuelne stranice nije izmenjena, ali je referencirana nakon poslednjeg vremenskog prekida.
- Pomenuti dvobitni broj sadrži 11, kada je kopija virtuelne stranice izmenjena i, uz to, referencirana nakon poslednjeg vremenskog prekida.
- Kandidati za zamenu su stranice, čiji dvobitni broj je najmanji.
- Prethodno opisani pristup oslobađanja fizičkih stranica je efikasan, ali nedovoljno precizno procenjuje koja fizička stranica će biti ubrzo referencirana.
- U opštem slučaju nema načina da se precizno ustanovi da li će i kada neka virtualna stranica biti referencirana.
- Ipak, zahvaljujući lokalnosti referenciranja, odnosno zapažanju da se pristupi lokacijama sa bliskim adresama dešavaju u bliskim trenucima, moguće je sa priličnom pouzdanošću zaključivati o referenciranju stranica u neposrednoj budućnosti na osnovu njihovog referenciranja u neposrednoj prošlosti.
- Prema tome, najmanju verovatnoću da bude referencirana u neposrednoj budućnosti ima stranica, čije poslednje referenciranje je najstarije, odnosno prethodi referenciranju svih ostalih stranica iz radnog skupa.
- Za pronalaženje najmanje korišćene fizičke stranice (Least Recently Used - LRU), odnosno stranice sa najstarijom referencicom, neophodno je registrovanje starosti referenciranja.
- To je moguće, ako postoji brojač, koji se automatski uvećava za jedan nakon izvršavanja svake naredbe.
- Ako se njegova zatečena vrednost automatski pridružuje virtuelnoj stranici prilikom njenog svakog referenciranja, tada je stranici sa najstarijom referencicom pridružena najmanja vrednost ovog brojača.
- Opisani pristup ima samo teoretsko značenje, jer se oslanja na hardver, koji u opštem slučaju nije raspoloživ.
- Međutim, moguće je i softverski simulirati pronalaženje najmanje korišćene fizičke stranice (Not Frequently Used - NFU/aging).
- Ovakva simulacija se zasniva na korišćenju bita referenciranja svake virtualne stranice i na uvođenju polja starosti referenci u elemente tabele stranica.
- Ovo polje sadrži n bita, po jedan bit za svaku od vrednosti bita referenciranja u poslednjih n trenutaka.

- Polje starosti referenci se periodično ažurira, tako što se iz njega periodično izbacuje najstariji bit referenciranja.
- To je zadatak obradivača vremenskog prekida koji:
  - pomera u desno za jedan bit polje starosti referenci svake od virtuelnih stranica iz radnog skupa aktivnog procesa
  - dodaje s leva, u upražnjenu bitnu poziciju ovog polja, zatečeni sadržaj bita referenciranja dotične virtuelne stranice
  - zatim očisti bit referenciranja.
- Na ovaj način polje starosti referenci sadrži najmanju vrednost za virtuelnu stranicu, koja je najranije referencirana u prethodnih n trenutaka.
- Polje starosti referenci se može iskoristiti i za određivanje radnog skupa.
- Kriterijum može biti postavljenost nekog od k najznačajnijih bita iz polja starosti referenci.
- Po tom kriterijumu radnom skupu pripadaju sve virtuelne stranice, koje imaju bar jedan bit postavljen u najznačajnijih k bita svog polja starosti referenci.
- Prethodno opisani pristupi oslobađanja fizičkih stranica (NRU, LRU, NFU) obezbeđuju smanjenje učestanosti straničnih prekida nakon povećanja broja fizičkih stranica procesa.
- Znači, obavljanje liste istih zahteva za pristupe virtuelnim stranicama dovodi do pojave manje straničnih prekida nakon povećanja broja fizičkih stranica procesa, nego pre toga.
- Ovo je važno istaći, jer svi pristupi oslobađanja fizičkih stranica ne dovode obavezno do smanjenja učestanosti straničnih prekida nakon povećanja broja fizičkih stranica procesa.
- To je karakteristično, na primer, za pristup, kod koga se oslobađa fizička stranica sa najstarijim sadržajem (First In First Out - FIFO), bez obzira da li je ona skoro referencirana.
- Ovaj pristup ima tendenciju da ne oslobađa fizičke stranice, čiji sadržaj je svežiji, čak i ako one neće biti uskoro referencirane.
- Međutim, manu poslednje pomenutog pristupa se otklanja, ako se on modifikuje, tako da se za oslobađanje prvo traži fizička stranica sa najstarijim nereferenciranim sadržajem, a ako su sadržaji svih fizičkih stranica referencirani, tek tada se oslobađa fizička stranica sa najstarijim sadržajem (second chance i clock pristupi).

- U sva tri poslednje pomenuta pristupa (FIFO, second chance i clock) fizičke stranice procesa se uvezuju u listu.
- Pri tome položaj u listi ukazuje na starost sadržaja fizičke stranice.
- U clock algoritmu zamene ovakva lista je kružna, a poseban pokazivač, kao kazaljka na satu, pokazuje na stranicu sa najstarijim sadržajem.
- Postoji i varijanta clock pristupa (wsclock) u kome se uz svaku fizičku stranicu iz liste čuva i podatak o trenutku poslednjeg referenciranja.
- Za sve fizičke stranice, za koje ovaj trenutak ispada iz unapred određenog vremenskog intervala (gledajući u prošlost), se smatra da ne spadaju u radni skup, pa su one kandidati za zamenu.
- Pristup NFU/aging i pristup wsclock imaju najveću praktičnu važnost.
- Virtuelna memorija pokazuje najbolje rezultate, kada uvek ima slobodnih fizičkih stranica.
- To se može postići, ako se uvede poseban stranični sistemski proces, koji se periodično aktivira, da bi oslobođio izvestan broj fizičkih stranica.
- On, pri tome, odabira fizičke stranice za oslobođanje po nekom od prethodno opisanih algoritama zamene.
- Zadatak straničnog sistemskog procesa je i da u masovnu memoriju prebacuje izmenjene kopije virtuelnih stranica i da tako čuva ažurnost masovne memorije.
- Opisani pristupi oslobođanja fizičkih stranica pokazuju da se u praksi ne prati učestanost straničnih prekida.
- Umesto toga, broj fizičkih stranica procesa varira između minimalnog skupa i iskustveno određenog maksimalnog skupa.
- Pri tome stranični prekidi izazivaju povećanje broja fizičkih stranica procesa do maksimalne veličine, a stranični sistemski proces se brine o njegovom smanjivanju.

## **Implementacija upravljanja virtuelnom memorijom**

- Sloj za rukovanje virtuelnom memorijom podržava operacije zaузimanja i oslobođanja, a oslanja se na stranični sistemski proces i obradivače vremenskog i straničnog prekida.

- Obradivač straničnog prekida se aktivira, kada je referencirana virtuelna stranica, čija kopija nije prisutna u fizičkoj radnoj memoriji, odnosno, u nekoj od njenih fizičkih stranica.
- Ako je, greškom, referencirana virtuelna stranica, koja uopšte ne postoji u slici procesa, obradivač straničnog prekida završava aktivnost prekinutog procesa (uz odgovarajuću poruku).
- Inače, ovaj obradivač odabira slobodnu fizičku stranicu i prema njoj usmerava prenos kopije potrebne virtuelne stranice sa masovne memorije.
- Kada se taj prenos završi, obradivač straničnog prekida ažurira polja odgovarajućeg elementa tabele stranica i omogućuje nastavak aktivnosti prekinutog procesa.
- Kada nema slobodne fizičke stranice, ovaj obradivač oslobađa neku od fizičkih stranica.
- Ako ona sadrži izmenjenu kopiju virtuelne stranice, on pokreće prenos ove kopije u masovnu memoriju, radi ažuriranja odgovarajuće virtuelne stranice.
- Po završetku ovoga prenosa, obradivač straničnog prekida usmerava ka oslobođenoj fizičkoj stranici prenos kopije potrebne virtuelne stranice.
- Nakon završetka ovog prenosa i ažuriranja polja odgovarajućih elemenata tabele stranica, on omogućava nastavak aktivnosti prekinutog procesa.
- Za uspešno obavljanje posla, obradivaču straničnog prekida je potrebna evidencija o položaju virtuelnih stranica u masovnoj memoriji.
- On takođe, koristi i evidenciju slobodnih fizičkih stranica.
- Nju koriste i operacija zauzimanja i operacija oslobađanja, pa rukovanje ovom evidencijom mora obezbediti sinhronizaciju procesa, i to onemogućenjem prekida.
- Operacija zauzimanja omogućuje zauzimanje bar minimalnog skupa, radi stvaranja procesa, a operacija oslobađanja oslobađa sve fizičke stranice iz radnog skupa unišavanog procesa.

### **Osnova sloja za rukovanje virtuelnom memorijom**

- Sloj za rukovanje virtuelnom memorijom se oslanja na operacije sloja za rukovanje kontrolerima, da bi obezbedio prenos kopija virtuelnih blokova na relaciju masovna i radna memorija i da bi

smestio adrese svojih obradivača prekida u odgovarajuće elemente tabele prekida.

## Pitanja

### Pitanja

- Kakav može biti logički adresni prostor?
- Šta karakteriše kontinualni logički adresni prostor?
- Šta karakteriše segmentirani logički adresni prostor?
- Šta karakteriše stranični logički adresni prostor?
- Šta karakteriše stranično segmentirani logički adresni prostor?
- Šta karakteriše translacione podatke?
- Šta karakteriše translaciju logičkih adresa kontinualnog logičkog adresnog prostora u fizičke?
- Koji logički adresni prostor se koristi kada veličina fizičke radne memorije prevazilazi potrebe procesa?
- Šta karakteriše segmentaciju?
- Šta sadrže elementi tabele stranica?
- Šta karakteriše virtuelni adresni prostor?
- Po kom principu se prebacuju kopije virtuelnih stranica?
- Šta karakteriše straničnu segmentaciju?
- Kako se deli fizička radna memorija?
- Kako se deli virtuelni adresni prostor?
- U kom obliku može biti evidencija slobodne fizičke memorije?
- Kod kog adresnog prostora se javlja eksterna fragmentacija?
- Kako se nazivaju skupovi fizičkih stranica, koji se dodeljuju procesima?
- Kada treba proširiti skup fizičkih stranica procesa?
- Kada treba smanjiti skup fizičkih stranica procesa?
- Kada ne treba menjati veličinu skupa fizičkih stranica procesa?
- Koji pristupi oslobođanja fizičkih stranica obezbeđuju smanjenje učestanosti straničnih prekida nakon povećanja broja fizičkih stranica procesa?

- Koji pristupi oslobađanja fizičkih stranica koriste bit referenciranja?
- 

### **Koji pristupi oslobađanja fizičkih stranica koriste bit izmene?**

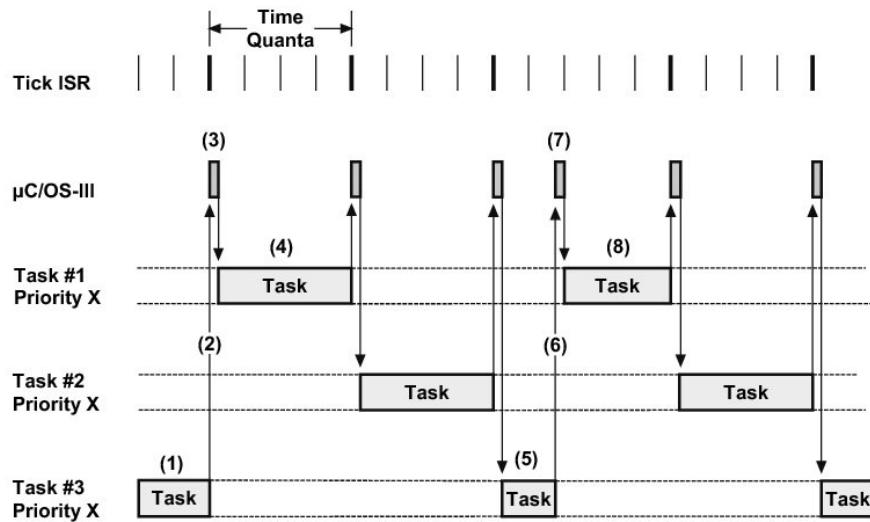
title: Procesor author: Veljko Petrović date: 2023-05 —

## **Sloj za rukovanje procesorom**

### **Raspoređivanje**

- Osnovni zadatak rukovanja procesorom je preključivanje procesora sa aktivnog procesa na neki od spremnih procesa.
- O izboru spremnog procesa, na koga se preključuje procesor, brine raspoređivanje (scheduling).
- Ovaj izbor zavisi od cilja raspoređivanja.
- Tipični ciljevi raspoređivanja su, na primer:
  - poboljšanje iskorišćenja procesorskog vremena,
  - ravnomerna raspodela procesorskog vremena
  - što kraći odziv na korisničku akciju ili neki drugi oblik postizanja potrebnog kvaliteta usluge (Quality of Service, QoS), kao što je rezervisanje procesorskog vremena radi obezbeđenja kvalitetne reprodukcije zvuka ili videa kod multimedijalnih aplikacija.
- Ovakvi ciljevi nisu saglasni, pa se ne mogu istovremeno ostvariti.
- Za neinteraktivno korišćenje računara cilj raspoređivanja je poboljšanje iskorišćenja procesorskog vremena.
- Ovakav cilj se ostvaruje minimiziranjem preključivanja na neophodan broj (samo nakon pozivanja blokirajućih sistemskih operacija ili nakon kraja aktivnosti procesa).
- Za interaktivno korišćenje računara (u višekorisničkom režimu rada) ciljevi raspoređivanja su ravnomerna raspodela procesorskog vremena između istovremeno postojećih procesa, odnosno, između njihovih vlasnika (korisnika, koji istovremeno koriste računar) i što kraći odziv na korisničku akciju.

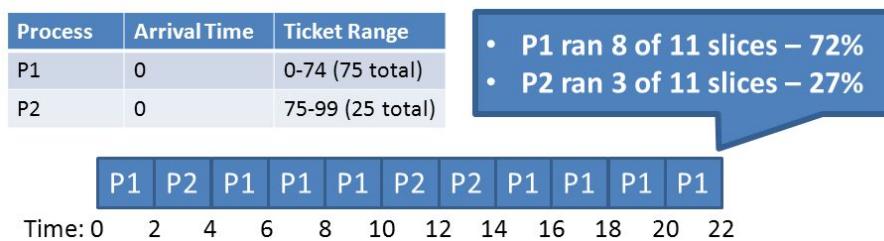
- Ovakvi ciljevi se ostvaruju kružnim raspoređivanjem (round robin scheduling), koje svakom od istovremeno postojećih procesa dodeljuje isti vremenski interval, nazvan kvantum.
- Po isticanju kvantuma, aktivni proces prepušta procesor spremnom procesu, koji najduže čeka na svoj kvantum.
- Neophodan preduslov za primenu kružnog raspoređivanja je da se preključivanje vezuje za trenutak u kome se završava tekući kvantum.
- Zato je neophodno da se preključivanje poziva neposredno nakon obrade prekida sata (pre nastavka prekinutog procesa).
- Kružno raspoređivanje se uspešno primenjuje i u situaciji kada hitnost svih procesa nije ista, pa se, zbog toga, procesima dodeljuju razni prioriteti.
- Pri tome se podrazumeva da kružno raspoređivanje važi u okviru grupe procesa sa istim prioritetom.
- Procesor se preključuje na procese sa nižim prioritetom samo kada se završi (zaustavi) aktivnost i poslednjeg od procesa sa višim prioritetom.
- Procesor se preključuje na proces sa višim prioritetom odmah po pojavi ovakvog procesa (preemptive scheduling), odnosno odmah po omogućavanju nastavka aktivnosti prioritetnijeg procesa.



- Izvor: <https://doc.micrium.com/display/osiidoc/Round-Robin+Scheduling>
- Dinamička izmena prioriteta procesa doprinosi ravnomernosti

raspodele procesorskog vremena između procesa, ako se uspostavi obrnuta proporcionalnost između prioriteta procesa i obima u kome je on iskoristio poslednji kvantum.

- Pri tome se periodično proverava iskoristiće poslednjeg kvantuma svakog od procesa i, u skladu s tim, procesima se dodeljuju novi prioriteti.
- Takođe, dinamička izmena prioriteta procesa doprinosi ravnomernosti raspodele procesorskog vremena između korisnika, ako se uspostavi obrnuta proporcionalnost između prioriteta procesa, koji pripadaju nekom korisniku, i ukupnog u dela u procesorskom vremenu tog korisnika u toku njegove interakcije sa računaram.
- Znači, što je ukupan ideo korisnika više ispod željenog proseka, to prioritet njegovih procesa više raste.
- Ravnomerna raspodela procesorskog vremena se može postići i bez izmena prioriteta, ako se uvede lutrijsko raspoređivanje (lottery scheduling).
- Ono se zasniva na dodeli procesima lutrijskih lozova.
- Nakon svakog kvantuma na slučajan način se izvlači broj loza, a procesor se preključuje na proces koji poseduje izvučeni loz.
- Tako, ako ukupno ima m lozova, proces, koji poseduje n od m lozova ( $n < m$ ), u proseku koristi  $n/m$  kvantuma procesorskog vremena.



- Izvor: Christo Wilson Lecture 6: Process Scheduling
- Za multimedijalne aplikacije, koje zahtevaju visoku propusnost podataka i njihovu isporuku sa pravilnim periodom, cilj raspoređivanja je garantovanje procesima potrebnog broja kvantuma u pravilnim vremenskim razmacima.
- Ostvarenje raznih ciljeva raspoređivanja se može zasnovati na istim mehanizmima raspoređivanja.
- U tom slučaju razni načini primene tih mehanizama ili razne

politike raspoređivanja dovode do ostvarenja raznih ciljeva raspoređivanja.

- Razdvajanje mehanizama raspoređivanja od politike raspoređivanja je važno zbog fleksibilnosti.
- Mehanizmi (mogućnosti) raspoređivanja omogućuju uticanje na dužinu kvantuma i na nivo prioriteta, a politika raspoređivanja (iskorišćenje neke od mogućnosti) određuje dužinu kvantuma i nivo prioriteta.
- Uticanje na dužinu kvantuma je važno, jer od dužine kvantuma zavisi iskorišćenje procesora, ali i odziv računara, odnosno brzina kojom on reaguje na korisničku akciju sa terminala.
- Pri tome, skraćenje (do određene granice) kvantuma doprinosi poboljšanju odziva, ali i smanjenju iskorišćenja procesora, jer povećava broj preključivanja koja troše procesorsko vreme.
- Sviše kratak kvantum počinje da ugrožava i odziv, kada se previelik procenat procesorskog vremena počne da troši na preključivanje.
- Sa stanovišta iskorišćenja procesora prihvatljiva su samo neophodna preključivanja (kada nije moguć nastavak aktivnosti procesa), odnosno, značajno smanjivanje učestanosti preključivanja.
- S tom idejom na umu moguće je iskoristiti dinamičku izmenu prioriteta procesa za:
  - održavanje dobrog odziva za procese, koji su u interakciji sa korisnicima
  - održavanje dobrog iskorišćenja procesora za pozadinske (background) procese, koji nisu u (čestoj) interakciji sa korisnicima.
  - Pri tome se interaktivnim procesima dodeljuje najviši prioritet i najkraći kvantum.
  - Pozadinskim procesima, koji su vrlo dugo aktivni bez ikakve interakcije sa korisnikom, se dodeljuje najniži prioritet i najduži kvantum.
  - Procesu automatski opada prioritet i produžava se kvantum što je on duže aktivan i ima manju interakciju sa korisnikom.
  - Povećanje interakcije sa korisnikom dovodi do porasta prioriteta procesa i smanjenja njegovog kvantuma.
  - Dinamička izmena prioriteta se obavlja periodično i nalazi se u nadležnosti politike raspoređivanja, koja je zadužena i za

vezivanje odgovarajućih dužina kvantuma za odgovarajuće prioritete.

- Za operacije sloja za rukovanje procesorom je zajedničko da se obavljaju pod onemogućenim prekidima, što je prihvatljivo, jer je reč o kratkotrajnim operacijama.
- To je naročito značajno za operacije koje rukuju deskriptorima procesa, jer jedino onemogućenje prekida osigurava ispravnost rukovanja listama u koje se uključuju i iz kojih se isključuju deskriptori procesa u toku ovih operacija (odnosno, osigurava konzistentnost ovih listi).
- Pod onemogućenim prekidima se obavljaju i operacija preključivanja (sa operacijom raspoređivanja), sistemska operacija za izmenu prioriteta procesa, kao i sistemske operacije za sinhronizaciju procesa.
- Operacija raspoređivanja obuhvata bar dve radnje.
- Jedna ubacuje proces među spremne procese, tako što njegov deskriptor uvezuje na kraj liste deskriptora spremnih procesa, koja odgovara prioritetu dotičnog procesa.
- U ovom slučaju se podrazumeva da za svaki prioritet postoji posebna lista deskriptora spremnih procesa, na koju se primenjuje kružno raspoređivanje.
- Druga od ove dve radnje izvezuje iz liste deskriptora spremnih procesa deskriptor najprioritetnijeg spremnog procesa.

## Pitanja

### Pitanja

- Šta karakteriše tipične ciljeve raspoređivanja?
- Šta je cilj raspoređivanja za neinteraktivno korišćenje računara?
- Šta je cilj raspoređivanja za interaktivno korišćenje računara?
- Zašto je uvedeno kružno raspoređivanje?
- Šta doprinosi ravnomernoj raspodeli procesorskog vremena?
- Šta je cilj raspoređivanja za multimedijalne aplikacije?
- Do čega dovodi skraćenje kvantuma?

- Šta se postiže uticanjem na nivo prioriteta i na dužinu kvantuma?— title: Klasifikacija Operativnih Sistema author: Veljko Petrović date: 2023-05 —

## **Klasifikacija Operativnih Sistema**

### **Kriterijum klasifikacije OS**

- Jedan od mogućih kriterijuma za klasifikaciju operativnih sistema je vrsta računara kojim operativni sistem upravlja. Po tom kriteriju mogu se izdvojiti:
  - operativni sistemi realnog vremena (RTOS)
  - distribuirani operativni sistemi

### **Operativni sistemi realnog vremena**

- Operativni sistemi realnog vremena (real time operating system) su namenjeni za primene računara u kojima je neophodno obezbediti reakciju na spoljni događaj u unapred zadanom vremenu.
- Ovakvi operativni sistemi su, zbog toga, podređeni ostvarenju što veće brzine izvršavanja korisničkih programa.
- Za operativne sisteme realnog vremena je tipično da su, zajedno sa računarom, ugrađeni (embedded) u sistem, čije ponašanje se ili samo prati, ili čijim ponašanjem se upravlja.
- Zadatak operativnih sistema realnog vremena je da samo stvore okruženje za korisničke programe, jer komunikaciju sa krajnjim korisnikom obavljaju korisnički programi.
- Zato se operativni sistemi realnog vremena obično koriste samo na programskom nivou.
- Modul za rukovanje procesima je podređen potrebi brzog stvaranja i uništenja procesa, njihove brze i lake saradnje, kao i brzog preključivanja procesora sa procesa na proces.
- Zato obično svi procesi dele isti fizički adresni prostor.
- To je moguće, jer ne postoji potreba za međusobnom zaštitom procesa, pošto oni imaju istog autora, ili njihovi autori pripadaju istom timu.
- Modul za rukovanje datotekama nije obavezni deo operativnog sistema realnog vremena, jer sve primene realnog vremena ne zahtevaju masovnu memoriju.

- Kada rukovanje datotekama postoji, ono obično podržava kontinuirane datoteke, zbog brzine pristupa podacima.
- Mane kontinuiranih datoteka se ovde ne ispoljavaju, jer su unapred poznati svi zahtevi primene.
- Modul za rukovanje radnom memorijom obično podržava efikasno zauzimanje memorijskih zona sa unapred određenom veličinom, da bi se izbegla ili umanjila eksterna fragmentacija.
- Modul za rukovanje kontrolerima podržava tipične ulazne i izlazne uređaje i, uz to, omogućava jednostavno uključivanje novih drajvera za specifične uređaje.
- Pri tome se nude blokirajuće i neblokirajuće sistemske operacije, ali i vremenski ograničene blokirajuće sistemske operacije.
- Zahvaljujući neblokirajućim sistemskim operacijama, moguće je vremenski preklopiti aktivnosti procesora i kontrolera.
- Vremenski ograničene blokirajuće sistemske operacije omogućuju reakciju u zadanom vremenskom intervalu na izostanak željenog događaja, odnosno, na izostanak obavljanja pozvane sistemske operacije.
- Modul za rukovanje procesorom mora da obezbedi efikasno rukovanje vremenom.
- Za to se često koriste posebni satovi - tajmeri.
- U sklopu toga, mora se obezbediti da aktivnost procesa bude završena do graničnog trenutka (deadline scheduling).
- To se postiže sortiranjem deskriptora spremnih procesa po dužini preostalog vremena do graničnog trenutka (earliest deadline first - EDF).
- Podrazumeva se da ovo sortiranje dovodi na prvo mesto deskriptor procesa sa najkraćim preostalim vremenom do graničnog trenutka, tako da je njegov proces prvi na redu za aktiviranje.
- Ako je aktivnost procesa periodična sa unapred određenim i nepromenljivim trajanjem aktivnosti u svakom periodu, tada se procesima mogu dodeliti prioriteti jednakih broju njihovih perioda u jedinici vremena (rate monotonic scheduling).
- Tako se može obezbediti da aktivnost procesa bude obavljena pre kraja svakog od njegovih perioda.
- U svakom slučaju, postavljeni cilj raspoređivanja može da se ostvari samo ako ima dovoljno procesorskog vremena za sve procese.

- Modul za rukovanje procesorom operativnog sistema realnog vremena obično podržava mehanizam semafora, jer je on jednostavan za korišćenje, brz i jer ne zahteva izmene kompjlera.

## **Distribuirani OS**

### **Distribuirani OS**

- Distribuirani operativni sistemi upravljaju međusobno povezanim računarima, koji su prostorno udaljeni.
- Potrebu za povezivanjem prostorno udaljenih (distribuiranih) računara nameće praksa.
- S jedne strane, prirodno je da računari budu na mestima svojih primena, na primer, uz korisnike ili uz delove industrijskih postrojenja, koje opslužuju.
- Na taj način računari mogu biti potpuno posvećeni lokalnim poslovima, koji su vezani za mesta njihove primene, pa mogu efikasno obavljati ovakve poslove.
- S druge strane, neophodno je omogućiti saradnju između prostorno udaljenih korisnika, odnosno obezbediti usaglašeni rad prostorno udaljenih delova istog industrijskog postrojenja.
- Za to je potrebno obezbediti razmenu podataka između računara, posvećenih pomenutim korisnicima, odnosno posvećenih pomenutim delovima industrijskog postrojenja.
- Za to je potrebno obezbediti razmenu podataka između računara, posvećenih pomenutim korisnicima, odnosno posvećenih pomenutim delovima industrijskog postrojenja.
- Radi toga, ovakvi, prostorno udaljeni računari se povezuju komunikacionim linijama, koje omogućuju prenos (razmenu) podataka, organizovanih u poruke.
- Na ovaj način nastaje distribuirani računarski sistem (distributed computer system).
- Za svaki od računara, povezanih u distribuirani računarski sistem, je neophodno da sadrže procesor, radnu memoriju i mrežni kontroler.
- Prisustvo masovne memorije i raznih ulaznih i izlaznih uređaja u sastavu ovakvih računara zavisi od mesta njihove primene i, u opštem slučaju, nije obavezno.

- Zato nema ni potrebe da ih podržava operativni sistem, prisutan na računarima iz distribuiranog računarskog sistema.
- Ovakav operativni sistem ima smanjenu funkcionalnost u odnosu na "običan" operativni sistem, pa se naziva mikrokernel (microkernel).
- Hijerarhijska struktura mikrokernela je prikazana na slici:

modul za rukovanje procesima
modul za razmenu poruka
modul za rukovanje radnom memorijom
modul za rukovanje kontrolerima
modul za rukovanje procesorom

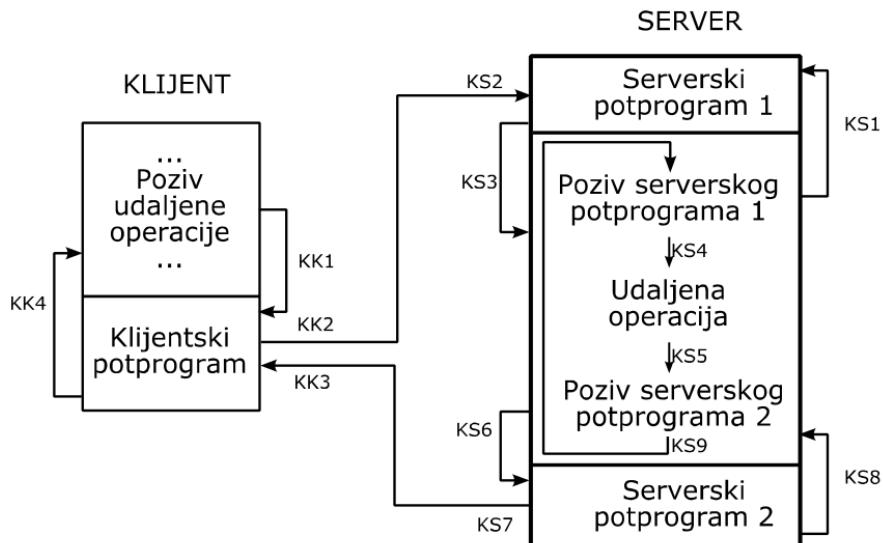
- Mikrokernel ne sadrži modul za rukovanje datotekama, jer on nije potreban za svaki od računara iz distribuiranog računarskog sistema.
- Zato se ovaj modul prebacuje u korisnički sloj (iznad mikrokernela), koji je predviđen za korisničke procese.
- Modul za rukovanje procesima se oslanja na modul za razmenu poruka, da bi pristupio izvršnoj datoteci koja je locirana na nekom drugom računaru.
- Modul za razmenu poruka se oslanja na modul za rukovanje radnom memorijom, radi dinamičkog zauzimanja i oslobođanja bafera, namenjenih za privremeno smeštanje poruka.
- Modul za razmenu poruka se oslanja i na modul za rukovanje kontrolerima, u kome se nalazi drajver mrežnog kontrolera, posredstvom koga se fizički razmenjuju poruke.
- Na kraju, modul za razmenu poruka se oslanja i na modul za rukovanje procesorom.
- Ovo je potrebno, da bi se, na primer, privremeno zaustavila aktivnost procesa do prijema poruke, bez koje nastavak aktivnosti nije moguć, ali i da bi se moglo reagovati na dugotrajni izostanak očekivanog prijema poruke.
- Modul za razmenu poruka nije samo na raspolaganju modulu za rukovanje procesima.
- On sadrži sistemske operacije, koje omogućuju razmenu poruka, odnosno saradnju između procesa, aktivnih na raznim računarima, kao i saradnju između procesa, aktivnih na istom računaru.

- Tipičan oblik saradnje procesa je da jedan proces traži uslugu od drugog procesa.
- To je potrebno, na primer, kada jedan proces želi da na svom računaru, koji je bez masovne memorije, stvori novi proces.
- On se, tada, posredstvom modula za rukovanje procesima, obraća drugom procesu, aktivnom na računaru sa masovnom memorijom, zahtevajući od njega, kao uslugu, da mu pošalje sadržaj odgovarajuće izvršne datoteke.
- Uobičajeni način traženja i dobijanja usluge se sastoji od pozivanja operacije, čije obavljanje dovodi do pružanja tražene usluge.
- Ako pozivana operacija ne odgovara potprogramu koji se lokalno izvršava u okviru aktivnosti procesa pozivaoca, nego odgovara potprogramu koji se izvršava u okviru aktivnosti drugog, udaljenog procesa, aktivnog na udaljenom računaru, reč je o pozivu udaljene operacije (Remote Procedure Call - RPC).
- Proces, koji poziva udaljenu operaciju, se nalazi u ulozi klijenta (primaoca usluge), a proces, koji obavlja udaljenu operaciju, se nalazi u ulozi servera (davaoca usluge).

### **Poziv udaljene operacije (RPC)**

- Poziv udaljene operacije liči na poziv (lokalne) operacije.
- Znači, on ima oblik poziva potprograma, u kome se navode oznaka (ime) operacije i njeni argumenti.
- Ovakav potprogram se naziva klijentski potprogram (client stub), jer je klijent njegov jedini pozivalac.
- U klijentskom potprogramu je sakriven niz koraka, koji se obavljaju, radi dobijanja zahtevane usluge.
- U ove korake spadaju:
  - pronalaženje procesa servera, koji pruža zahtevanu uslugu
  - pakovanje (marshalling) argumenata (navedenih u pozivu klijentskog potprograma) u poruku zahteva
  - slanje serveru ove poruke zahteva
  - prijem od servera poruke odgovora sa rezultatom pružanja zahtevane usluge
  - raspakivanje prispele poruke odgovora

- isporuka rezultata pružanja zahtevane usluge pozivaocu klijentskog potprograma
- Simetrično klijentskom potprogramu postoje dva serverska potprograma (server stub).
- Njih poziva jedino server, a oni kriju više koraka, koji se obavljaju, radi pružanja zahtevane usluge.
- Prvi od serverskih potprograma obuhvata:
  - prijem poruke zahteva
  - raspakivanje argumenata iz ove poruke
  - Drugi od serverskih potprograma obuhvata:
    - pakovanje rezultata usluge (koju je pružio server) u poruku odgovora
    - slanje klijentu ove poruke odgovora.
  - Između poziva ova dva serverska potprograma se nalazi lokalni poziv operacije, koja odgovara zahtevanoj usluzi, odnosno, programski tekst, koji opisuje aktivnost servera na pružanju zahtevane usluge.
  - Slika prikazuje redosled koraka klijenta i servera tokom poziva udaljene operacije (kk1 do kk4 označavaju sekvencu koraka klijenta, a ks1 do ks9 označavaju sekvencu koraka servera).



- Oslanjanje na poziv udaljene operacije olakšava posao programeru, jer od njega krije, na prethodno opisani način, detalje

saradnje klijenta i servera.

- Pri tome, klijentski potprogram pripada biblioteci udaljenih operacija.
- Ova biblioteka sadrži po jedan klijentski potprogram za svaku od postojećih udaljenih operacija.
- Klijentski potprogram se generiše, zajedno sa serverskim potprogramima, prilikom prevodenja programa, koji odgovara serveru.

## **Problemi poziva udaljene operacije (RPC)**

- Uprkos nastojanju da što više liči na poziv lokalne operacije, poziv udaljene operacije se značajno razlikuje od svog uzora.
- Te razlike su posledica koraka, sakrivenih u pozivu udaljene operacije, koji uzrokuju da se u toku poziva udaljene operacije mogu da pojave problemi, čija pojava nije moguća kod poziva lokalne operacije.
  - Tako je moguće:
    - da se ne pronađe server, koji pruža zahtevanu uslugu
    - da se, u toku prenosa, izgube ili poruka zahteva ili poruka odgovora
    - da dođe do otkaza ili servera, ili klijenta u toku njihovog rada
    - Ako nema servera, tada nije moguće pružanje tražene usluge.
    - To je nemoguća situacija kod poziva lokalne operacije.
  - Do istog rezultata dovode smetnje na komunikacionim linijama, koje onemogućuju prenos bilo poruke zahteva, bilo poruke odgovora.
  - Kada u očekivanom vremenu izostane prijem poruke odgovora, bilo zbog gubljenja poruke zahteva, bilo zbog gubljenja poruke odgovora, jedino što se na strani klijenta može uraditi je da se ponovo pošalje (retransmituje) poruka zahteva.
  - Pri tome je broj retransmisija ograničen.
  - Ako je izgubljena poruka zahteva, njenom retransmisijom se stvara mogućnost da ona stigne do servera i da on pruži traženu uslugu.
  - Međutim, ako je izgubljena poruka odgovora, tada treba sprečiti da, po prijemu retransmitovane poruke zahteva, server ponovi pružanje već pružene usluge.

- Da bi server razlikovao retransmitovanu poruku od originalne, dovoljno je da svaka originalna poruka ima jedinstven redni broj i da server za svakog klijenta pamti redni broj poslednje primljene poruke zahteva od tog klijenta.
- Prijem poruke sa zapamćenim rednim brojem ukazuje na retransmitovanu poruku, koja je već primljena.
- Otkaz servera, izazvan kvarom računara, je neprijatan zbog teškoća da se ustanovi da li je do otkaza došlo pre, u toku, ili posle pružanja usluge.
- Zato je problematično da poziv udaljene operacije garantuje da će zahtevana usluga biti pružena samo jednom, kao kod poziva lokalne operacije.
- Na primer, na strani klijenta otkaz servera se ispoljava kao izostajanje poruke odgovora.
- Tada retransmisija poruke zahteva može navesti ponovo pokrenutog servera da još jednom pruži već pruženu uslugu, jer je, u ovom slučaju, server izgubio, zbog otkaza, evidenciju o rednim brojevima poslednje primljenih poruka zahteva od klijenata.
- Kod poziva lokalne operacije ovo se ne može desiti, jer otkaz računara znači i kraj izvršavanja celog programa, bez pokušaja njegovog automatskog opravka.
- Garantovanje da će server pružiti zahtevanu uslugu samo jednom podrazumeva upotrebu stabilne memorije (stable storage) koja je ima visoku pouzdanost, pa može ispravno funkcionisati i u okolnostima otkaza servera.
- Otkaz klijenta znači da server uzaludno pruža zahtevanu uslugu.
- Ovo se izbegava tako što server obustavlja pružanje usluga klijentima, za koje ustanovi da su doživeli otkaz.
- To klijenti sami mogu da jave serveru, nakon svog ponovnog pokretanja, ili to server može sam da otkrije, periodičnom proverom stanja klijenata, koje opslužuje.
- Poziv udaljene operacije praktično dozvoljava da argumenti budu samo vrednosti, a ne i adrese, odnosno pokazivači, zbog problema kopiranja pokazanih vrednosti sa klijentovog računara na računar servera i u obrnutom smeru.
- Pored toga, ako su ovi računari različiti, javlja se i problem konverzije vrednosti, jer se, na primer, predstava realnih brojeva razlikuje od računara do računara.

- Činjenica da se u okviru klijentskog potprograma javlja potreba za pronalaženjem servera, ukazuje da u vreme pravljenja izvršnog oblika klijentskog programa nije poznato koji server će usluživati klijenta.
- U opštem slučaju, može biti više servera iste vrste i svaki od njih može istom klijentu da pruži zatraženu uslugu.
- Radi toga se uvodi poseban server imena (name server, binder).
- Njemu se, na početku svoje aktivnosti, obraćaju svi serveri i ostavljaju podatke o sebi, kao što je, na primer, podatak o vrsti usluge koje pružaju.
- Serveru imena se obraćaju i klijenti, radi pronalaženja servera, koji pruža zahtevanu uslugu.
- Na ovaj način se ostvaruje dinamičko linkovanje (dynamic binding) klijenta, koji zahteva uslugu, i servera, koji pruža zahtevanu uslugu.

## Razmena poruka

- Klijentski i serverski potprogrami, koji omogućuju poziv udaljene operacije, se oslanjaju na sistemske operacije modula za razmenu poruka.
- Prva od ovih operacija je sistemska operacija zahtevanja usluge, a druge dve su sistemske operacije prijema zahteva i slanja odgovora.
- Sistemska operacija zahtevanja usluge je namenjena klijentu i poziva se iz njegovog potprograma.
- Ona omogućuje slanje poruke zahteva i prijem poruke odgovora.
- Sistemske operacije prijema zahteva i slanja odgovora su namenjene serveru i omogućuju prijem poruke zahteva i slanje poruke odgovora.
- Sistemska operacija prijema zahteva se poziva iz prvog serverskog potprograma, a sistemska operacija slanja odgovora se poziva iz drugog serverskog potprograma.
- Ove tri sistemske operacije ostvaruju poseban protokol razmene poruka (request reply protocol), koji je prilagođen potrebama poziva udaljene operacije.
- Sistemske operacije zahtevanja usluge, prijema zahteva i slanja odgovora su blokirajuće.

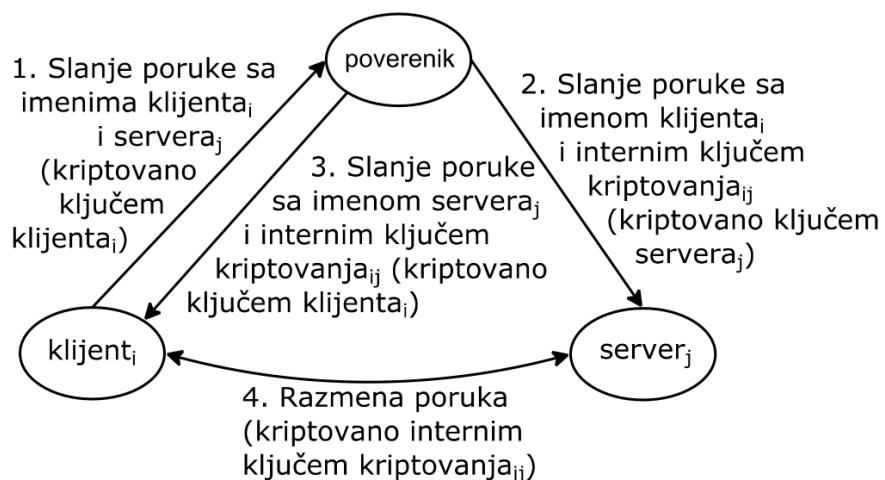
- Prva zaustavlja aktivnost klijenta do stizanja odgovora, ili do isticanja zadano vremenskog perioda.
- Druga zaustavlja aktivnost servera do stizanja zahteva, a treća zaustavlja aktivnost servera do isporuke odgovora ili do isticanja zadano vremenskog intervala.
- Ove tri sistemske operacije su zadužene za prenos poruka.
- Pored slanja i prijema poruka, one:
  - potvrđuju prijem poruka
  - retransmituju poruke, čiji prijem nije potvrđen
  - šalju upravljačke poruke, kojima se proverava i potvrđuje aktivnost servera (čime se omogućuje otkrivanje njegovog otkaza) i slično.
- U nadležnosti ovih operacija je i rastavljanje poruka u pakete, koji se prenose preko komunikacionih linija, sastavljanje poruka od paketa, pristiglih preko komunikacionih linija, potvrda prijema paketa i retransmisija paketa čiji prijem nije potvrđen, kao i prilagođavanje brzine slanja paketa brzini kojom oni mogu biti primani (flow control).
- Pomenute tri sistemske operacije koriste usluge drajvera sata, radi reagovanja na isticanje zadanih vremenskih intervala, nakon kojih je, na primer, potrebno ili retransmitovati poruku, ili poslati poruku potvrde.
- One pozivaju i (neblokirajuće) operacije gornjeg dela drajvera mrežnog kontrolera, radi fizičkog prenosa i prijema paketa.
- U donjem delu ovog drajvera se nalaze obradivači prekida, zaduženi za registrovanje uspešnog slanja i uspešnog prijema paketa.
- Sistemske operacije modula za razmenu poruka se brinu o baferima, namenjenim za (privremeno) smeštanje poruka.
- Na primer, ako server nije pozvao sistemsku operaciju prijema zahteva, jer je aktivan na usluživanju prethodno primljenog zahteva od jednog klijenta, a pristigla je poruka zahteva od drugog klijenta, ova poruka se smešta u slobodan bafer, da bi bila sačuvana i kasnije isporučena serveru.
- Ako ne postoji slobodan bafer, poruka zahteva se odbacuje, uz, eventualno, slanje odgovarajuće upravljačke poruke drugom klijentu.
- Svaka od poruka, koje se razmenjuju između procesa, se sastoji:

- od upravljačkog dela poruke
- od sadržaja poruke.
- Upravljački deo poruke obuhvata:
  - adresu odredišnog procesa (kome se poruka upućuje)
  - adresu izvorišnog procesa (od koga poruka kreće, a kome se, eventualno, kasnije upućuje odgovor)
  - opis poruke (njenu vrstu, njen redni broj i slično)
  - Adresa (odredišnog ili izvorišnog) procesa sadrži jedinstven redni broj računara, kome proces pripada (a po kome se razlikuju svi računari), kao i port (jedinstven redni broj po kome se razlikuju procesi, koji pripadaju istom računaru).
  - Na osnovu rednog broja računara, mrežni kontroler utvrđuje da li prihvata ili propušta poruku, a na osnovu porta se određuje proces, kome se poruka isporučuje.
  - U toku programiranja, zgodnije je, umesto ovih rednih brojeva, koristiti imena za označavanje i računara i procesa.
  - Korespondenciju između imena i rednih brojeva uspostavlja već pomenuti server imena.
  - Ove podatke o sebi ostavljaju svi serveri, kada se, na početku svoje aktivnosti, obrate serveru imena.

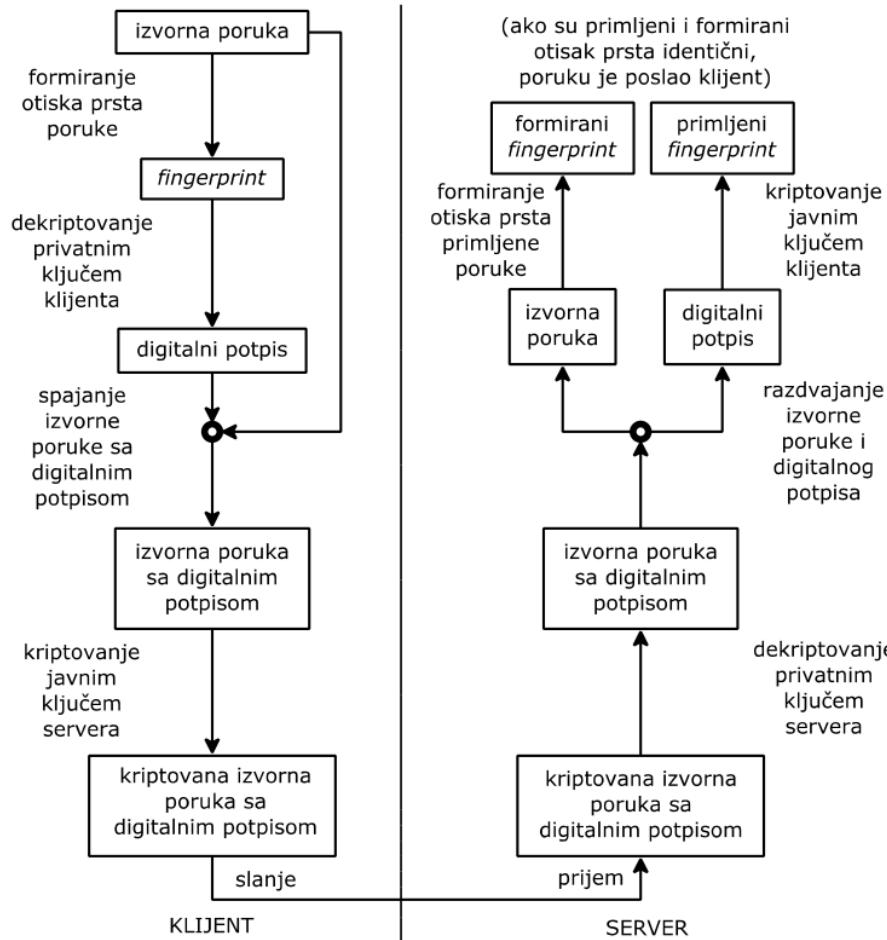
## **Problemi razmene poruka**

- Slaba tačka razmene poruka je sigurnost, jer su komunikacione linije pristupačne svim korisnicima, pa je svaki od njih u poziciji da preuzima tuđe poruke i da šalje poruke u tuđe ime.
- Sprečavanje preuzimanja tuđih poruka se zasniva na kriptovanju (encryption) poruka, a sprečavanje slanja poruka u tuđe ime se zasniva na nedvosmislenoj međusobnoj identifikaciji procesa (authentication).
- U slučaju simetrične kriptografije, da bi klijent i server mogli da razmenjuju poruke sa kriptovanim sadržajima, oba moraju da znaju i algoritam kriptovanja i zajednički interni ključ kriptovanja.
- Pod prepostavkom da je algoritam poznat svim procesima, a da interni ključ kriptovanja treba da znaju samo klijent i server, koji razmenjuju poruke, javlja se problem kako dostaviti interni ključ kriptovanja samo pomenutom klijentu i serveru.

- U tome može da pomogne poseban server, u koga svi procesi imaju poverenje i koji se, zato, naziva poverenik.
- Pri tome se podrazumeva da poverenik poseduje unapred dogovoren poseban ključ kriptovanja za komunikaciju sa svakim procesom.
- Zahvaljujući tome, klijent može da pošalje povereniku poruku, koja sadrži ime klijenta i ime servera sa kojim klijent želi da ostvari sigurnu komunikaciju.
- Sadržaj ove poruke je kriptovan ključem, koji je poznat samo klijentu i povereniku, tako da je razumljiv samo za poverenika, a on, na osnovu adrese izvorišnog procesa iz upravljačkog dela ove poruke, može da pronađe ključ za dekriptovanje njenog sadržaja.
- Sadržaj ove poruke je kriptovan ključem, koji je poznat samo povereniku i serveru, tako da je razumljiv samo za servera.
- Takođe, poverenik šalje poruku i klijentu, koja sadrži interni ključ kriptovanja i ime servera.
- Sadržaj ove poruke je kriptovan ključem, koji znaju samo poverenik i klijent, tako da je razumljiv samo za klijenta.
- Na ovaj način, samo klijent i samo server dobiju interni ključ kriptovanja za sigurnu međusobnu komunikaciju i ujedno se obavi njihova međusobna identifikacija, tako da se drugi procesi ne mogu nepramećeno umešati u njihovu komunikaciju.
- Opisano ponašanje klijenta i servera predstavlja oblik sigurnosnog protokola (security protocol, cryptographic protocol, encryption protocol).



- Ako se sigurna razmena poruka zasniva na asimetričnoj kriptografiji, tada je uloga poverenika da čuva javne ključeve i tako osigura međusobnu identifikaciju procesa.
- Znači, kada je potrebno ostvariti sigurnu komunikaciju između dva procesa, oni se obraćaju povereniku, da bi dobili javni ključ svog komunikacionog partnera.
- Za komunikaciju sa poverenikom ovi procesi koriste unapred dogovorenij javni ključ poverenika, a za komunikaciju sa njima poverenik koristi njihove unapred dogovorene javne ključeve.
- Asimetrična kriptografija, sa komutativnim algoritmima kriptovanja i dekriptovanja, omogućuje i digitalno potpisivanje poruka, radi neopozivog pripisivanja poruke njenom pošiljaocu.
- Digitalni potpis (digital signature) se šalje uz poruku.
- On sadrži podatke koji jednoznačno reprezentuju poruku, pa predstavljaju otisak prsta poruke (fingerprint, cryptographic checksum).
- Otisak prsta poruke formiraju jednosmerne funkcije (one-way functions) na osnovu sadržaja poruke.
- Digitalni potpis nastane kada se otisak prsta poruke dekriptuje (transformiše) primenom algoritma dekriptovanja i privatnog ključa.
- Primalac poruke kriptuje (retransformiše) digitalni potpis primenom algoritma kriptovanja i javnog ključa.
- Ako se rezultat kriptovanja digitalnog potpisa poklapa sa otiskom prsta primljene poruke, tada je poruka nedvosmisleno stigla od pošiljaoca.



- Sigurnu komunikaciju klijenta i servera mogu ometati drugi procesi zlonamernim retransmisijama starih poruka, ili izmenom sadržaja poruka.
- Ugrađivanjem u sadržaj poruke njenog rednog broja, mogu se otkriti retransmisije starih poruka, a ugrađivanjem u sadržaj poruke kodova za otkrivanje i oporavak od izmena sadržaja, mogu se otkriti, pa i ispraviti izmene sadržaja poruka.

### Razlika klijenata i servera

- Različita uloga, koju klijent i server imaju u toku međusobne komunikacije (saradnje), je prirodna posledica njihove namene.
- Iz toga proizlaze i razlike u njihovoј internoj organizaciji.

- Dok je za klijenta prihvatljivo da njegova aktivnost bude strogo sekvencijalna, za servera stroga sekvencijalnost njegove aktivnosti znači manju propusnost i sporije pružanje usluga.
- To je najlakše ilustrovati na primeru servera datoteka, zaduženog za pružanje usluga, kao što je čitanje ili pisanje datoteke.
- Stroga sekvencijalna aktivnost ovoga servera bi izazvala zaustavljanje njegove aktivnosti, radi usluživanja jednog klijenta, dok kontroler ne prenese blok sa sadržajem datoteke između masovne i radne memorije.
- U međuvremenu ne bi bilo usluživanja drugih klijenata, čak i ako bi se njihovi zahtevi odnosili na blokove datoteka, prisutne u baferima radne memorije.
- Ovakva sekvencijalnost nije prisutna kod tradicionalnih operativnih sistema, jer nakon zaustavljanja aktivnosti jednog procesa u modulu za rukovanje datotekama, drugi proces može nastaviti aktivnost u istom modulu.
- Zato je za servere potrebno obezbediti više niti.
- Pri tome, svaka od niti, u okviru istog servera, opslužuje različitog klijenta, a broj ovih niti zavisi od broja postavljenih zahteva i menja se u vremenu.
- Postojanje više niti zahteva njihovu sinhronizaciju, dok pristupaju globalnim (statičkim) promenljivim servera.
- Iako je primena više niti tipična za servere, ona ima svoje opravdanje i kod klijenata, jer može poboljšati njihovo ponašanje.

## **Poziv operacije udaljenog objekta**

- Poziv udaljene operacije ima kao alternativu poziv operacije udaljenog objekta (Remote Method Invocation - RMI).
- Za poziv operacija nekog objekta je potrebno raspolagati njegovom referencom i poznavati operacije koje su za njega definisane.
- Pri tome ne smeta ako je objekat udaljen, odnosno ako se ne nalazi na istoj mašini kao i proces koji poziva operaciju dotičnog objekta.
- Dobijanje reference udaljenog objekta, postupak poziva njegove operacije i dobijanje rezultata izvršavanja pozvane operacije se suštinski ne razlikuju od rešavanja sličnih problema kod poziva udaljene operacije.

# **Distribuirani sistem datoteka**

## **Distribuirani sistem datoteka**

- Distribuirani sistem datoteka obuhvata hijerarhijsku organizaciju datoteka čiji delovi se nalaze na raznim računarima.
- Ovakav distribuirani sistem datoteka se može oslanjati na više servera imenika i na više servera datoteka.
- Serveri imenika podržavaju hijerarhijsku organizaciju datoteka, a serveri datoteka podržavaju pristup sadržaju (običnih) datoteka.
- U imenicima, kojima rukuju serveri imenika, uz imena datoteka, odnosno uz imena imenika, ne stoje samo redni brojevi deskriptora datoteka, nego i redni brojevi servera datoteka, odnosno servera imenika, kojima pripadaju pomenuti deskriptori.
- U distribuiranom sistemu datoteka pristup datoteci podrazumeva konsultovanje servera imena, radi pronalaženja servera imenika, od koga kreće pretraživanje imenika.
- Pretraživanje imenika može zahtevati kontaktiranje različitih servera imenika, dok se ne stigne do servera datoteka sa traženom datotekom.
- Serveri imenika i datoteka mogu da ubrzaju pružanje usluga, ako kopiju često korišćenih podataka čuvaju u radnoj memoriji.
- Ubrzjanju pružanja usluga doprinosi i repliciranje datoteka, da bi one bile fizički bliže korisnicima.
- Međutim, to stvara probleme, kada razni korisnici istovremeno menjaju razne kopije iste datoteke, jer se tada postavlja pitanje koja od izmena je važeća.
- Za distribuirani sistem datoteka zaštitu datoteka je primerenije zasnovati na dozvolama (capability), nego na pravima pristupa.
- To znači da u okviru deskriptora datoteka, odnosno imenika, ne postoje navedena prava pristupa za pojedine grupe korisnika, nego se za svaku datoteku, odnosno imenik, generišu različite dozvole.
- One omogućuju razne vrste pristupa datoteci, odnosno imeniku.
- Da bi klijent dobio neku uslugu, on mora da poseduje odgovarajuću dozvolu, koju prosleđuje serveru u okviru zahteva za uslугom. Dozvola sadrži:
  - redni broj servera,

- redni broj deskriptora datoteke (odnosno, deskriptora imenika)
- oznaku vrste usluge
- oznaku ispravnosti dozvole
- Sadržaj dozvole je zaštićen kriptovanjem, tako da nije moguće, izmenom oznake vrste usluge prepraviti dozvolu.
- Pre pružanja usluge, server dekriptuje sadržaj dozvole, i proverava da li je ona ispravna i da li se njena oznaka vrste usluge podudara sa zatraženom uslugom.
- Dozvole deli server na zahtev klijenata, koji ih čuvaju i po potrebi prosleđuju jedan drugom.
- Pri tome se podrazumeva da klijent, stvaralac datoteke, po njenom stvaranju automatski dobije dozvolu za sve vrste usluga, koja uključuje i uslugu stvaranja drugih, restriktivnijih dozvola.
- Kada želi da poništi određenu dozvolu, server samo proglaši njenu oznaku ispravnosti nevažećom.
- Prednost zasnivanja zaštite datoteka na dozvolama umesto na pravima pristupa je u tome da prvi pristup ne zahteva razlikovanje korisnika, niti njihovo označavanje.
- To je važno, jer rukovanje jedinstvenim i neponovljivim oznakama korisnika u distribuiranom računarskom sistemu nije jednostavno.
- Sem toga, dozvole omogućuju veću selektivnost, jer grupišu korisnike po kriterijumu posedovanja dozvole određene vrste, a ne na osnovu njihovih unapred uvedenih (numeričih) oznaka.
- Komercijalni distribuirani sistem datoteka nastaje spajanjem lokalnog i udaljenog sistema datoteka.
- Osnovu za stvaranje komercijalnog distribuiranog sistema datoteka nudi, na primer, NFS (Sun Microsystem's Network File Service).
- Pristup udaljenom sistemu datoteka podrazumeva mrežnu komunikaciju lokalnog klijenta i udaljenog mrežnog servera datoteka, o čijim detaljima korisnik ne mora da vodi računa.

## **Raspoređivanje procesa u distribuiranom računarskom sistemu**

- Cilj raspoređivanja procesa u distribuiranom računarskom sistemu je ostvarenje najboljeg iskorišćenja računara, ili

ostvarenje najkraćeg vremena odziva, odnosno, najbržeg usluživanja korisnika.

- Zadatak raspoređivanja komplikuju razlike između računara, jer u opštem slučaju svaki računar ne može da prihvati svaki izvršni oblik programa, pošto su izvršni oblici programa vezani za procesor, za raspoloživu radnu memoriju i slično.
- Raspoređivanje komplikuje i zahtev za omogućavanje migracije procesa sa računara na računar, da bi se prezaposlen računar rasteretio, a nezaposlen zaposlio.
- Raspoređivanje je olakšano, ako su unapred poznate karakteristike opterećenja računara, odnosno vrsta i broj njihovih procesa.
- Kod rasporedivanja procesa po računarima, važno je voditi računa o saradnji procesa i procese, koji tesno međusobno sarađuju, raspoređivati na isti računar.

## Distribuirana sinhronizacija

- Saradnja procesa, aktivnih na raznim računarima, zatvara njuhovu sinhronizaciju, što se ostvaruje razmenom poruka.
- Pri tome, najjednostavniji način za ostvarenje sinhronizacije se zasniva na uvođenju procesa koordinatora.
- Njemu se obraćaju svi procesi, zainteresovani za sinhronizaciju, a koordinator donosi odluke o njihovoj sinhronizaciji.
- Tako, ako je potrebno, na primer, ostvariti međusobnu isključivost procesa u pristupu istoj datoteci, svi procesi traže od koordinatora dozvolu za pristup, a on dozvoljava uvek samo jednom procesu da pristupi datoteci.
- Na sličan način se može ostvariti i uslovna sinhronizacija.
- Za razliku od ovakvog centralizovanog algoritma sinhronizacije, koji se zasniva na uvođenju koordinatora, postoje i distribuirani algoritmi sinhronizacije, koji se zasnivaju na međusobnom dogovaranju procesa, zainteresovanih za sinhronizaciju.
- Distribuirani algoritmi sinhronizacije zahtevaju sredstva za grupnu komunikaciju procesa, odnosno, efikasna sredstva koja omogućuju da jedan proces pošalje poruke svim ostalim procesima iz grupe procesa, zainteresovanih za sinhronizaciju, i da od njih primi odgovore.
- Pored veće razmene poruka, distribuirani algoritmi sinhronizacije su komplikovani od centralizovanih algoritama, a

pri tome ne nude prednosti, tako da je njihov razvoj više od principijelnog, nego od praktičnog značaja.

- Za distribuirane operativne sisteme nije samo bitno da omoguće efikasnu sinhronizaciju procesa, nego i da podrže poseban oblik sinhronizacije procesa, koji obezbeđuje da se obave ili sve operacije iz nekog niza pojedinačnih operacija, ili ni jedna od njih.
- Ovakav niz operacija se naziva transakcija, a transakcije, koje imaju svojstvo da se obave u celosti ili nikako, se nazivaju atomske transakcije (atomic transaction).
- Primer niza operacija, za koje je neophodno da obrazuju atomsku transakciju, je prebacivanje nekog iznosa sa računa jedne banke na račun druge banke.
- Pri tome, proces klijent, koji u ime korisnika obavlja ovo prebacivanje, kontaktira dva servera, koji reprezentuju dve banke, da bi obavio transfer iznosa sa jednog na drugi račun.
- Transfer se mora obaviti tako, da drugi klijenti mogu videti oba računa samo u stanju ili pre, ili posle transakcije.
- Znači, za atomske transakcije je neophodno da budu međusobno isključive, ako pristupaju istim podacima, ali i da njihovi rezultati budu trajni, da se jednom napravljena izmena ne može izgubiti.
- Saradnja procesa, aktivnih na raznim procesorima, otvara mogućnost pojave mrtve petlje.
- Pri tome, u uslovima distribuiranog računarskog sistema, algoritmi za izbegavanje pojave mrtve petlje, odnosno za otkrivanje i za oporavak od pojave mrtve petlje su još neefikasniji i sa još manjim praktičnim značajem, nego u slučaju centralizovanog (jednoprocesorskog) računara.
- Zato, u uslovima distribuiranog računarskog sistema, kada nije prihvatljiv pristup ignorisanja problema mrtve petlje, preostaje da se spriči njena pojava, na primer, spričavanjem ispunjenja uslova, neophodnih za pojavu mrtve petlje.

## Svojstva distribuiranog računarskog sistema

- Distribuirani računarski sistem je zamišljen tako da integriše mnoštvo računara u moćan multiračunarski sistem.
- Na taj način je moguće od više jeftinih i malih računara napraviti moćan multiračunarski sistem povoljne cene.

- Ovakav multiračunarski sistem, uz to, nudi i veću pouzdanost, jer kvar pojedinačnog računara nije fatalan za ceo sistem, kao i mogućnost proširenja, jer je moguće naknadno dodavanje računara u sistem.
- Pored integrisanja pojedinačnih računara u multiračunarski sistem, distribuirani računarski sistem omogućuje i deljenje skupih resursa ovakvog multiračunarskog sistema između više korisnika, a nudi i prilagodljivost zahtevima korisnika, željenu raspoloživost i predvidivost odziva, pa, čak, i veću sigurnost, jer korisnici mogu da čuvaju poverljive podatke na svom računaru, koga fizički štite i čije korišćenje mogu da kontrolišu.

## **Implementacija distribuiranog OS**

- Zadatak distribuiranog operativnog sistema je da objedini sve računare distribuiranog računarskog sistema, tako da korisnik ne vidi pojedine računare, nego jedinstven sistem, koji pruža usaglašene usluge.
- Sve ovakve usluge se pružaju na uniforman način, koji zanemaruje mesto i druge specifičnosti pružanja usluge.
- Uniforman način pružanja usluga podrazumeva sakrivanje različitosti (heterogenosti) računara od kojih je obrazovan distribuirani računarski sistem.
- To se može postići, ako se iznad raznorodnih operativnih sistema pojedinih računara distribuiranog računarskog sistema napravi posebna distribuirana softverska platforma (middleware) za razvoj distribuiranih softverskih sistema.
- Ona ima ulogu distribuiranog operativnog sistema.
- Distribuirana softverska platforma je obično specijalizovana tako da nudi konzistentan skup operacija, koje omogućuju razvoj željene vrste distribuiranih softverskih sistema.
- Ovakav skup operacija podržava saradnju komponenti koje obrazuju distribuirani softverski sistem.
- U tom pogledu se može govoriti o raznim distribuiranim softverskim platformama, poput one namenjene za podršku distribuiranih dokumenata (World Wide Web), ili one namenjene za podršku distribuiranom objektno orijentisanim programiraju (CORBA, Java middleware, .NET).
- U osnovi ovakvih sistema se krije klijent-server model.

- Server se nalazi na strani rukovaoca distribuiranim dokumentima (web site) ili rukovaoca objektima, a klijent se nalazi na strani korisnika distribuiranih dokumenata (web browser) ili pozivaoca operacija udaljenih objekata (pri čemu je pozivanje ovih operacija zasnovano na RPC mehanizmima).

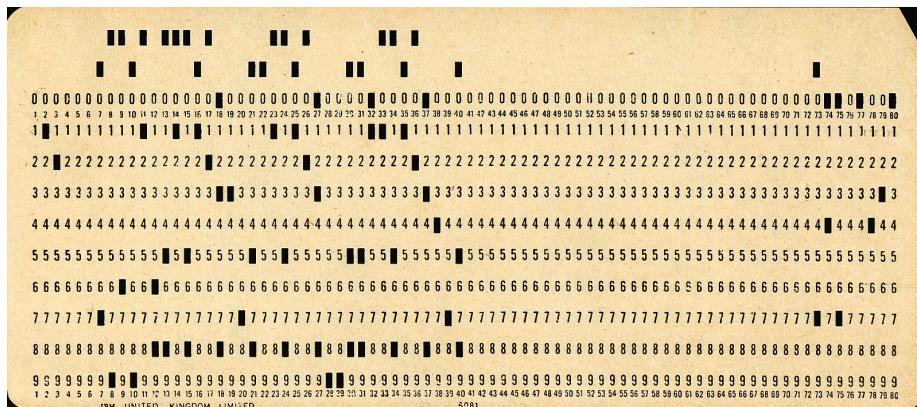
## Pitanja

### Pitanja

- Šta karakteriše operativne sisteme realnog vremena?
- Šta karakteriše multiprocesorske operativne sisteme?
- Koje module sadrži mikrokernel?
- Šta karakteriše poziv udaljene operacije (RPC)?
- Šta radi klijentski potprogram?
- Za šta su zaduženi serverski potprogrami?
- Koji problemi su vezani za poziv udaljene operacije?
- Šta podrazumeva dinamičko linkovanje klijenta i servera?
- Koje operacije podržava protokol razmene poruka između klijenta i servera?
- Za šta su zadužene sistemske operacije koje ostvaruju protokol razmene poruka?
- Šta sadrže poruke koje razmenjuju klijent i server?
- Šta je potrebno za sigurnu razmenu poruka između klijenta i servera?
- Šta karakteriše digitalni potpis?
- Od čega zavisi propusnost servera?
- Šta sadrže dozvole na kojima se zasniva zaštita datoteka u distribuiranom sistemu?
- Šta karakteriše distribuiranu sinhronizaciju?
- Šta karakteriše distribuirani računarski sistem?
- Šta karakteriše distribuiranu softversku platformu?— title: Komunikacija sa Operativnim Sistemom author: Veljko Petrović date: 2023-06 —

## **Kratka, nekompletna istorija ključnih momenata u interfejsima**

## Početak



(Izvor: Pete Birkinshaw CC-Generic-Attribution 2.0)

## Računari bez interfejsa

- Prvobitni računari nisu ni imali interfejs u klasičnom značenju
  - Bili su mašine koje transformišu podatke iz jedne forme u drugu: a podaci su ulazili (i verovatno izlazili) na apsolutno isti način, kao bušene kartice
  - Bušene kartice su znatno starije od računara: najraniji poznat primer jesu kartice korišćene za 'programiranje' automatizovanih razboja, tzv. Žakardove kartice

## Bušene kartice

- Semjon Korsakov i Čarls Bebidž su predložili da se bušene kartice mogu interpretirati kao univerzalan način za čuvanje informacija.
  - Prva osoba koja je to uradila praktično je Herman Hollerit koji je iskoristio kartice ne bi li automatizovao analizu podataka popisa stanovništva 1890 u SAD.
  - Ovo je dovelo do čitave gomile mašina koje su uzimale kartice i sortirale ih ili pak ih prebrojavale po raznim kategorijama kroz tabulaciju, sumiranje, i slične aktivnosti.
  - Ovo su bile poznate kao 'biznis mašine' koje opstaju još samo kao lingvistički fosil u imenu IBM: International Business Machines.

## Uloga ranih računara

- Računari su, dakle, smatrani samo naročito bistrim biznis mašinama koji se mogu navesti da rade kompleksnije operacije i to operacije koje se mogu promeniti
- Nije bilo ni *pokušaja* da se njima da ikakav smislen interfejs tokom prvih godina njihove primene.

## Prvi interaktivni računari

- Svi računari su podržavali minimalan stepen interaktivnosti zbog debagovanja i testiranja
- Verovatno prvi računar koji bi mogli opisati kao donekle interaktiv je bio SSEM (small scale experimental machine) koji je konstruisan u Mancesteru 1948.
- SSEM je imao mehaničke prekidače koji su mogli da utiču na ponašanje računara i ekran... donekle.
- Ekran je bio više slučajnost: sistem je koristio Viljems-Kilburn tip memorije koji je, bizarno, bio katodna cev.

## SSEM



## Prvi interaktivni računari

- U zavisnosti da li se prikaže jedan vizuelni oblik ili drugi, katodna cev je nanelektrisana pozitivno ili negativno.
- Efektivno na lokacijama na ekranu sistem je prikazivao tačke ili crticice i onda pomoću detektorske ploče je hvatao pozizivan ili negativan naboј i tako očitavao memoriju.
- Prikaz na ekranu je trajao samo oko 200ms, ali je automatski proces mogao da 'osveži' memoriju tako da ona traje.
- Ovo je omogućilo gigantskih 1024 bita memorije.
- Budući da je već koristio vizuelni displej za svoju memoriju nije ništa 'koštalo' da se taj signal pošalje i na običan displej i time da vizeuelni uvid u stanje cele memorije
- Stoga, iako je ovo bio *tehnički* ekran, nije bio ekran u današnjem smislu.
- Među prvim interaktivnim računarima (naročito onima koji su dobro dokumentovani) je TX-0 računar, razvijen na Lincoln laboratoriji 1955/6 i kasnije modifikovan u okviru RLE na MIT-u.
- Svrha interaktivnih osobina je bila verovatno zato što je računar napravljen da bude test za mnogo veći računarski sistem (Whirlwind) koji, zbog svoje vojne primene, je morao da bude interaktivan.
- Bilo kako bilo, sistem je imao način da se izdaju komande direktno računaru (preko teleprinterske tastature) i čak i ekran efektivne rezolucije 512x512.
- Valja napomenuti da je ekran bio, efektivno, osciloskop, te vektorski displej tako da je informacija o rezoluciji malo neprecizna.

## **TX-0**



(Izvor: Muzej Računarske Istorije, <https://www.computerhistory.org/pdp-1/2e1b209cb40237b91228cdf26a60e3f8/>)

## **PDP-1**

- Jako bitan za razvoj interfejsa je bio PDP-1, računar koji je nastao 1959 kao direktni naslednik TX-0 mašine i koji je od početka dizajniran kao visoko interaktivna mašina.
- Posedovao je CRT ekran, teleprinter 'tastaturu,' čitačke bušenih papirnih traka (što je bila zamena za disk, diskete, i svaki drugi način čuvanja podataka) i čak analogni ulaz preko 'svetlosne olovke.'
- Ovo je rana preteča miševa bazirana na tome da katodne cevi osim što proizvode svetlost takođe 'cure' beta zračenje
- Svetlosna olovka je detektor takvog značenja: na osnovu tajminga elektronskog topa katodne cevi i vremena kada ta olovka detektuje impuls zračenja moguće je precizno odrediti gde se nalazi na ekranu.



(Izvor: Aleksej Komarov, CC-AttributionShareAlike 4.0)

## Pionirski interaktivni programi

- PDP-1 i TX-0 su bili mesto gde su razvijeni razni bitni rani programi za interaktivn rad
- Colossal Typewriter koji je bio prvi tekst editor opšte namene
- TECO koji je bio verovatno prvi programerski tekst editor: jako kompleksan jezik za transformaciju teksta sa papirne trake. Isprogramirali bi sekvencu TECO komandi i onda pustili da ona izmeni tekst i onda ga snimila na drugu traku. TECO je direktn predak modernog Emacs editora.
- Naravno, ima jedan interaktivni komad softvera prvo razvijen za PDP-1 koji je znantno bitniji od svih ostalih.

## Spacewar!



(Spacewar! – Jedan od kandidata za prvu računarsku igru (1962))

### Jedan veliki problem

- Ništa od ovoga nije slučaj interakcije sa *operativnim sistemom*.
- Svi ovi interfejsi pričaju sa mašinom vrlo neposredno i pristupaju hardveru sasvim direktno.
- Programi koji imaju civilizovaniji interfejs i dalje pričaju sa hardverom direktno i zauzimaju kontrolu nad računarom u potpunosti.
- Operativni sistemi su neraskidivo vezani za problem višestruke upotrebe jednog računarskog sistema, tkzv. "time-sharing." Operativni sistemi su postojali i ranije, ali nisu imali isti značaj.

### Interfejs multiprogramske operativne sistema

- Ako delite hardver sa drugim korisnicima očigledno je da neko mora da stoji između vas i hardvera i sprečava ekskluzivan pristup: to je bila tada i ostala danas uloga operativnog sistema.

- Ovo pak znači da je neophodno pričati sa operativnim sistemom i ‘zamoliti’ ga da radi stvari ili kroz sistemske podprograme (programska nivo komunikacije - danas poznato kao sistemski pozivi) ili kroz izdavanje nekakvih komandi kao korisnik (korisnički nivo pristupa).
- Ovo je u igru uvelo koncept *shell* programa.

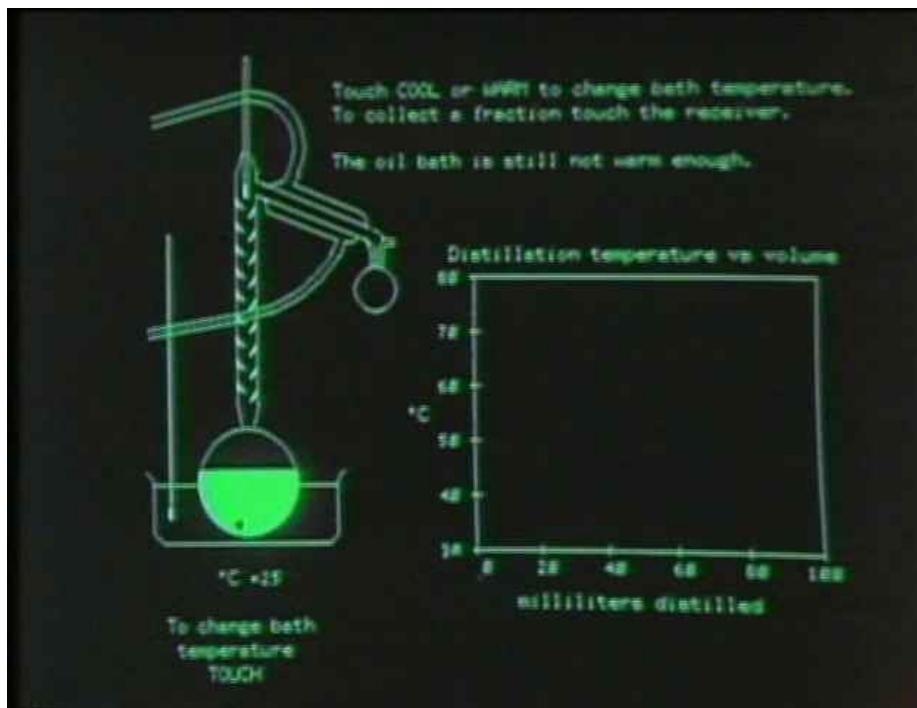
## **Shell**

- Shell možete zamisliti kao poseban program (ili u ranim sistemima direktni deo operativnog sistema) koji vam omogućava da nekako zadajete komande koje će shell pretvoriti u operacije operativnog sistema.
- Prvi operativni sistemi su, bez izuzetka, imali takozvani interfejs komandne linije gde se komande kucaju ili direktno ili kroz nešto više softisticiran komandni jezik koji omogućava da se više komandi kombinuje.
- Računari i danas podržavaju istu stvar to su cmd i PowerShell u Windows-u, odnosno sh, bash, zsh, fish, xonsh... na POSIX sistemima.

## **Rana upotreba grafike - PLATO sistem**

- PLATO je bio sistem razvijan na Stanfordu od ranih šezdesetih do sredine sedamdesetih
- Bio to sistem specijalne namene, specifično, bio je namenjen interaktivnom učenju uz pomoć računara.
- PLATO je imao, za svoje vreme, nezapamćeno veliki broj korisnika i veliku interaktivnost.
- Takođe je imao veoma neobičnu formu displeja gde se mogao mešati displej koji ima i bitmap i vektorske elemente.

## PLATO



(Izvoj: Džejson Skot, CC SA-Attr-3.0 US)

## Rana upotreba grafike - NLS i Majka Svih Demoa

- Većina ideja čiji će razvoj obeležiti 90-te i rane 2000-te su bile u opticaju mnogo ranije.
- Dramatičan primer ovoga je računarski sistem (i njegova demonstracija poznata, po svom izuzetnom uticaju, kao 'Majka Svih Demoa') poznat kao NLS (oNLine System).
- NLS je demonstrirao njegov tvorac, Daglas Engelbart 1968 i dramatično promenio razvoj računarskih interfejsa.

## NLS

- Demonstracija NLS-a je publici, po prvi put, prikazala sledeće stvari
  - Računarski miš
  - Grafički editor teksta sa više procesa i mogućnosti interaktivnog editovanja sa više učesnika.
  - Video pozivi kombinovani sa kolaborativnim editovanjem
  - Računarski hipertekst

- Kao pokušaj futurološkog predviđanja budućnosti, NLS je možda najveći zabeleženi uspeh, budući da ono što je NLS demonstrirao je, efektivno, tipična kancelarija cca 2010.
- Naravno, sam NLS nije bio ekonomski moguć: demo je samo mogao biti rezultat guranja tehnologije do samih granica.
- Uprkos tome ideje koje su tu prezentovane su se raširile po celom svetu i dovele vremenom do prvih komercijalno mogućih grafičkih računarskih interfejsa.

## Nastavak rada na NLS

- Tim koji je radio na NLS sistemu je uglavnom završio u kompaniji Ziroks (Xerox) gde su razvili eksperimentalni Xerox Alto računar (1973) koji je verovatno prvi lični računar napravljen da ima grafički korisnički interfejs.
- Nikada nije komercijalnizovan, ali je imao veliki uticaj na dalji rad.

## Xerox Star

- Nešto kao što je NLS bio predviđanje budućnosti, tako je i Xerox Star bio daleko ispred svog vremena
- To je bio računar koji je izašao 1981 (iste godine kao i IBM PC) koji je imao grafički korisnički interfejs (sa ikonama, prozorima, i dekstop metaforom), na ekranu za svoje doba *izuzetno* visoke rezolucije (1024x808).
- Takođe je podržavao Ethernet mrežu, servere za deljenje fajlova, štampača, i slanje elektronske pošte.
- Softver je mogao da se piše koristeći objektno orijentisan programski jezik.



(Preuzeto sa [https://en.wikipedia.org/wiki/File:Xerox\\_Star\\_8010\\_workstation.jpg](https://en.wikipedia.org/wiki/File:Xerox_Star_8010_workstation.jpg)  
nejasno je čiji je copyright)

- Ovaj računar je, dakle, efektivno nudio mogućnosti kancelarijskog računara iz sredine devedesetih.
- Zašto niste (verovatno) čuli za njega?
- Pa, prvo, koštao je oko \$50 000 u modernom novcu.

- Drugo, kao i bilo koja tehnologija koja je tek izmišljena postojali su razni bagovi uključujući i spektakularno spori fajl sistem.

### **Efekti Xerox Star-a**

- Možda nije slučajnost da pošto je linija propala na tržištu, ključni ljudi koji su radili na ovom sistemu su promenili poslodavca i počeli da rade u malim, opskurnim softverskim firmama kao što je Microsoft i Apple.
- Ideje koje su razvijene preko NLS, Alta, i Star-a su se raširile kroz industriju i moderni koncept grafičkog korisničkog interfejsa baziranog na prozorima, ikonama, i nekim sredstvom za pokazivanje (miš, ekran osetljiv na dodir...) duguje jako puno ovom ranom poslu.

## **Sistematizacija Komunikacija sa Operativnim Sistemom**

### **Programski nivo komunikacije sa OS**

- Komunikacija sa operativnim sistemom na programskom nivou se ostvaruje pozivanjem sistemskih operacija.
- Znači da bi se u toku izvršavanja korisničkog programa dobila neka usluga od operativnog sistema, potrebno je pozvati odgovarajuću sistemsku operaciju.
- Na primer, da bi se preuzeo znak sa tastature, u korisničkom programu je neophodno navesti poziv odgovarajuće sistemske operacije.

### **Interaktivni nivo komunikacije sa OS - CLI**

- Interaktivni nivo korišćenja operativnog sistema se ostvaruje pomoću komandi komandnog jezika.
- One, na primer, omogućuju rukovanje datotekama i procesima.
- Najjednostavniju komandu komandnog jezika predstavlja putanja izvršne datoteke.
- Kao operand ovakve komande se može, opet, javiti putanja datoteke, ako je komanda namenjena za rukovanje datotekama.
- Prema tome, na prethodni način oblikovana komanda započinje operatorom u obliku putanje izvršne datoteke, koja opisuje rukovanje, a završava operandom (ili operandima) u obliku putanja datoteka, kojima se rukuje. Tako:

- kopiraj godina1.txt godina2.txt
- Ili u slučaju Linux okruženja kroz bash cp godina1.txt godina2.txt
- U ovom primeru se pretpostavlja da **radni imenik** obuhvata izvršnu datoteku sa imenom kopiraj i tekst datoteku godina1.txt.
- Rani imenik je nešto što je kombinacija programske i interaktivne interakcije: svaki proces mora da u svakom trenutku ima svoj radni imenik, i korisnik koji koristi računar preko komandnog interfejsa se uvek nalazi u nekom direktorijumu koji predstavlja kontekst naših operacija nad fajlovima.

## Interaktivni nivo komunikacije sa OS

- Prethodno opisani način zadavanja komandi odgovara znakovnom komandnom jeziku.
- Komandni jezik može olakšati zadavanje komandi, ako omogući korisniku da operator komande bira u spisku operatora (menu), umesto da ga pamti i u celosti navodi.
- Spisak operatora se prikazuje na ekranu, a izbor operatora se vrši pomoću namenskih dirki tastature ili miša.
- Nakon izbora operatora sledi, po potrebi, dijalog u kome korisnik navodi (ili opet bira) operand (operande) komande.
- Ovakvi komandni jezici se nazivaju grafički komandni jezici (menu driven user interface, graphical user interface - GUI).
- Oni još više pojednostavljaju zadavanje komandi, ako korisniku omogućuju da ne bira operator, nego samo operative komandi.
- U ovom slučaju, izbor operaanda se svodi na izbor nekog od imena datoteka, prikazanih na ekranu, a operator se podrazumeva ili na osnovu tipa odabrane datoteke, ili, eventualno, na osnovu upotrebljene namenske dirke.

## Važnost GUI-ja i metafore

- Svaki GUI mora nužno operisati sa nekakvom metaforom: grafičkim i konceptualnim jezikom koji mapira ponašanje računara na nekakve simbole kojima se grafički manipuliše.
- Ako ste ikada 'prevukli' fajl sa mesta na mesto, to znači da ste uspešno konceptualizovali računar preko metafore: grafička ikona menja fajl, 'prozori' interfejsa menjaju kontekste rada

(različita mesta na računaru), a akt držanja pristinutog tastera miša (ili prsta na ekranu) je gest premeštanja.

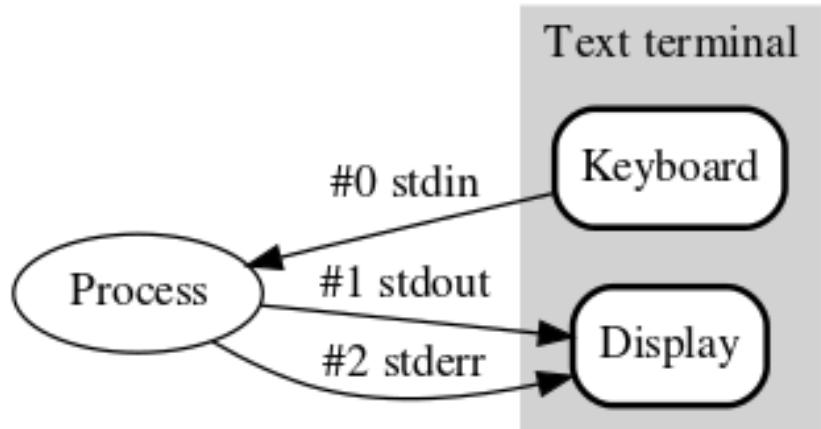
## Interaktivni nivo komunikacije sa OS

- U svakom slučaju, zadatak komandnog jezika je da omogući korisniku da zada komandu, koja precizno određuje i vrstu rukovanja i objekat rukovanja, a zadatak interpretiranja komande je da pokrene proces, u okviru čije aktivnosti usledi rukovanje, zatraženo komandom.

## Znakovni komandni jezici

- Izgled, način rada i mogućnosti interpretera znakovnog komandnog jezika (command language interpreter, shell) zavise od ciljeva, koje komandni jezik treba da ostvari.
- Ciljevi znakovnih komandnih jezika obuhvataju:
  - omogućavanje izvršavanja pojedinih (korisničkih) programa
  - omogućavanje kombinovanja izvršavanja više (korisničkih) programa
  - omogućavanje pravljenja komandnih datoteka (command file, shell script).
- Interpreter znakovnog komandnog jezika ostvaruje prethodne ciljeve tako što sa standardnog ulaza prima niz znakova, koji obrazuju komandu, prepoznaje u tom nizu znakova operator komande (i, eventualno, njene operande) i preduzima zahtevanu akciju.
- Rezultat svoje akcije ovaj interpreter prikazuje na standardnom izlazu.
- Prilikom preduzimanja zahtevane akcije, interpreter znakovnog komandnog jezika se oslanja na sistemske operacije.
- Pri tome, on koristi delove komandi, odnosno njen operator i njene operande, kao argumente sistemskih operacija.
- Na primer, do izvršavanja pojedinih korisničkih programa dolazi tako što interpreter znakovnog komandnog jezika poziva sistemsku operaciju stvaranja procesa, a kao njene argumente upotrebi operator i operande komande, odnosno putanju izvršne datoteke i putanje datoteka sa obrađivanim podacima.
- Ovi argumenti su namenjeni stvaranom procesu.

- Tako, na primer, kod programskog jezika C parametri funkcije main() omogućuju preuzimanje broja argumenata (argc) iz komandne linije, kao i stringova pojedinih argumenata (argv), jer su ovi smešteni na stek procesa prilikom njegovog stvaranja.
- Interpreter znakovnog komandnog jezika obavlja obradu znakova komande, pre nego što ih iskoristi kao argumente neke sistemske operacije.
- Zahvaljujući tome, u okviru operanada komandi se mogu javiti specijalni znakovi(magic character, wild cards), kao što je, na primer, znak \*. Njegova upotreba je vezana, pre svega, za imena datoteka i namenjena je za skraćeno označavanje grupa datoteka.
- Tako na primer: \*.obj označava sve objektne datoteke u radnom imeniku a d\*1.txt označava sve tekst datoteke u radnom imeniku, čiji prvi deo imena započinju znakom d, a završava cifrom 1.
- Zahvaljujući obradi znakova komande, moguće je interpreteru znakovnog komandnog jezika saopštiti i da preusmeri (redirect) standardni ulaz i standardni izlaz sa tastature i ekrana na proizvoljno odabране datoteke.
- Ovo je važno za pozadinske procese, koji nisu u interakciji sa korisnikom.
- Zahvaljujući preusmeravanju, pozadinski proces ne ometa interaktivni rad korisnika, jer, umesto tastature i ekrana, koristi odabранe datoteke.
- Međutim, da bi se korisnik upozorio na greške u toku aktivnosti pozadinskog procesa, uz standardni izlaz se uvodi i standardni izlaz greške.
- Pošto je standardni izlaz greške namenjen, pre svega, za prikazivanje poruka o greškama, kao podrazumevajući standardni izlaz greške služi specijalna datoteka, koja odgovara ekranu.
- Ova datoteka se otvara za vreme stvaranja procesa, a kao njen indeks za tabelu otvorenih datoteka može da služi vrednost 2.
- I standardni izlaz greške se može preusmeriti na proizvoljnu datoteku, čiji sadržaj tada ukazuje na eventualne greške u toku aktivnosti pozadinskog procesa.
- Izvor: <https://upload.wikimedia.org/wikipedia/commons/7/70/Stdstreams-.pdf>



notitle.svg

- Uobičajeno je da preusmeravanje standardnog ulaza najavljuje znak <, a da preusmeravanje standardnog izlaza (kao i standardnog izlaza greške) najavljuje znak >. Tako, na primer, komanda:
- kompiliraj < program.c > program.obj
- saopštava interpretoru znakovnog komandnog jezika da stvori proces na osnovu izvršne datoteke kompiliraj.bin, pri čemu kao standardni ulaz procesa služi datoteka program.c, a kao njegov standardni izlaz datoteka program.obj.
- U ovom primeru ekran i dalje služi kao standardni izlaz greške.
- Izvršavanje prethodne komande omogućuje kompilaciju C programa, sadržanog u datoteci program.c.
- Rezultat kompilacije se smešta u datoteku program.obj, a eventualne poruke o greškama kompilacije se prikazuju na ekranu.
- Preusmeravanje standardnog ulaza i izlaza predstavlja osnovu za kombinovanje izvršavanja više korisničkih programa.
- Važnost pomenutog kombinovanja se može pokazati na primeru uređivanja (sortiranja) reči iz nekog rečnika po kriterijumu rimovanja.
- Nakon sortiranja reči po ovom kriterijumu, sve reči, koje se rimuju, nalaze se jedna uz drugu.
- Umesto pravljenja posebnog programa za sortiranje reči po kriterijumu rimovanja, jednostavnije je napraviti program za obrtanje redosleda znakova u rečima (tako da prvi i poslednji znak zamene svoja mesta u reči, da drugi i pretposlednji znak zamene

svoja mesta u reči i tako redom) i kombinovati izvršavanje ovog programa sa izvršavanjem postojećeg programa za sortiranje:

- obrni < recnik.txt > obrnuti\_recnik.txt
- sortiraj < obrnuti\_recnik.txt > sortirani\_obrnuti\_recnik.txt
- obrni < sortirani\_obrnuti\_recnik.txt > rime.txt
- Umesto preusmeravanja standardnog ulaza i standardnog izlaza, moguće je nadovezati standardni izlaz jednog procesa na standardni ulaz drugog procesa i tako obrazovati tok procesa(pipe)
- Nadovezivanje u tok se označava pomoću znaka |.
- Za prethodni primer ovakav tok bi izgledao `obrni<recnik.txt | sortiraj | obrni> rime.txt`
- Razmena podataka između dva procesa, koji su povezani u tok, se ostvaruje posredstvom posebne specijalne datoteke.
- Njoj odgovara bafer u radnoj memoriji.
- Ova baferovana specijalna datoteka služi prvom od ovih procesa kao standardni izlaz, a drugom od njih kao standardni ulaz.
- Znači, prvi proces samo piše u ovu datoteku, a drugi samo čita iz nje.
- Prilikom obrazovanja toka procesa, interpreter znakovnog komandnog jezika stvara procese, koji se povezuju u tok.
- Pri tome on koristi istu posebnu specijalnu datoteku kao standardni izlaz i ulaz za svaki od parova ovih procesa.
- Zatim interpreter znakovnog komandnog jezika čeka na kraj aktivnosti poslednjeg od ovih procesa.
- Pozadinski procesi se razlikuju od običnih (interaktivnih) procesa po tome što interpreter znakovnog komandnog jezika, nakon stvaranja pozadinskog procesa, ne čeka kraj njegove aktivnosti, nego nastavlja interakciju sa korisnikom.
- Zato su pozadinski procesi u principu neinteraktivni.
- Na primer, komanda:
- kompiliraj < program.c > program.obj &
- Omogućuje stvaranje pozadinskog procesa, koji obavlja komplikaciju programa, sadržanog u datoteci program.c, i rezultat kompilacije smešta u datoteku program.obj, a eventualne greške u kompilaciji prikazuje na ekranu.

- U prethodnom primeru znak & sa kraja komande je naveo interpreter znakovnog komandnog jezika na stvaranje pozadinskog procesa.
- Svaka komanda, upućena interpreteru znakovnog komandnog jezika, ne dovodi do stvaranja procesa.
- Komande, koje se često koriste, pa je važno da budu brzo obavljene, interpreter znakovnog komandnog jezika obavlja sam.
- Za ostale komande, za koje se stvaraju procesi, interpreter znakovnog komandnog jezika čeka kraj aktivnosti stvorenog procesa da bi od njega dobio, kao povratnu informaciju, završno stanje stvorenog procesa.
- Ovo stanje se obično kodira celim brojem.
- Ako interpreter znakovnog komandnog jezika protumači ovaj broj kao logičku vrednost (0 - tačno, različito od 0 - netačno), tada on može da podrži uslovno izvršavanje programa.
- Ovo znači da bilo koja komanda može da se tretira kao logički uslov i da se koristi u logičkim izlazima (recimo kroz && ili ||) ili kao uslov komande if za uslovno izvršavanje komandi.
- Pored prethodno opisane komande za uslovno izvršavanje programa, interpreteri znakovnih komandnih jezika podržavaju komandu za ponavljanje izvršavanja programa, ali i druge komande, tipične za procedurne programske jezike, koje omogućuju rukovanje promenljivim, konstantama, ulazom, izlazom i slično.
- To dozvoljava pravljenje komandnih datoteka (shell scripts) čija su preteče runcom fajlovi iz grubo govoreći najstarijeg time-share operativnog sistema.
- One opisuju okolnosti pod kojima se izvršavaju korisnički programi, a sadržaj komandne datoteke preuzima na interpretiranje interpreter znakovnog komandnog jezika.
- Zato komandne datoteke imaju poseban tip, da bi ih interpreter znakovnog komandnog jezika mogao prepoznati.
- Zahvaljujući tome, ime svake komandne datoteke, uostalom, kao i ime svake izvršne datoteke, predstavlja ispravnu komandu znakovnog komandnog jezika.
- Iako broj i vrste ovakvih komandi nisu ograničeni, jer zavise samo od kreativnosti i potreba korisnika, ipak je moguće napraviti njihovu klasifikaciju i navesti neke neizbežne grupe komandi.
- Najgrublja podela komandi je na:

- korisničke komande
  - administratorske komande
- Korisničke komande, između ostalog, omogućuju:
    - rukovanje datotekama
    - rukovanje imenicima
    - rukovanje procesima
    - razmenu poruka između korisnika
  - Standardne komande za rukovanje datotekama omogućuju:
    - izmenu imena, atributa datoteke i pomeranje (`mv`)
    - poređenje sadržaja datoteka (`diff`)
    - kopiranje datoteka (`cp`)
    - uništenje datoteka (`rm`)
  - U komande za rukovanje imenicima spadaju:
    - komande za stvaranje i uništenje imenika (`mkdir`)
    - komanda za promenu radnog imenika (`cd`)
    - komanda za pregledanje sadržaja imenika (imena datoteka i imena imenika, sadržanih u njemu) (`ls`)
    - komande za izmenu imena, lokacije i ostalih atributa imenika. (`mv`)
  - Administratorske komande omogućuju:
    - pokretanje i zaustavljanje rada računara,
    - spašavanje (backup) i vraćanje (restore) datoteka rukovanje vremenom
    - rukovanje vremenom
    - sabijanje (compaction) datoteka
    - ažuriranje podataka o korisnicima računara i njihovim pravima
    - generisanje izveštaja o korišćenju računara (o korišćenju procesorskog vremena ili o korišćenju prostora na disku)
  - rukovanje konfiguracijom računara (određivanje načina rada uređaja i programa koji ulaze u njegov sastav)
  - proveru ispravnosti rada računara
  - pripremu diskova za korišćenje (ovo obuhvata pronalaženje oštećenih blokova i njihovo isključivanje iz upotrebe, pronalaženje izgubljenih blokova i njihovo uključivanje u evidenciju slobodnih blokova, formiranje skupa datoteka na disku i njegovo uključivanje u skup datoteka računara).

## **Pitanja**

### **Pitanja**

- Od čega se sastoje komande znakovnog komandnog jezika?
- Kako se zadaju komande grafičkih komandnih jezika?
- Šta su ciljevi znakovnih komandnih jezika?
- Šta omogućuju znakovni komandni jezici?
- Šta omogućuje preusmeravanje?
- Čemu služi pipe?
- Čemu služi baferovana specijalna datoteka?
- Šta karakteriše pozadinske procese?
- Šta karakteriše komandne datoteke?
- Šta omogućuju korisničke komande?
- Šta omogućuju administratorske komande?— title: Procesi author: Veljko Petrović date: 2023-06 —

## **Sloj za Rukovanje Procesima**

### **O prezentaciji**

- Ovo je već predeno kroz druga predavanja, ali ovo služi kao obnova i sistemarizacija već predenog
- Tematika jesu procesi u operativnom sistemu: budući da smo ih pomenuli stotinama puta i da smo ih i pravili od nule, sve bi ovo trebalo da bude poznato

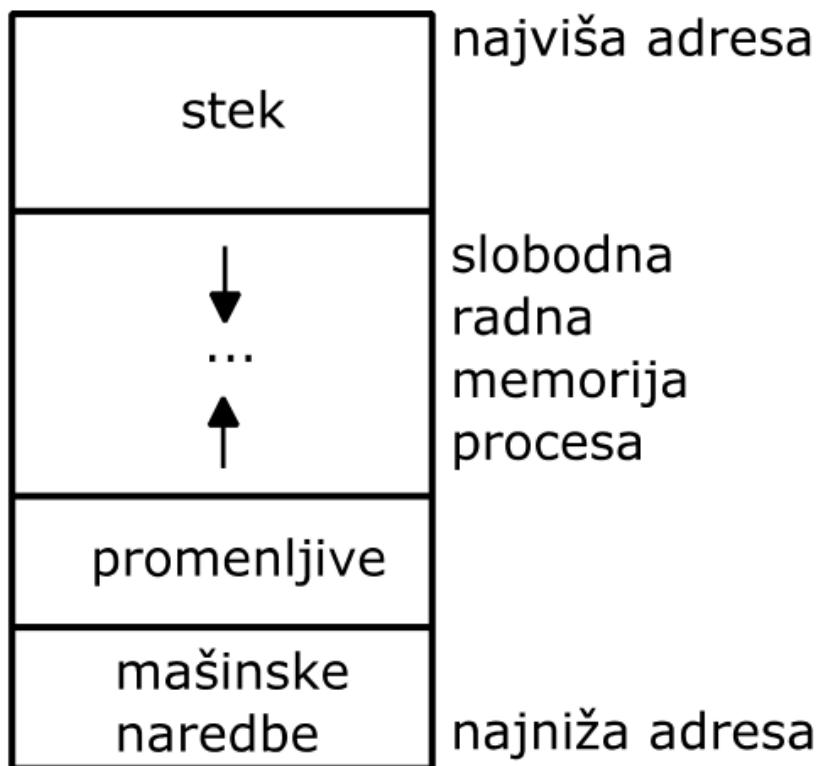
### **Sloj OS za rukovanje procesima**

- Osnovni zadaci sloja za rukovanje procesima su:
  - stvaranje procesa
  - uništenje procesa.
- Stvaranje procesa obuhvata:
  - stvaranje slike procesa
  - stvaranje deskriptora procesa
  - pokretanje aktivnosti procesa
- Uništenje procesa obuhvata:
  - zaustavljanje aktivnosti procesa
  - uništenje slike procesa
  - uništenje deskriptora procesa

## Osnovni zadaci sloja za rukovanje procesima

- Pored sistemskih operacija stvaranja i uništenja procesa, potrebne su i sistemske operacije za izmenu atributa procesa, na primer, za izmenu radnog imenika (direktorijuma) procesa.
- Slika procesa obuhvata niz lokacija radne memorije sa uzastopnim (logičkim) adresama. Ona sadrži:
  - izvršavane mašinske naredbe
  - promenljive
  - stek

## Šema rasporeda u memoriji



## Raspored u memoriji

- Slika procesa počinje od lokacije radne memorije sa najnižom adresom → mašinske naredbe, a završava na lokaciji radne mem-

- orije sa najvišom adresom - stek se puni u smeru nižih adresa
- Iza mašinskih naredbi dolaze statičke promenljive (inicijalizovane, pa neinicijalizovane).
  - Između ovih promenljivih i steka se nalazi slobodna radna memorija procesa.

## Osnovni zadaci sloja za rukovanje procesima

- Radna memorija procesa je na raspolaganju procesu za:
  - širenje (punjenje) steka
  - za stvaranje dinamičkih promenljivih (heap)
- Svi dinamički zahtevi za zauzimanjem radne memorije, postavljeni u toku aktivnosti procesa, se zadovoljavaju samo na račun slobodne radne memorije procesa.
- Ovakva organizacija slike procesa uslovljava da proces prvo ugrozi svoju aktivnost, kada njegovi zahtevi za radnom memorijom nadmaše njegovu raspoloživu slobodnu radnu memoriju (preklapanja steka i promenljivih dovodi do fatalnog ishoda po aktivnost procesa)
- Pored slike, za aktivnost procesa je važan i deskriptor procesa, koji sadrži atribute procesa.
- Ovi atributi karakterišu aktivnost procesa. Oni obuhvataju:
  - stanje procesa ("spreman", "aktivran", "čeka")
  - sadržaje procesorskih registara (zatećene u njima pre poslednjeg preključivanja procesora sa procesa)
  - numeričku oznaku vlasnika procesa
  - oznaku procesa stvaraoca
  - trenutak pokretanja aktivnosti procesa
- ukupno trajanje aktivnosti procesa (odnosno, ukupno vreme angažovanja procesora)
- podatke o slici procesa (njenoj veličini i njenom položaju u radnoj i masovnoj memoriji)
- podatke o datotekama koje proces koristi
- podatak o radnom imeniku procesa
- razne podatke neophodne za upravljanje aktivnošću procesa (poput prioriteta procesa ili položaja sistemskog steka procesa, koga koristi operativni sistem u toku obavljanja sistemskih operacija).

## Sistemske operacije za stvaranje i uništenje procesa

- Za stvaranje procesa potrebno je pristupiti odgovarajućoj izvršnoj datoteci sa inicijalnom slikom procesa. Ona sadrži:
  - mašinske naredbe
  - početne vrednosti statičkih promenljivih
  - podatak o veličini delova slike procesa.
- Takođe, potrebno je zauzeti deskriptor procesa, kao i dovoljno veliku zonu radne memorije za sliku procesa.
- Sve ovo spada u nadležnost sistemske operacije stvaranja procesa (`fork()` i `exec()`).
- Ovu operaciju poziva proces stvaralac i ona se obavlja u toku njegove aktivnosti.
- U okviru poziva sistemske operacije stvaranja procesa kao argument se navodi putanja odgovarajuće izvršne datoteke.
- Svi atributi stvaranog procesa ne moraju biti navedeni u okviru poziva sistemske operacije stvaranja procesa, jer se jedan njihov deo nasleđuje iz deskriptora procesa stvaraoca:
  - numerička oznaka vlasnika procesa
  - podatak o radnom imeniku procesa
  - njegov prioritet
- Za uništenje procesa potrebno je oslobođiti njegov deskriptor i zonu radne memorije sa njegovom slikom što spada u nadležnost sistemske operacije uništenja procesa (`exit()`).
- Nju automatski poziva proces na kraju svoje aktivnosti, čime izaziva svoje samouništenje.
- Uništenje procesa se završava preključivanjem procesora sa uništavanog na neki od spremnih procesa.
- U okviru operacije uništenja procesa uputno je predvideti argument kojim proces saopštava svoje završno stanje, odnosno informaciju da li je aktivnost uništavanog procesa bila uspešna ili ne.
- Da bi proces stvaralac mogao iskoristiti ovakvu povratnu informaciju od stvorenog procesa, on mora pozivom posebne sistemske operacije (`wait()`) da zatraži zaustavljanje svoje aktivnosti i tako omogući preključivanje procesora na stvarani proces.

## Zamena slika procesa

- Swapovanje procesa je procedura gde se (delovi) nekog procesa koji je pokrenut izbacuju iz fizičke memorije zato što se ne smatra da će biti pomenuti: dosta smo o ovome pričali kada smo pričali o *paging* tehnici i virtuelnoj memoriji.
- U nadležnosti ove operacije zamene (swap) je dugoročno raspoređivanje (long term scheduling), u okviru koga se odabiraju stranice koje se izbacuju i potencijalno (prefetch) ubacuju.
- Važno je uočiti da se dugoročno raspoređivanje razlikuje od običnog ili kratkoročnog raspoređivanja (short term scheduling), koje među spremnim procesima odabira proces na koga se preključuje procesor.

## Rukovanje nitima

- Rukovanje nitima može, ali i ne mora, biti u nadležnosti sloja za rukovanje procesima.
- Kada je rukovanje nitima povereno sloju za rukovanje procesima, tada operativni sistem nudi sistemske operacije za rukovanje nitima, koje omogućuju stvaranje, uništavanje i sinhronizaciju niti.
- U ovom slučaju, deskriptori i sistemski stek niti se nalaze u sistemskom prostoru, dok se sopstveni stek niti nalazi u korisničkom prostoru (unutar slike procesa).
- U slučaju kada rukovanje nitima nije u nadležnosti operativnog sistema, brigu o nitima potpuno preuzima konkurentna biblioteka.
- Pošto ona pripada slici procesa, rukovanje nitima u ovom slučaju se potpuno odvija u korisničkom prostoru, u kome se nalaze i deskriptori niti, kao i stekovi niti.
- Osnovna prednost rukovanja nitima van operativnog sistema (ovo se još zovu i "zelene niti") je efikasnost, jer su pozivi potprograma konkurentne biblioteke brži (kraći) od poziva sistemskih operacija.
- Ove niti se ne *izvršavaju* brže, štaviše tipično se izvršavaju *sporije*
- Ono što ih čini bržim jeste to što njihovo stvaranje, uništavanje, itd. zahteva manje vremena i memorije
- Generalno pravilo jeste da se sistemske niti koriste kada želite paralelizam, a ne-sistemske preferiraju kada je u pitanju asinhron proces koji obrađuje ulaz i izlaz.

- Ali, kada operativni sistem ne rukuje nitima, tada poziv blokirajuće sistemske operacije iz jedne niti dovodi do zaustavljanja aktivnosti procesa kome ta nit pripada, jer operativni sistem pripisuje sve pozive sistemskih operacija samo procesima, pošto ne registruje postojanje niti.
- Na taj način se sprečava konkurentnost unutar procesa, jer zaustavljanje aktivnosti procesa sprečava aktivnost njegovih spremnih niti.
- Zato savremeni operativni sistemi podržavaju rukovanje nitima. Tako, u okviru POSIX (Portable Operating System Interface) standarda (IEEE 1003 ili ISO/IEC 9945) postoji deo pthread (POSIX threads) koji je posvećen nitima.
- Ako ipak želimo ne-sistemsко rukovanje onda okruženje koje se koristi mora vrlo pažljivo koristiti (ispod haube) isključivo ne-blokirajuće operacije i uspostavljati preključivanje među nitima koje je efektno.

## Oslove komunikacije između procesa

- Komunikacija između procesa može biti, kroz POSIX:
- Signal
- Baferi (Pipe, FIFO, Message Queue)
- Deljena memorija
- Semafori

## Signalni

- Signali su softverski prekidi koji označavaju da se u sistemu desio nekakav događaj.
- Ponekad ih generišu drugi procesi, ponekad sam operativni sistem, a ponekad korisnik direktno.
- Signali imaju različito značenje
- Može se dobiti kompletan lista iz operativnog sistema

```
veljko@HPC:~$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS     34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

## Reakcija na signale

- Svaki tip signala ima jednu podrazumevanu akciju iz sledećeg skupa:
  - **Term**
    - \* Proces koji dobija signal se terminira.
  - **Ign**
    - \* Proces koji dobija signal ga ignoriše.
  - **Core**
    - \* Podrazumevana akcija jeste da se terminira proces i da se u fajl izbací sva memorija procesa.
  - **Stop**
    - \* Proces se pauzira.
  - **Cont**
    - \* Proces se nastavi ako je pauziran.

## Namena signala

Ime signala	Broj signala	Reakcija	Svrha
SIGTERM	15	Term	Signal za terminaciju
SIGUSR1	10	Term	Rezervisan za korisnika.
SIGUSR2	12	Term	Rezervisan za korisnika.
SIGCHLD	17	Ign	Dete-proces je prekinut.
SIGCONT	18	Cont	Nastavi izvršavanje.
SIGSTOP	19	Stop	Pauziraj izvršavanje.
SIGSTP	20	Stop	Pauziraj izvršavanje (pokrenut sa terminala)

Ime signala	Broj signala	Reakcija	Svrha
SIGBUS	7	Core	Greška magistrale
SIGPOLL	29	Term	Sinonim za SIGIO
SIGPROF	27	Term	Tajmer za profiliranje je istekao
SIGSYS	31	Core	Greška u sistemskom pozivu.
SIGTRAP	5	Core	Breakpoint dostignut.
SIGURG	23	Ign	Hitna reakcija na socket-u.
ISGVTALRM	26	Term	Virtualni alarm

Ime signala	Broj signala	Reakcija	Svrha
SIGIOT	6	Core	Isto što i SIGABRT.
SIGIO	29	Term	I/O sada moguć.
SIGPWR	30	Term	Greška sa napajanjem.

Ime signala	Broj signala	Reakcija	Svrha
SIGTTIN	21	Stop	Komunikacija sa terminalom za poz. proces
SIGTTOU	22	Stop	Komunikacija sa terminalom za poz. proces
SIGXCPU	24	Core	Potrošeno svo CPU vreme.
SIGXFSZ	25	Core	Potrošeno ograničenje veličine fajla.

## Slanje signala

```
kill -<signal> <pid>
```

```
kill -SIGUSR1 10366
```

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main() {
    for(int i = 0; i < 999999999; i++ ){
        if(i == 4817) kill(getpid(), SIGTERM);
    }
    return 0;
}
```

## Reagovanje na signal

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void signalCallback(int sig){
    printf("Blokiram TERM: %d\n", sig);
}

int main() {
    signal(SIGTERM, signalCallback);
    for(int i = 0; i < 999999999; i++ ){
        if(i == 4817) kill(getpid(), SIGTERM);
        if(i == 9044) kill(getpid(), SIGKILL);
    }
    return 0;
}
```

## Baferi

- Bafer je opšte ime za nekoliko metoda za komunikaciju.
- Ovde ćemo pogledati malo bolje 'pipe' i 'fifo' mehanizam koji funkcioniše kao fajl koji, kad se napravi, ima dva deskriptora: iz jednog se čita, a u drugi piše.
- Više procesa može da deli ove, sa tim da ako jedan piše, drugi ne može. To se rešava tako što svaki proces zatvoriti ono što nije neophodno.
- Pipe živi koliko i jedan proces, FIFO je permanentan i ima ime.

## Pipe—Include

```
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <cstring>

using namespace std
```

## Pipe Main

```
int main(){
    int pipeD[2];
    int pipeR[2];
    pid_t pid;
    char inbuf[256];
    char outbuf[256];
    int status = pipe(pipeD);
    if(status == -1){
        cout << "Baffer nije napravljen." << endl;
        return 1;
    }
    status = pipe(pipeR);
    if(status == -1){
        cout << "Baffer nije napravljen." << endl;
        return 1;
    }
    pid = fork();
    if(pid == 0){
        //deti proces
    }else{
        //roditelj proces
    }
}
```

## Pipe—Dete proces

```
if(pid == 0){
    close(pipeR[1]);
    close(pipeD[0]);
    read(pipeR[0], inbuff, 256);
    cout << "[DETE] Dobio sam: " << inbuff << endl;
    strcpy(outbuff, "Test 123");
    cout << "[DETE] Saljem: " << outbuff << endl;
    write(pipeD[1], outbuff, strlen(outbuff) + 1);
    cout << "[DETE] Poslato!" << endl;
}
```

## Pipe—Roditelj proces

```
else{
    close(pipeR[0]);
    close(pipeD[1]);
    strcpy(outbuff, "Test 321");
    cout << "[RODITELJ] Saljem: " << outbuff << endl;
    write(pipeR[1], outbuff, strlen(outbuff) + 1);
    cout << "[RODITELJ] Poslato!" << endl;
    read(pipeD[0], inbuff, 256);
    cout << "[RODITELJ] Dobio sam: " << inbuff << endl;
}
```

## FIFO

- FIFO je apsolutno identičan Pipe mehanizmu sa tom razlikom da ima ime i postojanje koje je nezavisno u odnosu na proces koji ga stvara.
- FIFO uvek postoji negde na disku, tj. ima putanju. Mora se napraviti tako što se otkuca mknod IME p

## Deljena memorija

- Deljenu memoriju više proučavamo kroz kod, ali ideja je jednostačna: definišemo region memorije koji dele dva procesa, i komuniciramo kroz njega.
- Jako je brzo, ali su problemi sinhronizacije poprilični.

# **Sistemski Procesi**

## **Sistemski procesi - nulti proces**

- Nulti proces je kao baš i nulta nit: služi da bi uvek bilo nešto što može da se izvrši, večno ili aktivno ili spremno.
- Njegov prioritet je niži od prioriteta svih ostalih procesa, a on postoji za sve vreme aktivnosti operativnog sistema.

## **Sistemski procesi - dugoročni rasporedivač**

- Drugi primer sistemskog procesa je proces dugoročni rasporedivač, koji se brine o tome koje su stranice u radnoj memoriji
- On se periodično aktivira
- Da bi se proces uspavao, odnosno da bi se njegova aktivnost zaustavila do nastupanja zadanog trenutka, on poziva odgovarajuću sistemsku operaciju.
- Ona pripada sloju za rukovanje kontrolerima, jer proticanje vremena registruje dajver sata.
- I proces dugoročni rasporedivač postoji za sve vreme aktivnosti operativnog sistema (jasno, ako ima potrebe za dugoročnim raspoređivanjem).

## **Sistemski procesi - procesi identifikator i komunikator**

- U sistemske proceze spada i proces identifikator (login process), koji podržava predstavljanje korisnika.
- Proces identifikator koristi terminal, da bi posredstvom njega stupio u interakciju sa korisnikom u toku predstavljanja, radi preuzimanja imena i lozinke korisnika.
- Po preuzimanju imena i lozinke, proces identifikator proverava njihovu ispravnost i, ako je prepoznao korisnika, tada stvara proces komunikator, koji nastavlja interakciju sa korisnikom.

## **Identifikator i komunikator**

- Moderni računari imaju proces identifikator koji je grafički i radi dodatne poslove, rečimo pokreće grafički shell, ali je princip isti: mora se ustanoviti identitet korisnika.
- Proces komunikator je, naravno, shell: nešto što zahteve korisnika pretvoriti u operacije operativnog sistema.

## Sistemske procese - procesi identifikator i komunikator

- Za proveru ispravnosti imena i lozinke korisnika, neophodno je raspolažati spiskovima imena i lozinki registrovanih korisnika.
- Ovi spiskovi se čuvaju u posebnoj datoteci lozinki (password file).
- Svaki slog ove datoteke sadrži:
  - ime i lozinku korisnika
  - numeričku oznaku korisnika
  - putanju radnog imenika korisnika
  - putanju izvršne datoteke, sa inicijalnom slikom korisničkog procesa komunikatora
  - Nekada se lozinke korisnika čuvaju u posebnoj datoteci (shadow file).

### /etc/passwd

```
[veljko@Episteme ~]$ cat /etc/passwd
root:x:0:0::/root:/bin/bash
nobody:x:65534:65534:Nobody:/sbin/nologin
dbus:x:81:81:System Message Bus:/sbin/nologin
bin:x:1:1::/sbin/nologin
daemon:x:2:2::/sbin/nologin
mail:x:8:12::/var/spool/mail:/sbin/nologin
ftp:x:14:11::/srv/ftp:/sbin/nologin
http:x:33:33::/srv/http:/sbin/nologin
systemd-journal-remote:x:982:982:systemd Journal Remote:/sbin/nologin
systemd-coredump:x:981:981:systemd Core Dumper:/sbin/nologin
uidadd:x:68:68::/sbin/nologin
dnsmasq:x:980:980:dnsmasq daemon:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/sbin/nologin
avahi:x:979:979:Avahi mDNS/DNS-SD daemon:/sbin/nologin
colord:x:978:978:Color management daemon:/var/lib/colord:/sbin/nologin
cups:x:209:209:cups helper user:/sbin/nologin
deluge:x:977:977:Deluge BitTorrent daemon:/srv/deluge:/sbin/nologin
git:x:976:976:git daemon user:/usr/bin/git-shell
lightdm:x:620:620:Light Display Manager:/var/lib/lightdm:/sbin/nologin
nm-openconnect:x:975:975:NetworkManager OpenConnect:/sbin/nologin
nm-openvpn:x:974:974:NetworkManager OpenVPN:/sbin/nologin
ntp:x:87:87:Network Time Protocol:/var/lib/ntp:/bin/false
polkitd:x:102:102:PolicyKit daemon:/sbin/nologin
usbmux:x:140:140:usbmux user:/sbin/nologin
veljko:x:1000:1000:Veljko Petrovic:/home/veljko:/bin/bash
systemd-network:x:973:973:systemd Network Management:/sbin/nologin
systemd-resolve:x:972:972:systemd Resolver:/sbin/nologin
systemd-timesync:x:971:971:systemd Time Synchronization:/sbin/nologin
geoclue:x:970:970:Geoinformation service:/var/lib/geoclue:/sbin/nologin
```

```
veljko:x:1000:1000:Veljko Petrovic:/home/veljko:/bin/bash
```

## **Sistemske procese - procesi identifikator i komunikator**

- Za zaštitu datoteka ključno je onemogućiti neovlaštene pristupe datoteci lozinki.
- Prirodno je da njen vlasnik bude administrator i da jedino sebi dodeli pravo čitanja i pisanja ove datoteke.
- Pošto su procesi identifikatori sistemske procese, koji nastaju pri pokretanju operativnog sistema, nema prepreke da njihov vlasnik bude administrator.
- Iako na taj način procesi identifikatori dobijaju i pravo čitanja i pravo pisanja datoteke lozinki, to pravo korisnici ne mogu da zloupotrebe, jer, posredstvom procesa identifikatora, jedino mogu proveriti da li su registrovani u datoteci lozinki.
- Pri tome je bitno da svoja prava proces identifikator ne prenosi na proces komunikator. Zato proces komunikator ne nasleđuje numeričku oznaku vlasnika od svog stvaraoca procesa identifikatora.
- Administrator bez problema pristupa datoteci lozinki, radi izmene njenog sadržaja, jer je on vlasnik svih procesa, koje je stvorio i pokrenuo da bi izvršili njegove komande.
- Pošto korisnici nemaju načina da pristupe datoteci lozinki, javlja se problem kako omogućiti korisniku da sam izmeni svoju lozinku.
- Taj problem se može rešiti po uzoru na proces identifikator, koga koriste svi korisnici, iako nisu njegovi vlasnici.
- Prema tome, ako je administrator vlasnik izvršne datoteke sa inicijalnom slikom procesa za izmenu lozinki, dovoljno je naznačiti da on treba da bude vlasnik i procesa nastalog na osnovu ove izvršne datoteke (SUID - Switch User IDentification program).
- Datoteka lozinki je dodatno zaštićena, ako su lozinke u nju upisane u izmenjenom, odnosno u kriptovanom (encrypted) obliku, jer tada administrator može da posmatra sadržaj datoteke lozinki na ekranu, ili da ga štampa, bez straha da to može biti zloupotrebljeno.
- Da bi se sprečilo pogađanje tuđih lozinki, proces identifikator treba da reaguje na više uzastopnih neuspešnih pokušaja predstavljanja, obaveštavajući o tome administratora, ili odbijajući neko vreme da prihvati nove pokušaje predstavljanja.

- Takođe, korisnici moraju biti oprezni da sami ne odaju svoju lozinku lažnom procesu identifikatoru. To se može desiti, ako se njihov prethodnik ne odjavi, nego ostavi svoj proces da opslužuje terminal, oponašajući proces identifikator.

## Kriptografija

Umetnost primenjene paranoje

### Istorijska kriptografija

- Od starogrčkog: κρυπτός (skrivena) i λόγος (reč)
- Sa tim je povezan i termin ‘šifra’ odn. engleski ‘cypher’ ili ‘cipher’
- Cipher/šifra dolaze od arapskog ‘al sifr’ što znači nula.
- Zašto? Zato što su srednjovekovni Evropljani imali sujeveran strah od nule kao koncepta.
- Najranije forme su stvari kao što je Cezarova šifra čiji je moderni ekvivalent rot13.
- Šta je problem? Ako znam algoritam, znam da provalim šta je šifra. Jedini parametar cezarove šifre je za koliko mesta pomjeramo abecedu, budući da je to 26 mogućih vrednosti odn. ni 5 bita bezbednosti za brute-force, to i nije nekakva zaštita.
- Možemo to i bolje, uzmite Polibijev Kvadrat.
- To je već 25! mogućih varijacija, odn.  $15511210043330985984000000$  mogućih ključeva. To je skoro 84 bita bezbednosti. Mnogo bolje.
- Uprkos tome, mogli bi ste da potpuno razbijete bilo koji Polibijev Kvadrat bez puno napora.
- Zašto? Informacije cure kroz distribuciju slova, u engleskom, na primer, najčešćih prvih 12 slova su, redom, ETAOIN SHRDLU. Ovo je osobina koju Polibijev kvadrat uopšte ne krije.

### Šenonovi kriterijumi i šta čini dobar metod šifrovanja

- Konfuzija
  - Svaki bit šifrovanog izlaza mora zavisiti od više bitova ključa
- Difuzija

- Izmena od jednog bita u latnog, nešifrovanog teksta, bi trebalo da ima matematičko očekivanje da će se izmeniti pola bitova šifrovanog izlaza.

### Istoriski, ovo nam i nije najbolje išlo

- Prvi pokušaj da se to ispravi jesu polialfabetske šifre: prvo slovo menjamo po prvoj azbuci zamene, drugo po drugoj itd. Nije baš radilo zbog **napada preko dekompozicije**.
- Drugi pokušaj? Kontinualno kližuća azbuka preko rotora.
- Ovo može da bude prilično bezbedno, čak i danas. RC4 je rotorska šifra. Čak postoje moderno-bezbedni algoritmi koji se mogu raditi rukom kao što je Šajnerov 'pasijans' algoritam.
- No, lako je napraviti grešku — primer Enigme.

### Da li postoji savršena šifra?

- Da! Od 1882, štaviše!
- U pitanju je tzv 'jednostruka zamena' odn. 'one-time pad'
- Problem?
- Trebaju nam stvarno slučajni brojevi
- Treba nam razmena ključa velikog koliko i poruka sa savršenom bezbednošću. To je... problematično, jer ako već imao sigurni kanal komunikacije, što šifrovati?

### O slučajnim brojevima

- Generacija slučajnih brojeva je apsolutno ključna za efektну kriptografiju, ne samo za jednostrukе zamene.
  - Da stvar bude zanimljivija, to je funkcionalnost koju baš očekujemo od operativnog sistema.
  - Ipak, to su samo slučajni brojevi... koliko teško to može biti?
- 

#### Veoma

**Anyone who considered arithmetical methods of producing random digits is, of course, in a state of sin.** —John von Neumann, General-Purpose Genius

**Random number generation is too important to be left to chance.** —Robert Coveyou, Oak Ridge National Laboratory

## **Pseudo-slučajni brojevi**

- Računari su determinističke mašine. Kao takve ne mogu da stvaraju slučajne brojeve.
- Sa druge strane mogu da stvaraju pseudo slučajne brojeve, odn. niz nepredvidivih vrednosti koje su takve da ako znaš proizvoljno mnogo slučajnih brojeva i dalje ne možeš da predviđiš sledeću.
- Da bi ovo radio potreban je odličan algoritam koji se mora na početku nahraniti sa malo istinske slučajnosti: izvorom entropije. Ovo je ograničen, dragocen resurs.

## **Linux**

- /dev/random je kako Linux pokušava da nam ovde pomogne: to je izvor kvalitetne entropije koji se formira na osnovu šuma sa sistemске magistrale.
- Ovo je odličan izvor ali se lako isprazni, a ako se to desi pokušaj da se pozove 'read' na tome će blokirati dok ne dobijemo svežu entropiju što zahteva jako puno vremena.
- Ovo je napadačka površina za DDOS iscrpljivanjem entropije.
- /dev/urandom nikada ne blokira, ali su brojevi generisani algoritmom.
- Srećom, to je barem jako kvalitetan algoritam, barem od verzije 4.8.

## **Hardverski generatori slučajnih brojeva**

- Za jako bitne primene postoje hardverski generatori bazirani na, npr, atmosferskom šumu ili, najčešće, kvantno-mehanički procesi kao što je raspad nuklearnih izotopa. Ovo je skupo i nezgodno, budući da niko ne želi radio-izotop na svojoj matičnoj ploči.
- Intel nudi rešenje: svaki Ivy Bridge i kasniji procesor ima na čipu RDRAND instrukciju koja proizvodi hardverski-generisane (rezistivan šum) slučajne brojeve.

## **Intel nudi problem**

- Prvi problem: Bezbednosno ključni mehanizam je sada (bukvalno) crna kutija iz koje izlaze bitovi, bez načina da se uverimo da se prave na pravi način.

- Gore, mnogo gore od toga, jeste da je RDRAND implementacija... podešena od strane NSA.
- NSA je permanentan protivnik kvalitetne kriptografije. Već se zna da su ključne NIST konstante manipulisane.

## Šta hoćemo od moderne kriptografije?

- Poverljivost
- Integritet
- Identitet

## Kripto-sistem

- Kripto-sistem je mehanizam koji postiže jedan od prethodnih ciljeva, najčešće sva tri, i sastoji se od protokola slanja, primanja, verifikacije poruka između nekog broja učesnika u nekom redosledu.
- Kripto-sistemi se grade od kripto-primitiva, bazičnih matematičkih konstrukta koji nam omogućavaju da operišemo.

## Kripto-primitivi

- Jednosmerne, heš, funkcije.
- Simetrična, tajni-ključ, enkripcija.
- Asimetrična, javni-ključ, enkripcija.

## Heš funkcije

- Kriptografske
  - HMAC
  - Lozinka
  - KDF
- Opšte
  - Verifikacija
  - Adresiranje

## Kriptografska heš funkcija

- Neka funkcija  $H$  je kriptografski heš ako je komputaciono izuzetno skupo da:
- Za neku vrednost  $h$  naći ulaz  $i$  takav da  $H(i) = h$
- Za neki ulaz  $i$  naći drugi ulaz  $j$  takav da  $H(i) = H(j)$
- Naći bilo koja dva ulaza  $i$  i  $j$  takva da  $H(i) = H(j)$

## Moderne kriptografske heš funkcije

- SHA-2 i SHA-3 i koncept kripto-agilnosti.
- Šta je problem sa SHA-2, današnjom najčešće korišćenom heš funkcijom?
  - Kletva NSA
  - Izuzetna efikasnost implementacije
  - Delimični napadi
  - Napad povećanjem dužine

## Napad sa povećanjem dužine

Funkcije porodice SHA - 2, tj.svih koje koriste tkzv.Merkle-Damgard konstrukciju kao svoju osnovu imaju slabost gde ako znamo  $H(M_1)$  i dužinu  $M_1$  onda možemo da izračunamo  $H(M_1 \parallel M_2)$  za proizvoljni  $M_2$  gde je  $\parallel$  operator konkatenacije.

## Alternativne Heš Funkcije i Heš Funkcije Posebne Namene

- Blake2b (Opšta kriptografska)
- SipHash (Štiti od DDOS potencijala malicioznih kolizija u okviru MurmurHash3)
- Argon2 (Šifre i KDF)

## Simetrični algoritmi

- Imamo deljenu tajnu između dve strane koje tajno komuniciraju.
- Bezbednost zavisi 100% of toga da je ta tajna deljena a za druge tajna. Ta tajna se obično zove 'tajni ključ' ili samo 'ključ'
- Simetrični algoritmi mogu raditi na blokovima ili na tokovima.
- Algoritmi na tokovima rade kao jednostruka zamena, samo što postoji algoritam koji generiše 'blokče' koje koristimo. Jako moćan sistem, ali jako osetljiv na periodičnost i početna podešavanja.

## ChaCha

- Tačnije, ChaCha20: algoritam toka
- Koristi je interno Google, OpenSSH polako prelazi na ChaCha, proizvodi CSPRNG za Linux od 4.8 (simetrični algoritmi enkripcije su takođe dobri generatori PRN-ova)
- Jedini ulaz su podaci i nonce.

## **Nonce?**

- Number used ONCE.
- Veliki broj algoritama za enkripciju zahteva neki početni broj koji se apsolutno ne sme ponovo koristiti.
- Veliki broj naizgled kompetentnih programera ponovo upotrebi Nonce ili IV i dobije se debakl kao što je DRM za PS3.

## **Blok-šifre**

- Rade na blokovima fiksne dužine.
- Trik kod njih su lukavo složene operacije mešanja i zamene nad ulaznim podacima na način koji tajni ključ parametrizuje.
- Ideal je da se postigne efekat lavine—svaki bit izlaza zavisi od svakog bita ulaza.

## **AES**

- Izuzetno dobar standard za simetričnu enkripciju, naročito AES-256.
- Postoje kvalitetne hardverske implementacije.
- Ali, AES takođe ima mračnu stranu: strašno je lako upucati se u nogu kada se koristi. Zašto? Zato što simetrične šifre enkriptuju jedan jedini blok i to je to.
- Ako hoćemo više (hoćemo) treba nam operativni mod AES-a, a tu problemi nastaju.

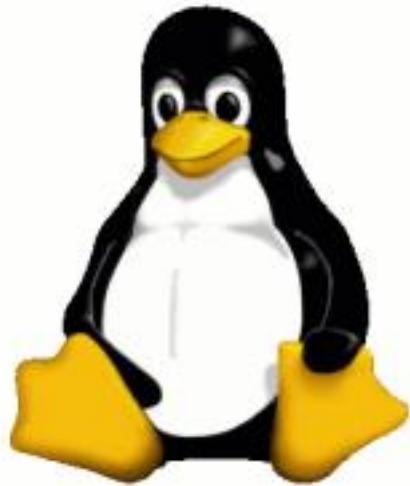
## **Operativni Modovi i IV**

- IV je kao nonce ali sa sledećim osobinama:
- Može biti javan
- Ne sme da se ponavlja
- Ne sme da bude predvidiv

## **Zamka ECB**

- Trivijalan način da se AES primeni na više podataka jeste da se podaci podeli na blokove i ista operacija izvrši na svakom bloku.
- Ovo se zove Electronic Code Book
- To vam je čak i ponuđeno u nekim implementacijama.
- Ovo je kataklizmično loša ideja.

**ECB primjenjen na slici - Original**



**ECB primjenjen na slici - Enkriptovana slika**



## **Brojački režimi**

- Bezbedan režim za upotrebu AES-a ovih dana jeste da se, efektivno, pretvori u sistem za enkripciju preko toka, tako što se tok generiše kroz AES nad nekakvim brojačem koji se dodaje (ne aritmetički) na IV.
- Današnji state-of-the-art je AES-GCM

## **Asimetrična kriptografija**

- Simetrična kriptografija nije podesna za kriptovanje poruka, jer tada ključ kriptovanja mora znati svaki pošiljalac poruke, što ga dovodi u poziciju da može da dekriptuje poruke drugih pošiljalaca.
- To nije moguće u asimetričnoj kriptografiji (public-key cryptography), jer je njena osobina da se iz ključa kriptovanja ne može odrediti ključ dekriptovanja, pa poznavanje ključa kriptovanja ne omogućuje dekriptovanje.
- Ovakav ključ kriptovanja se zove javni ključ (public key), jer je on dostupan svima.
- Njemu odgovarajući ključ dekriptovanja je privatni (tajan), jer je dostupan samo osobama ovlašćenim za dekriptovanje. Zato se on naziva privatni ključ (private key).
- Prema tome, svaki pošiljalac poruke raspolaže javnim ključem, da bi mogao da kriptuje poruke, dok privatni ključ poseduje samo primalac poruka, da bi jedini mogao da dekriptuje poruke.
- Asimetrična kriptografija se temelji na korišćenju jednostavnih algoritama kriptovanja kojima odgovaraju komplikovani algoritmi dekriptovanja.
- Zato je asimetrična kriptografija mnogo sporija od simetrične.
- Obično se asimetrična kriptografija koristi samo za razmenu ključeva potrebnih za simetričnu kriptografiju, pomoću koje se, zatim, kriptuju i dekriptuju poruke.

## **Pitanja**

### **Pitanja**

- Čime se bavi sigurnost?
- Šta omogućuju sistemske opercije za rukovanje procesima?

- Šta obuhvata stvaranje procesa?
- Šta obuhvata uništenje procesa?
- Šta sadrži slika procesa?
- Za šta se koristi slobodna radna memorije procesa?
- Koji atributi procesa postoje?
- Koje sistemske operacije za rukovanje procesima postoje?
- Koji se atributi nasleđuju od procesa stvaraoca prilikom stvaranja procesa?
- Koji se atributi procesa nastanu prilikom njegovog stvaranja?
- Šta karakteriše kopiju slike procesa?
- Koje raspoređivanje je vezano za zamenu slika/stranica procesa?
- Šta karakteriše rukovanje nitima unutar operativnog sistema?
- Šta karakteriše rukovanje nitima van operativnog sistema?
- Šta karakteriše nulti proces?
- Šta je karakteristično za proces dugoročni rasporedivač?
- Šta radi proces identifikator?
- Ko stvara proces komunikator?
- Šta označava SUID (switch user identification)?
- Šta karakteriše heš (hash/jednosmernu) funkciju?
- Šta karakteriše simetričnu kriptografiju?
- Šta karakteriše asimetričnu kriptografiju?
- Na čemu se temelji tajnost kriptovanja?