

Deo I: Uvod u operative sisteme

- ❖ Pojam i funkcije operativnog sistema
- ❖ Osnovni pojmovi
- ❖ Vrste računarskih i operativnih sistema

Mart 2020.

Copyright 2020 by Dragan Milićev

2

dr Dragan Milićev

redovni profesor

Elektrotehnički fakultet u Beogradu

dmilicev@etf.rs, www.rcub.bg.ac.rs/~dmilicev

Operativni sistemi

Osnovni kurs

Slajdovi za predavanja

Deo I - Uvod u operative
sisteme

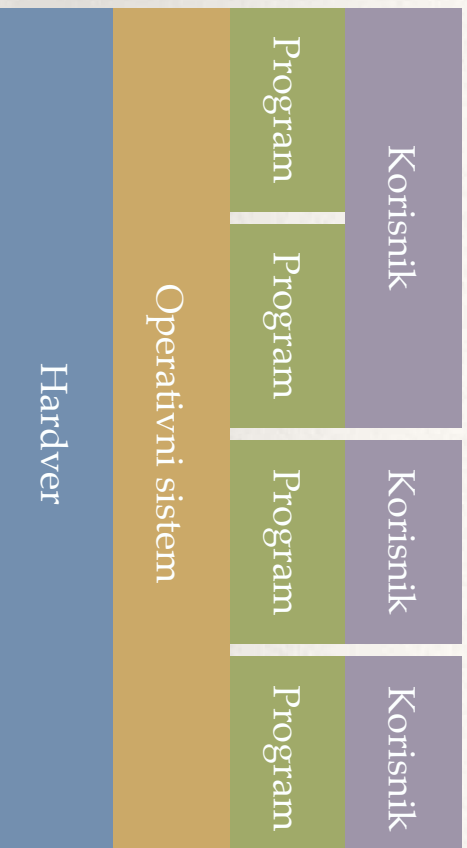
Za izbor slajda drži miša
uz levu ivicu ekrana

Mart 2020.

Copyright 2020 by Dragan Milićev

1

Šta je operativni sistem



Operativni sistem je program (softver) koji omogućava izvršavanje korisničkih programa na računaru i služi kao posrednik između tih programa i računarskog hardvera, pružajući usluge tim programima

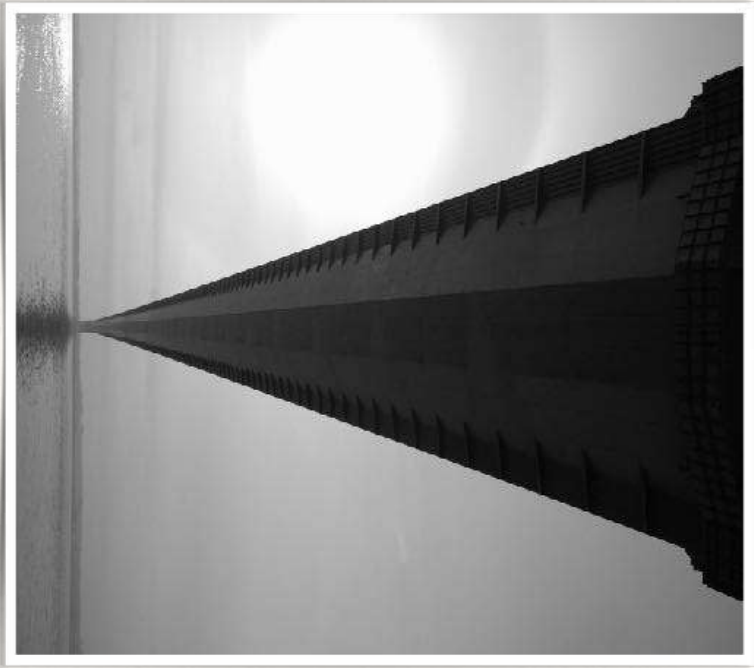
Mart 2020.

Copyright 2020 by Dragan Milićev

4

Glava 1: Pojam i funkcije operativnog sistema

- ❖ Šta je operativni sistem
- ❖ Funkcije operativnog sistema
- ❖ Delovi operativnog sistema
- ❖ Korisnički interfejs



Mart 2020.

Copyright 2020 by Dragan Milićev

3

Šta je operativni sistem

- ❖ Ove funkcije moraju da pristupaju hardverskim ulazno-izlaznim uređajima (tastatura, ekran) kako bi obavile svoje operacije; za to je neophodno poznavati način na koji funkcioniše taj hardver i na koji program treba da interaguje sa njim kako bi obavio operaciju; na primer:
 - ❖ da bi se učitao znak sa tastature, treba sačekati signal od tastature da je taster pritisnut i potom pročitati kod pristignutog tastera iz određenog registra kom se pristupa na određeni način (npr. operacijom čitanja sa neke adrese)
 - ❖ da bi se ispisao znak na ekran, potrebno je znati kakav je tip ekrana (znakovni ili rasterski) i kako mu se prenose podaci: da li znak po znak, pa ih on ispisuje sam ("konzola", znakovni monitor kakvi su nekada bili u upotrebi), ili se definišu pikseli upisom bajtova koji definišu sadržaj piksela (rasterski monitori), potom izračunati poziciju piksela, poziciju reči u memoriji u koju treba upisati vrednost, samu tu vrednost itd.
- ❖ Sve ovo je zametno i veoma komplikovano praviti, a jako zavisi od tipa uređaja

Mart 2020.

Copyright 2020 by Dragan Milićev

6

Šta je operativni sistem

- ❖ Zašto je uopšte potreban OS? Pretpostavimo da ga nema
- ❖ Ko će uopšte učitati neki program i pokrenuti njegovo izvršavanje? Svakako neki program, makar onaj koji uvek postoji u memoriji računara. To već *jeste OS*
- ❖ Pretpostavimo da, osim samo malog softvera koji učitava program u računar i pokreće njegovo izvršavanje, u računaru nema drugog softvera. Pretpostavimo da je potrebno napraviti npr. najjednostavniji program koji učitava znak po znak sa tastature i ispisuje te iste znakove na ekran:

```
int main () {  
    char c = getc();  
    while (c != '\n') {  
        putc(c);  
        c = getc();  
    }  
}
```

- ❖ Kako realizovati funkcije *getc* i *putc*?

Mart 2020.

Copyright 2020 by Dragan Milićev

5

Šta je operativni sistem

- ❖ Ideja: zašto ovakve rutine (potprograme, delove softvera) ne implementirati jednom za dati tip računara i uređaja, držati ih u memoriji i obezbediti njihove usluge svim programima koji se izvršavaju na tom računaru
- ❖ Upravo to i *jest*e operativni sistem: skup rutina koje obavljaju operacije sa hardverskim uređajima računara, koje se nalaze u memoriji računara i koje se mogu koristiti kao usluge; te usluge programi koji se izvršavaju na tom računaru pozivaju kao tzv. *sistemske pozive*
- ❖ Jedina konfiguracija u kojoj se može bez operativnog sistema je računar koji izvršava samo jedan program, a čiji su sastavni deo sve rutine koje rukuju hardverom. Takvi sistemi danas su jedino krajnje jednostavni tzv. *ugrađeni (embedded)* sistemi kojim obrada informacija nije primarna svrha, već obradu informacija koriste kao sredstvo za drugi cilj - upravljanje nekim specijalizovanim inženjerskim, hardverskim sistemima i uređajima
- ❖ Međutim, čak i u tom slučaju, mnogo je praktičnije i jeftinije koristiti usluge postojećeg operativnog sistema, nego ih razvijati iznova i posebno; sistem koji koristi usluge operativnog sistema je lakši za održavanje (zbog modularnosti) i robusniji (zbog izolacije koju obezbeđuje OS)

Mart 2020.

Copyright 2020 by Dragan Milićev

8

Šta je operativni sistem

- ❖ Ako se ovo napravi za potrebe jednog programa, svakako ga treba koristiti i u drugim programima koji zahtevaju operacije sa istim uređajima
- ❖ Naravno, upravo zbog toga su ove operacije i apstrahovane u potprograme, kako bi njihova upotreba bila:
 - ❖ jednostavnija, jer je lakše pristupati uređaju kao apstraktnom objektu, preko jednostavnog i apstraktnog interfejsa ("daj mi učitani znak", "ispiši znak"), nego pristupati hardveru na niskom nivou apstrakcije kakav pruža hardver
 - ❖ ponovljiva u različitim programima
- ❖ Ali šta ako računar treba da izvršava više programa uporedo? Svi oni unutar svog koda imaju iste ove potprograme koji rade istu stvar? Neracionalno!
- ❖ Šta ako su ove funkcije realizovane za jedan tip uređaja, a uređaj se promeni? Treba menjati implementaciju tih funkcija i ponovo prevoditi sve programe koji ih koriste. Nepraktično!

Mart 2020.

Copyright 2020 by Dragan Milićev

7

Funkcije operativnog sistema

Osnovni zadaci i funkcije operativnog sistema su sledeće:

- ❖ Omogućiti pokretanje izvršavanja, kao i izvršavanje jednog programa ili više programa uporedo na raspoloživom hardveru računara
- ❖ Obezbediti efikasno korišćenje procesora, memorije i drugih hardverskih resursa, kako bi se programi izvršavali uporedo što je brže moguće
- ❖ Na sistemima na kojima je moguće pokrenuti neograničen broj izvršavanja programa, svakom od njih obezbediti potrebne resurse iz konačnog skupa raspoloživih - fizički ograničen skup resursa učiniti logički neograničenim
- ❖ Programima koji se izvršavaju obezbediti usluge obavljanja ulazno-izlaznih operacija pomoću uređaja koje računar poseduje
- ❖ Omogućiti razmenu informacija (komunikaciju) između programa koji se izvršavaju na istom računaru ili na udaljenim računarima
- ❖ Omogućiti programima pristup i operacije sa fajlovima na istom ili na udaljenim računarima
- ❖ Omogućiti druge usluge koje mogu biti korisne i potrebne različitim programima
- ❖ Obezbediti zaštitu od neželjenog uticaja izvršavanja jednog programa na izvršavanje drugih programa
- ❖ Obezbediti zaštitu računara (hardvera i softvera) od neovlašćenog pristupa i malicioznog softvera
- ❖ Obezbediti način da korisnici računara interaguju sa njim - *korisnički interfejs (user interface)*

Mart 2020.

Copyright 2020 by Dragan Milićev

10

Šta je operativni sistem

- ❖ Posledica: OS *apstrahuje hardware*, skrivajući njegovu implementaciju od programa iza apstraktnog interfejsa koji je jednostavan za upotrebu
- ❖ Zbog toga programi postaju nezavisni od hardvera i time lakše *prenosivi* na druge računare (pod uslovom da na njima postoji isti interfejs)
- ❖ Prema tome, OS čini sistem pogodnim za upotrebu; jednostavnije je sa njim nego bez njega (da ga nema, brzo bi bio izmišljen)

Mart 2020.

Copyright 2020 by Dragan Milićev

9

Delovi operativnog sistema

- ❖ Pojam osnovnih funkcija OS-a koje obezbeđuje kernel nije određen i strogo definisan, već je vrlo rastežljiv: neke od funkcija OS-a se svakako podrazumevaju pod tim, međutim, mnoge funkcionalnosti su kod nekih sistema deo kernela, a kod nekih izvan njega (izvršavaju se kao sistemski programi); ovo je pitanje arhitekture konkretnog OS-a i njihove raznolikosti
- ❖ Na primer, kod nekih sistema (npr. Windows) je korisnički interfejs deo kernela; kod drugih (npr. Linux) se korisnički interfejs izvršava kao sistemski program, tzv. *školjka (shell)*
- ❖ Sistemski programi mogu da obavljaju:
 - ❖ neke opšte sistemske stvari koje su uvek potrebne; ovakvi programi se pokreću pri pokretanju sistema ili na zahtev i onda izvršavaju uporedo sa korisničkim programima; zašto onda nisu unutar kernela? zbog modularnosti (obezbeđuje nezavisnost izvedbe) i robusnosti (otkaz u sistemskom programu neće uzrokovati otkaz celog sistema, dok otkaz u kernelu hoće), a za tim nema potrebe
 - ❖ neke radnje koje su opšteg karaktera (npr. kopiranje fajlova, pravljenje rezervne kopije, prikaz sadržaja fajlova u nekim standardnim formatima itd.) koje su često potrebne većini korisnika; zašto one nisu deo kernela? jer za tim nema potrebe - ako ove radnje nisu potrebne, ne zauzimaju prostor u operativnoj memoriji koju zauzima kernel, a pokreću se ako su potrebni

Mart 2020.

Copyright 2020 by Dragan Milićev

12

Delovi operativnog sistema

- ❖ Tradicionalno shvatanje osnovnih komponentata operativnog sistema:
 - ❖ *jezgro ili kernel (kernel)*: deo operativnog sistema koji je uvek učitao u operativnu memoriju ili se u nju učitava pri uključivanju računara i tu ostaje stalno do isključenja, izvršava osnovne funkcije operativnog sistema i pruža usluge programima koji se na računaru izvršavaju
 - ❖ *sistemski programi (system program)*: programi koji se izvršavaju kao i svi ostali, samo što se isporučuju kao sastavni deo operativnog sistema, jer obavljaju neke opšte radnje
 - ❖ *korisnički interfejs (user interface)*: deo za interakciju sa korisnikom (čovekom)
- ❖ Današnje komercijalno shvatanje - sve što je proizvođač uvrstio u paket instalacije OS-a: sve osnovne komponente, ali i niz drugih korisničkih aplikativnih programa (aplikacije za personalnu upotrebu, za razne usluge na Internetu i pristup društvenim mrežama, prikaz fajlova različitih formata, igrice, ...)

Mart 2020.

Copyright 2020 by Dragan Milićev

11

Korisnički interfejs

Interpreter komandne linije (*command line interpreter*, CLI) ili *konzola* (*console*) je sistemski program (školjka) ili deo kernela koji interaguje sa korisnikom samo pomoću sledećih uređaja:

- ❖ tastatura: ulazni, znakovno orijentisan, sekvencijalan uređaj - učitava se znak po znak u strogom poretku, u kom su ti znakovi otkucani
- ❖ ekran (monitor): izlazni, znakovno orijentisan, sekvencijalan uređaj - ispisuje se znak po znak u strogom poretku, u kom su ti znakovi poslati na uređaj; znak sa posebnim kodom CR (*carriage return*) uzrokuje da se tekuća pozicija za ispis (kurzor) pomera na početak reda; znak sa posebnim kodom LF (*line feed*) uzrokuje da se tekuća pozicija za ispis (kurzor) pomera na sledeći red (monitor sam pomera sadržaj nagore kada ispis dođe do poslednje linije, *scroll*);

```
Last login: Thu Mar 12 21:44:51 on ttye003
Dragan-Milic@x-drt ~ % cat /dev/urandom | fold -w 60 | tr -d '\n' | fold -w 60 | tr -d '\n' | cat
dr0s----- 6 00:11:ce9 192 Oct 27 2015 Applications
dr0s----- 7 00:11:ce9 224 Mar 2 21:51 Desktop
dr0s----- 1 00:11:ce9 541 Sep 11 2013 Documents
dr0s----- 13 00:11:ce9 415 Mar 1 2012 Downloads
dr0s----- 9 79 00:11:ce9 2528 Oct 26 11:39 Library
dr0s----- 5 00:11:ce9 163 Aug 25 2017 Music
dr0s----- 6 00:11:ce9 192 Jan 6 2017 Movies
dr0s----- 17 00:11:ce9 544 Aug 16 2013 Pictures
dr0s-rs-+ 4 00:11:ce9 128 Sep 27 2013 Public
dr0s-rs-+ 3 00:11:ce9 163 Mar 22 2017 Sites
Dragan-Milic@x-drt ~ %
```

Mart 2020.

Copyright 2020 by Dragan Milicev

14

Korisnički interfejs

- ❖ Dva tipa korisničkog interfejsa:
 - ❖ *interpreter komandne linije* (*command line interpreter*, CLI): tradicionalni, datira iz doba prvih interaktivnih računara
 - ❖ *grafički korisnički interfejs* (*graphical user interface*, GUI): moderniji (iako je prvi osmišljen još 1973. godine)

```
Last login: Thu Mar 12 21:44:51 on ttye003
Dragan-Milic@x-drt ~ % cat /dev/urandom | fold -w 60 | tr -d '\n' | fold -w 60 | tr -d '\n' | cat
dr0s----- 6 00:11:ce9 192 Oct 27 2015 Applications
dr0s----- 7 00:11:ce9 224 Mar 2 21:51 Desktop
dr0s----- 1 00:11:ce9 541 Sep 11 2013 Documents
dr0s----- 13 00:11:ce9 415 Mar 1 2012 Downloads
dr0s----- 9 79 00:11:ce9 2528 Oct 26 11:39 Library
dr0s----- 5 00:11:ce9 163 Aug 25 2017 Music
dr0s----- 6 00:11:ce9 192 Jan 6 2017 Movies
dr0s----- 17 00:11:ce9 544 Aug 16 2013 Pictures
dr0s-rs-+ 4 00:11:ce9 128 Sep 27 2013 Public
dr0s-rs-+ 3 00:11:ce9 163 Mar 22 2017 Sites
Dragan-Milic@x-drt ~ %
```



Mart 2020.

Copyright 2020 by Dragan Milicev

13

Korisnički interfejs

Grafički korisnički interfejs:

- ❖ rasterski ekran (matrica piksela)
- ❖ pokazivački ulazni uređaj (miš) ili ekran osetljiv na dodir (*touch screen*)
- ❖ pojam *radne površine* (*desktop*)
- ❖ vizuelne predstave resursa (objekata) - ikonice za fajlove, programe itd.
- ❖ intuitivne i polimorfne operacije sa objektima: klik, dupli klik, prevlačenje (*drag&drop*), gestikulacije (*gestures*)



Mart 2020.

Copyright 2020 by Dragan Milićev

16

Korisnički interfejs

Interpreter ciklično obavlja sledeće:

- ❖ na ekran ispisuje znak koji simbolizuje spremnost za učitavanje komande (*command prompt*)
- ❖ učitava znak po znak sa tastature, dok ne naiđe na poseban znak kojim se koduje taster *Enter*
- ❖ uneseni niz znakova rastavlja na podstringove; po pravilu, prvi interpretira kao komandu, ostale kao argumente komande
- ❖ izvršava komandu; komanda kao svoj efekat može dati neki ispis na ekran
- ❖ po završetku izvršavanja komande, ponavlja sve ovo

```
last login: Thu Mar 12 21:44:51 on ttyss002
Dragan@MacBook-Air:~$ DMilicev ls -o
total 0
drwx----- 6 DMilicev 192 Oct 27 2015 Applications
drwx----- 7 DMilicev 224 Mar  2 21:51 Desktop
drwx----- 17 DMilicev 544 Sep 13 2018 Documents
drwx----- 13 DMilicev 416 Mar  3 20:23 Downloads
drwx----- 73 DMilicev 2528 Oct 20 11:39 Library
drwx----- 5 DMilicev 160 Aug 29 2017 Movies
drwx----- 6 DMilicev 192 Jan  6 2017 Music
drwx----- 17 DMilicev 544 Aug 10 2015 Pictures
drwxr-xr-x+ 4 DMilicev 128 Sep 27 2013 Public
drwxr-xr-x  5 DMilicev 160 Mar 21 2017 Sites
drwxr-xr-x  3 DMilicev  96 Jul 27 2017 workspace-jupyter
Dragan@MacBook-Air:~$ DMilicev _
```

Command prompt

Komanda

Argument komande

Ispis rezultata komande

Mart 2020.

Copyright 2020 by Dragan Milićev

15

Sistemske pozivi

Sistemske pozivi

- ❖ *Sistemske pozivi* (*system call*) je metod kojim program koji se izvršava na nekom OS-u traži određenu uslugu od tog OS-a
- ❖ Kada se program piše na višem programskom jeziku, a jezici C i C++ su osnovni i klasičan slučaj, sistemske pozivi se vrše pozivanjem bibliotечnih potprograma iz standardnih, sistemskih biblioteka - sistemske pozivi se vidi kao najobičnija C funkcija
- ❖ Skup dostupnih bibliotечnih potprograma koji vrše sistemske pozive na nekom OS-u čini *aplikativni programski interfejs* (*application programming interface*, API) datog OS-a na datom programskom jeziku
- ❖ Implementacija tih potprograma unutar biblioteke sadrži instrukcije koje vrše sistemske pozivi na mašinskom, binarnom nivou, urađeno na način kako dati OS to zahteva - detalji kasnije; ovaj nivo naziva se *interfejs sistemskih poziva* (*system call interface*)
- ❖ Poziv ovakvog potprograma za pozivajući program ima istu semantiku kao i poziv običnog potprograma: izvršavanje pozivajućeg programa se nastavlja tek kada se sistemske pozivi završi i vrati kontrolu pozivaocu, što može značiti i *suspenciju* (odlaganje, zaustavljanje) izvršavanja programa na neko vreme

Mart 2020.

Copyright 2020 by Dragan Milićev

Sistemske pozivi

```
#include <stdio.h>
#include <string.h>
#define N 80
char buffer[N+1];

int main () {
    int i=0;
    char c = getc(stdin);

    while (i<N && c!='\n') {
        buffer[i++] = c;
        c = getc(stdin);
    }
    buffer[i] = '\0';

    char* p = strchr(buffer, ' ');
    printf("Arguments: %s\n", p);

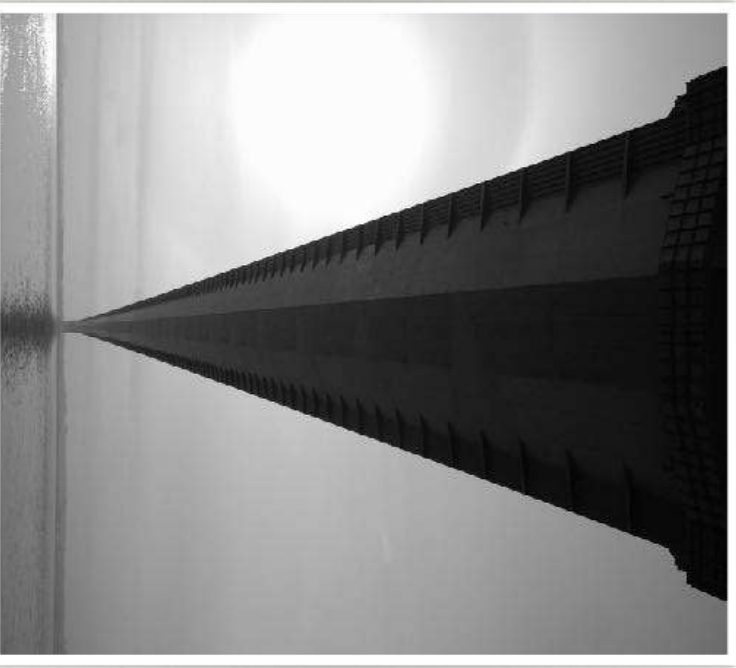
    exit(0);
}
```

strchr nije sistemske poziv, već obična bibliotечna funkcija koja pretražuje niz znakova u memoriji samog programa i ne traži nikakvu uslugu od OS-a

printf je složena bibliotечna funkcija koja analizira format dat prvim argumentom, konvertuje ostale u nizove znakova i onda vrši višestruke sistemske pozive *putc*

Glava 2: Osnovni pojmovi

- ❖ Sistemske pozivi
- ❖ Proces
- ❖ Multiprogramiranje
- ❖ Konzola i standardni ulaz / izlaz
- ❖ Preotimanje i raspodela vremena
- ❖ Fail



Proces

- ❖ *Proces (process)* je jedno izvršavanje nekog programa na računaru, koje potencijalno teče uporedo sa drugim takvim izvršavanjima istog ili drugih programa
- ❖ *Program (program)* je statički zapis, specifikacija onoga što računar treba da uradi; jedna aktivacija programa predstavlja proces; nad istim programom može se pokrenuti više procesa, više nezavisnih izvršavanja, svako sa svojim podacima
- ❖ Računarski sistem može biti:
 - ❖ monoprogramski: izvršava samo jedan program, a ne bilo koji, proizvoljan program koji mu se zada; primer su ugrađeni (*embedded*) sistemi koji izvršavaju samo program koji je ugrađen u sistem, a ne bilo koji program koji zada korisnik (npr. jednostavni kućni uređaji - veš mašine, šporeti, usisivači itd.)
 - ❖ multiprogramski: mogu da mu se zadaju različiti, proizvoljni programi pisani za taj sistem
- ❖ Operativni sistem može biti:
 - ❖ monoprocesni: omogućava samo jedno izvršavanje programa u datom trenutku, sledeći program može da se pokrene tek kad se prethodni završi
 - ❖ multiprocesni: omogućava uporedno izvršavanje više procesa

Monoprocesni			Multiprocesni	
Monoprogramski	Jednostavni ugrađeni računari		Nema praktičnog značaja	
Multiprogramski	Najraniji (tzv. paketni) računari Kućni mikroročunari Prvi personalni računari		Svi današnji operativni sistemi opšte namene	

Mart 2020.

Copyright 2020 by Dragan Milićev

20

Sistemski poziv

- ❖ Neki najpoznatiji i najšire zastupljeni API za sistemske pozive:
 - ❖ standardna biblioteka jezika C/C++ (tzv. *libc*) sadrži, između ostalih, i funkcije koje u sebi imaju sistemske pozive (npr. neke iz *stdlib.h*, *stdio.h*, *thread.h*)
 - ❖ POSIX (*Portable Operating System interface based on Unix*): standardni API koji je napravljen tako da odgovara skupu koncepata i operacija, kao i semantici tih koncepata i operacija na sistemu Unix, sa ciljem da programi koji koriste ovaj API budu prenosivi na sve sisteme nalik Unixu (tzv. *Unix-like systems*) koji na ih isti ili sličan način podržavaju: razne verzije sistema Unix, AIX, Solaris, HP-UX, Linux itd. Međutim, postoje implementacije ovog API i na svim ostalim operativnim sistemima opšte namene (Windows, Mac OS X itd.)
 - ❖ Windows API (WinAPI, Win32): API za sistem Windows (u različitim izvedbama)

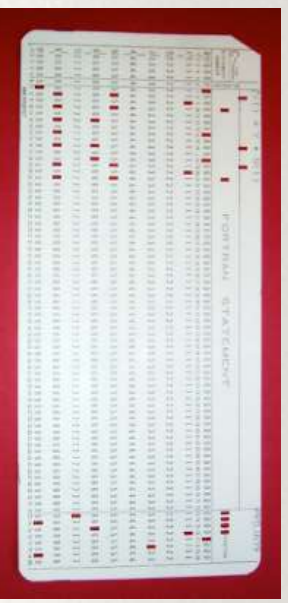
Mart 2020.

Copyright 2020 by Dragan Milićev

19

Multiprogramiranje

- ❖ Tekst programa ili podaci unose se, znak po znak i liniju po liniju (od tada datira ograničenje linije na 80 znakova), preko tastature u *bušač kartica* (*card puncher*): uređaj koji nije povezan sa računarom, već služi da binarno koduje svaki znak u liniji postojanjem ili nepostojanjem rupica na 7 ili 8 pozicija (binarni kod znaka)
- ❖ Za svaku liniju pravi se jedna bušena kartonska kartica
- ❖ Ceo špil kartica za jedan program i njegove ulazne podatke, ili za *paket* (*batch*, odatle naziv) sa više takvih programa, unosi se u ulazni uređaj - *čitač bušenih kartica* (*punched card reader*) koji učitava jednu po jednu karticu i optoelektronskim uređajem (fotočelijom) očitava binarni zapis na kartici



Bušena kartica za jednu liniju FORTRAN programa (Wikipedia)

Mart 2020.

Copyright 2020 by Dragan Milićev

22

Multiprogramiranje

- ❖ Prvi komercijalni računari, u upotrebi krajem 1950-ih i tokom 1960-ih, tzv. *paketni* (*batch*) sistemi, bili su monoprocesni
- ❖ Iako veliki po gabaritu, imali su vrlo skroman skup uređaja, ali principijelno istu arhitekturu (procesor, operativna memorija), kao i:
 - ❖ ulazni uređaj: samo čitač bušenih kartica (*punched card reader*)
 - ❖ izlazni uređaji: bušač kartica (*card puncher*) ili *linijski štampač* (*line printer*)
- ❖ Nešto kasnije koriste se i *magnetne trake* (*magnetic tape*) za snimanje ili učitavanje programa i podataka, ali i dalje sekvencijalno, po redosledu kretanja trake



IBM 704, 1957.

Mart 2020.

Copyright 2020 by Dragan Milićev

21

Multiprogramiranje

Promenljive čija imena počinju sa I, J, K, L, M i N su podrazumevano celobrojne

Identifikator ulazne jedinice (5) - čitač kartica; identifikator linije programa sa formatom (501)

- ❖ Program na višem programskom jeziku (FORTRAN je u to vreme najpopularniji) se sastojao od instrukcija i ulaznih podataka koje taj program treba da obradi (FORTRAN ima posebnu direktivu *DATA* za definisanje podataka)
- ❖ Program po pravilu viši neku matematičku obradu ulaznih podataka (*FORmula TRANslation*) i rezultate obrade ispisuje kao nizove znakova na izlaz - linijski štampač ili bušać kartica
- ❖ Programi nisu bili interaktivni: za vreme izvršavanja programa nema nikakve interakcije sa korisnikom (odatle potiče naziv *batch* za obradu "u pozadini")
- ❖ Program sa svojim ulaznim podacima podnet na izvršavanje, pod nazivom koji se tada koristio *posao* (*job*), smešta se na ulazni uređaj (čitač kartica ili kasnije magnetna traka) i čeka da bude pokrenut

Mart 2020.

Copyright 2020 by Dragan Milićev

24

Multiprogramiranje

- ❖ Linijski štampač: izlazni, sekvencijalni, znakovno orijentisan uređaj koji štampa znak po znak, redom kojim ih računar šalje:
 - ❖ štampač uvlači neprekidnu traku papira sa perforiranim ivicama; pomoću perforacije valjak sa zupcima uvlači papir
 - ❖ glava se pomera sleva nadesno i iglicama, kojima udara papir preko mastiljave trake, tačkicama iscrtaava znakove (matični štampač, npr. znak od 8x8 tačaka)
 - ❖ specijalni znak *CR* (*carriage return*) pomera glavu na početak reda, na levo
 - ❖ specijalni znak *LF* (*line feed*) uzrokuje da valjak pomeri papir za jedan red dalje - prelazak u novi red
 - ❖ ostali specijalni, kontrolni znaci (tzv. *non-printable characters*): *page feed*, *tab* i ostali vode poreklo iz ovih uređaja, kao upravljački znakovi kojima računar kontroliše format štampe



Linijski štampač IBM 1403 (Wikipedia)

Mart 2020.

Copyright 2020 by Dragan Milićev

23

Multiprogramiranje

- ❖ Karakteristično ponašanje svakog programa - smenjuju se dve faze:
 - ❖ *nalet izvršavanja na procesoru (CPU burst)*: sekvenca instrukcija koje rade samo sa registrima procesora i operativnom memorijom, ne koristite usluge OS-a, tj. U / I operacije
 - ❖ *ulazno-izlazna operacija (I/O operation)*: proces traži sistemsku uslugu, tj. ulaznu ili izlaznu operaciju, npr. učitavanje podataka ili ispis rezultata
- ❖ Problem: dok se obavlja U / I operacija, procesor čeka, tj. ne izvršava nikakve korisne instrukcije
- ❖ Zbog odnosa brzine rada procesora i uređaja (i u ono vreme, kada su procesori bili spori, bili su sporiji i uređaji, pa je odnos trajanja naleta uvek nekoliko redova veličine), procesor je jako slabo iskorišćen - dugi su periodi u kojima on čeka u odnosu na kratke periode korisnog rada
- ❖ Za sisteme koji su u to vreme bili izuzetno retki i skupi, ovakvo neiskorišćenje ključnog resursa računara - procesora - bilo je neprihvatljivo



Mart 2020.

Copyright 2020 by Dragan Milićev

26

Multiprogramiranje

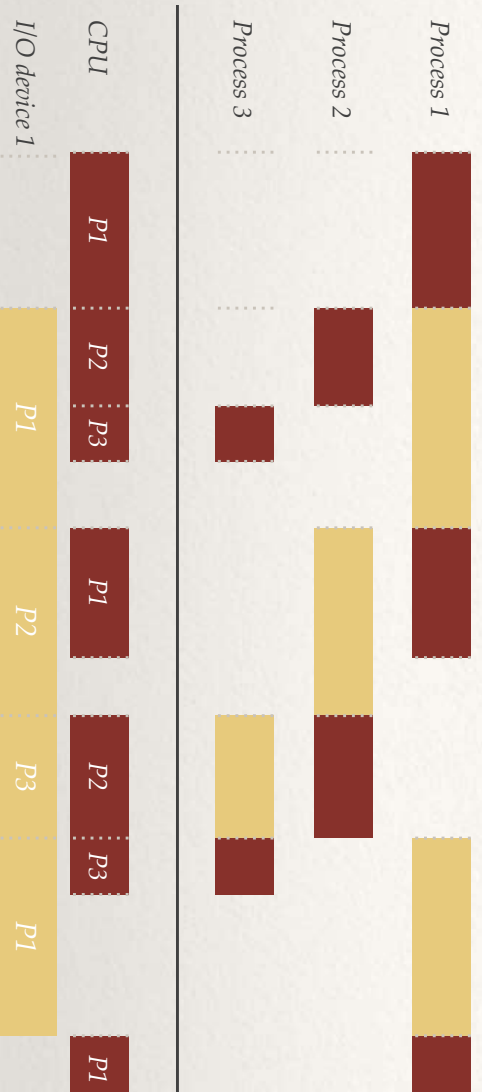
- ❖ Operativni sistem (zapravo njegova preteča, tada se zvao "monitor") ima jednostavan zadatak: sa ulaznog uređaja (čitača kartica ili magnetne trake) učita posao (program sa podacima) koji je sledeći na redu (po redosledu kojim su podneti), učita ga u memoriju i pokrene njegovo izvršavanje
- ❖ Za vreme izvršavanja, izvršava se samo taj posao (proces), koji je, osim OS-a, jedini u operativnoj memoriji računara
- ❖ OS procesu obezbeđuje usluge ulazno-izlaznih operacija (učitavanje podataka ili ispis rezultata), ali isključivo znakovno orijentisanih i sekvencijalnih; uređaje program identifikuje identifikatorima (npr. ceo broj), a OS obezbeđuje da se operacija usmerava na odgovarajući uređaj
- ❖ Kada proces završi (naredba *STOP* u Fortranu), OS učitava naredni posao

Mart 2020.

Copyright 2020 by Dragan Milićev

25

Multiprogramiranje



- ❖ Posledice:
 - ❖ procesor je mnogo bolje iskorišten: u idealnom slučaju 100%, ali u realnom ispod toga; stepen iskorištenja raste sa porastom stepena multiprogramiranja (broj aktivnih procesa), jer je veća šansa da postoji proces koji može da nastavi izvršavanje
 - ❖ procesor i U/I uređaji rade *parallelno*, istovremeno, pa je moguće uraditi više posla za isto vreme
 - ❖ zato je ukupna *propusnost (throughput)* sistema - količina urađenog posla, odnosno broj završenih procesa u jedinici vremena - daleko veća nego za jednoprocetni sistem
 - ❖ svaki proces se sada izvršava potencijalno duže nego što bi se izvršavao kad bi bio sam u sistemu, jer trpi čekanje zbog toga što neki drugi proces koristi resurs koji je njemu potreban (procesor, U/I uređaj)

Mart 2020.

Copyright 2020 by Dragan Milićev

28

Multiprogramiranje

- ❖ Ideja za rešenje: u operativnu memoriju učitati više procesa i izvršavati ih *uporedo*, na sledeći način: dok jedan proces čeka na završetak svoje U/I operacije, procesor može da izvršava instrukcije drugog procesa - sistem postaje *multiprocetni*
- ❖ Ovaj koncept uporednog izvršavanja procesa, pri čemu se procesor *vremenski multipleksira* između različitih procesa (u jednom intervalu izvršava se deo jednog procesa, pa onda nekog drugog itd.) naziva se *multiprogramiranje (multiprogramming)*
- ❖ Multiprogramiranje je jedan od ključnih pomaka u razvoju računarstva i koncept koji je u upotrebi i danas u svim operativnim sistemima
- ❖ Osnovni preduslov jeste to da se u memoriju učita više procesa
- ❖ Jedan od bitnih tehnoloških preduslova za ovo bilo je uvođenje *magnetnih diskova*: za razliku od magnetnih traka, OS može da pristupa podacima na disku *direktno*, u proizvoljnom redosledu, bez značajnih razlika u vremenu odziva kao kod traka (zbog potrebe premotavanja), bez obzira na to kako su podaci smešteni na samom disku (na kom su mestu)
- ❖ Disk je *blokovski orijentisan ulazno-izlazni uređaj*: podaci se mogu i učitavati i upisivati na disk, ali se prenose isključivo u blokovima fiksne veličine (npr. od po 512 B)
- ❖ Zbog ovoga je skup poslova koji su podneti na izvršavanje mogao da se snimi na disk, a da OS sa njega učitava one koje odluči da će izvršavati, na osnovu zauzeća procesora i memorije, a ne na osnovu redosleda kojim su podneti

Mart 2020.

Copyright 2020 by Dragan Milićev

27

Konzola i standardni ulaz/izlaz

- ❖ Dalji razvoj računarstva donosi nov koncept *interaktivnih* računara, tzv. *mejnfrjem računara* (*mainframe*); pojavljuju se 1970-ih i u upotrebi su sve do 1990-ih
- ❖ Računar se sastoji iz svih standardnih komponentata (procesor, memorija) i U/I uređaja (diskovi, linijski štampači, magnetne trake), ali se sada omogućava i interakcija sa korisnicima preko tzv. *terminala* (*terminal*) ili *konzola* (*console*)
- ❖ Na računar je priključeno na desetine ili čak i stotine terminala; računar ih vidi kao svoje U/I uređaje
- ❖ Jedan terminal (konzola) se sastoji iz dva uređaja:
 - ❖ tastatura: ulazni, sekvencijalni, znakovno orijentisan uređaj
 - ❖ monitor (monohromatski - jednobojni): izlazni, sekvencijalni, znakovno orijentisan uređaj; ponaša se potpuno analognog linijskom štampaču, jer računar na njega može da šalje znakove koje monitor ispisuje u linijama (uz kontrolne znakove CR, LF itd.), samo što umesto uvlačenja papira, monitor radi pomeranje prikaza (*scroll*) kada ispuní ekran



Terminal računara IBM 360

Mart 2020.

Copyright 2020 by Dragan Miličev

30

Multiprogramiranje

OS sada ima mnogo novih zadataka i postaje značajno složeniji:

- ❖ Iz skupa poslova koji su podneti za izvršavanje (*submitted*), treba izabrati one za koje će pokrenuti procese - pitanje *raspoređivanja poslova* (*job scheduling*); zbog drugačije koncepcije današnjih sistema, u kojima nove procese pokreću korisnici ili njihovi procesi, ova funkcionalnost više ne postoji u sistemima, dok sve ostale i dalje ostaju aktuelne
- ❖ Treba obezbediti prelazak sa izvršavanja jednog procesa na izvršavanje drugog procesa - *promenu konteksta* (*context switch*) tako da procesor nastavi izvršavanje instrukcija tog drugog procesa, ali tako da se onaj prvi proces može kasnije nastaviti kao da nije bio prekidán
- ❖ Iz skupa procesa koji mogu da nastave izvršavanje, izabrati onaj koji će dobiti procesor - *raspoređivanje procesa na procesoru* (*process/processor scheduling*)
- ❖ Treba omogućiti da svaki proces adresira svoje instrukcije i podatke u memoriji bez obzira na to što se ne zna unapred, prilikom prevođenja programa, mesto u memoriji na kom će proces biti smešten (alociran) - problem *adresiranja memorije*
- ❖ Treba smestiti više procesa u operativnu memoriju i rukovati slobodnim i zauzetim delovima memorije (*memory management*)
- ❖ Kada procesi izdaju zahteve za U/I operacijama na uređajima, treba opsluživati te zahteve na neki način i nekim redom - pitanje *raspoređivanja operacija sa uređajima* (*device scheduling*)
- ❖ Treba se zaštititi od situacije u kojoj neki proces nikada ne pozove sistemski poziv (npr. zbog greške ili loše namere) ili ga ne poziva jako dugo - treba mu *preoteti procesor* (*preemption*)
- ❖ Treba zaštititi delove memorije koji pripadaju drugim procesima, kao i kernelu, od uticaja drugih procesa zbog grešaka ili loše namere - pitanje *zaštite* (*protection*)

Mart 2020.

Copyright 2020 by Dragan Miličev

29

Konzola i standardni ulaz/izlaz

- ❖ Za udobno korišćenje interpretera komandi i ostalih programa na terminalu, važno je imati dobar odziv računara, isto kao i danas: na neku akciju korisnika, npr. pritisak tastera, računar treba da odreaguje *nekakvin* signalom (npr. prosto samo tako što će ispisati taj isti učitani znak, tzv. eho) u kratkom roku reda pola sekunde (onoliko brzo koliko i korisnik pritisaka tastere), inače korisnik ima osećaj da računar “nije primio” akciju i ne prati kucanje
- ❖ Eho pritisnutog tastera u komandnoj liniji nije efekat nikakve direktne hardverske veze između tastature i monitora, već CLI, kao proces, mora da učitati znak sa tastature, sistemskim pozivom, a onda taj znak da obradi i ako treba, ispiše na ekran, opet sistemskim pozivom (može i da ga ne ispiše, ako taj znak ima neko drugo značenje u tom trenutku)

- ❖ Zbog toga interaktivan proces (CLI ili korisnički proces) ima sledeći generički oblik:

```
loop
wait_for_user_input;
compute_response;
display_response;
end loop;
```

Na primer, učitaj znak sa tastature (standardnog ulaza)

Na primer, ispiši znakove na ekran (standardni izlaz)

Mart 2020.

Copyright 2020 by Dragan Milićev

32

Konzola i standardni ulaz/izlaz

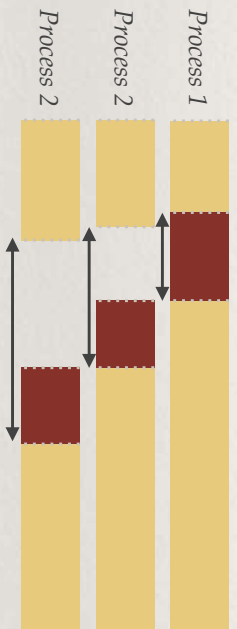

- ❖ Na svakom terminalu može da radi po jedan korisnik koji se najpre prijavljuje na sistem (*log in*), kao što se to radi i danas, a onda izvršava komande preko interpretera komandne linije (CLI); koncept *konzole* koji postoji u današnjim sistemima je zapravo virtualizacija ovakvih starih terminala
- ❖ Komande mogu biti neke sistemske, poput ispisivanja sadržaja tekućeg direktorijuma i slično, ili pokretanja procesa
- ❖ Da bi se implementirala obrada zahteva sa terminala, OS izvršava po jedan proces za isti program - CLI, za svaki terminal, odnosno korisnika
- ❖ Ovaj proces koristi svoj uređaj (konzolu) za svoj ulaz (učitavanje linije znakova koje interpretira kao komandu sa argumentima) i ispis rezultata komande
- ❖ Kada komanda znači pokretanje novog procesa, CLI kao *roditeljski proces*, kreira nov proces *dete* nad zadatim programom, odgovarajućim sistemskim pozivom; CLI se po pravilu *suspenduje* dok se pokrenuti proces ne završi
- ❖ Da bi pokrenuti proces-dete mogao da učitava znakove sa (iste) tastature sa koje je pokrenut i ispisuje svoj izlaz na (isti) monitor sa kog je pokrenut:
 - ❖ OS uvodi koncept *logičkog uređaja*, tzv. *standardnog ulaza* i *standardnog izlaza*
 - ❖ proces *dete nasleđuje* standardni ulaz i standardni izlaz od roditelja
 - ❖ OS obezbeđuje mehanizam kojim se argumenti iz komandne linije procesa roditelja mogu preneti procesu detetu; proces dete ih onda može dobiti sistemskim pozivom (program na jeziku C ih onda vidi kao argumente *argc* i *argv* funkcije *main*)

Mart 2020.

Copyright 2020 by Dragan Milićev

31

Preotimanje i raspodela vremena

- ❖ Za udoban rad nije bitan samo relativno brz, nego i *racionalan* odziv
 - ❖ Posmatrajmo situaciju u kojoj više (potencijalno mnogo) korisnika izvršava procese, npr. CLI, svi su jednakog "prava prvenstva", pa ne preotimaju procesor jedan drugom
 - ❖ Neka su svi korisnici uradili neku akciju (pritisnuli taster) u približno istom trenutku. Kakav će odziv korisnici dobiti?
- 
- 
- ❖ U zavisnosti od toga u kom hronološkom poretku su se dogodile akcije korisnika, što je potpuno nepredvidiv i slučajan poredak, korisnici će dobiti različite odzive - vrlo brze ili vrlo spore
 - ❖ Tokom rada, korisnik će osećati varijacije u vremenu odziva, od vrlo brzog do dugog, potpuno nepredvidivo, u zavisnosti od broja i aktivnosti drugih korisnika
 - ❖ Ovaj neprijatan efekat postaje sve vidljiviji što je broj korisnika veći - korisnici "smetaju" jedni drugima

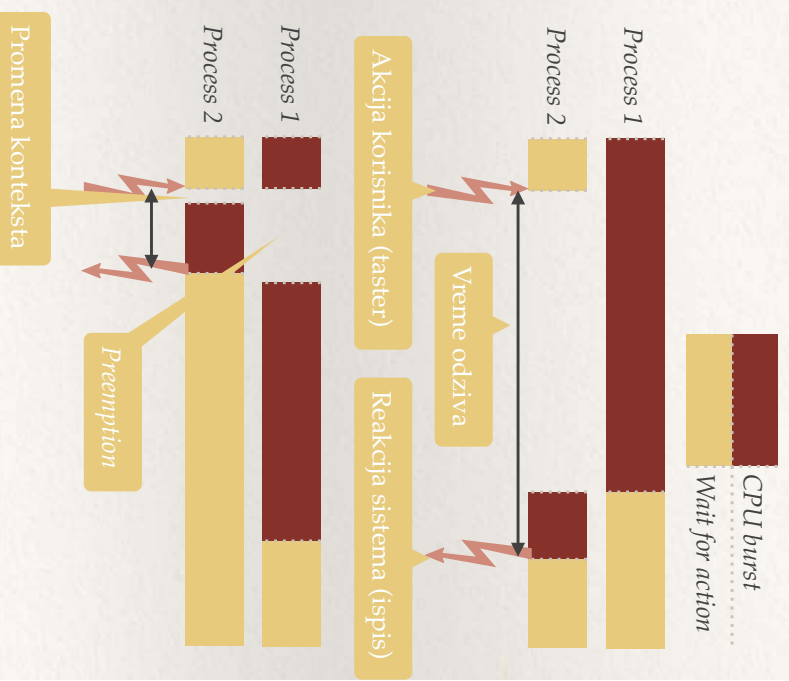
Mart 2020.

Copyright 2020 by Dragan Milićev

34

Preotimanje i raspodela vremena

- ❖ Kakav će biti odziv ako se OS implementira tako da vrši promenu konteksta samo onda kada neki proces pozove sistemski poziv, npr. zatraži U/I operaciju?
- ❖ Ako procesor (jedan u računaru) trenutno izvršava neki drugi proces i taj proces ima dugačak nalet izvršavanja na procesoru (*CPU burst*), promena konteksta se neće dogoditi sve dok taj proces ne završi taj svoj nalet i ne izvrši neki sistemski poziv, kada kernel preuzima kontrolu
- ❖ Ako je tokom tog naleta korisnik na drugom terminalu pritisnuo taster, vreme odziva (*response time*) na njegovu akciju može da bude veoma dugo - neudoban rad
- ❖ Zato je neophodno obezbediti mehanizam kojim bi:
 - ❖ procesor dobio signal da treba da prekine tekuće izvršavanje i pređe na kod kernela - *prekid (interrupt)*
 - ❖ kernel izvrši promenu konteksta i procesor preda procesu koji treba da da odziv i koji je zbog toga hitniji
- ❖ Ovaj postupak naziva se *preotimanje procesora (preemption)*



Mart 2020.

Copyright 2020 by Dragan Milićev

33

Preotimanje i raspodela vremena

- ❖ Osim svih dosadašnjih zadataka koje ima i pitanja koja treba da reši multiprocesni OS, ovakav sistem sada ima i dodatne probleme:
 - ❖ Obezbediti preotimanje na prekid od hardverskih uređaja i od vremenskog brojača (tajmera, *timer*) zbog raspodele vremena
 - ❖ Pošto korisnici mogu da pokrenu neograničen broj procesa, rešiti pitanje njihovog uporednog izvršavanja i smeštanja u ograničenu operativnu memoriju
 - ❖ Uvodi se koncept korisnika, kao entiteta koji OS poznaje, pa sistem postaje *multikorisnički* (*multiuser*); za svakog korisnika treba obezbediti zaštitu pristupa njegovim podacima i programima, kao i ličnim podacima (kredencijalima, *credentials* - korisničko ime i lozinka)
 - ❖ Ovakvi sistemi su obračunavali usluge svojim korisnicima: utrošeno procesorsko vreme, prostor na disku, utrošak štampača itd, pa OS treba da vodi evidenciju o tome
- ❖ I današnji operativni sistemi opšte namene koriste raspodelu vremena za procese

Mart 2020.

Copyright 2020 by Dragan Milićev

36

Preotimanje i raspodela vremena

- ❖ Rešenje: kada se proces aktivira i OS mu dodeli procesor, ne daje mu da se izvršava do kraja njegovog naleta izvršavanja, dok on ne zatraži sistemski poziv, već mu se procesor da na ograničeno vreme (reda nekoliko desetina milisekundi), tzv. *vremenski odrezak* (*time slice*), nakon kog mu OS preotme procesor, izvrši promenu konteksta i procesor preda drugom procesu
- ❖ Na ovaj način procesi dele vreme na procesoru - koncept *raspodele vremena* (*time sharing*)



- ❖ Sada će neki procesi imati duži, a neki kraći odziv nego u prethodnom slučaju, ali će svi imati ravnomanan odziv
- ❖ Zato korisnici neće osećati jedni druge, više ne smetaju jedni drugima: svaki korisnik će imati utisak kao da sam radi za računarom
- ❖ Vreme odziva polako raste sa brojem aktivnih korisnika, ali je vreme odziva i dalje ravnomerno; sve dok broj aktivnih korisnika ne pređe neku metu, rad i dalje ostaje udoban

Mart 2020.

Copyright 2020 by Dragan Milićev

35

Fajl

- ❖ Uvođenjem fajla koncept logičkog ulaznog ili izlaznog uređaja procesa se dalje apstrahuje: proces može učtavati znakove sa svog *standardnog ulaza*, i ispisivati ih na svoj *standardni izlaz*, ne znajući šta se iza tih apstrakcija krije - OS implementira operacije u zavisnosti od toga šta se iza krije (polimorfizam unutar OS-a)
- ❖ OS može preusmeriti standardni ulaz ili izlaz pri pokretanju procesa iz komandne linije:
 - ❖ u tekstualni fajl ili iz tekstualnog fajla - *redirekcija (redirection)*, $>$ i $<$

```
ls -al > listing.txt
```

Izlaz komande / procesa *ls -al* preusmerava se u fajl *listing.txt*
 - ❖ izlaz jednog procesa prosleđuje se kao ulaz drugog procesa - *cevovod (pipe)*, $|$

```
cat listing.txt | less
```

Izlaz komande / procesa *cat* usmerava se na ulaz komande / procesa *less*
- ❖ Radi bolje organizacije i preglednosti, fajlovi se organizuju u *direktorijume (directories)*
- ❖ OS sada dobija nove zadatke:
 - ❖ upravljati fajl sistemima na različitim uređajima
 - ❖ obezbediti zaštitu kroz *prava pristupa* korisnicima do različitih fajlova i direktorijuma

Mart 2020.

Copyright 2020 by Dragan Milićev

38

Fajl

- ❖ Uvođenjem višekorisničkih sistema sa sve raznovrsnijim uređajima za smeštanje programa i podataka (diskovi, pre njih doboši, magnetne trake), operativni sistemi uvode i usluge pristupa tim programima i podacima (čitanje i upis)
- ❖ U početku je svaki OS nudio svoj API za rad sa podacima na uređajima, i to na niskom nivou apstrakcije, bliskom fizičkom načinu organizacije podataka na uređaju i načinu pristupa do njih
- ❖ Zato su programi bili jako zavisni i od operativnog sistema, ali i od uređaja na kom su podaci koje koriste
- ❖ Zbog toga je bilo neophodno apstrahovati pristup korisničkim podacima i programima, osmišljavajući univerzalan logički koncept koji će sakriti sve detalje i raznolikosti načina fizičkog pristupa i organizacije podataka na uređaju
- ❖ Upravo to predstavlja koncept *fajla (file)*: univerzalan, jednobrazan, apstraktan logički koncept za smeštanje podataka i programa na najrazličitijim uređajima; pristup do sadržaja fajla je jednobrazan i obavlja se kroz standardizovan API sistemskih poziva, dok OS sakriva sve raznolikosti i promenljivosti uređaja na kojima su ti podaci smešteni i načina na koji im se pristupa
- ❖ Danas je taj koncept potpuno generalizovan: OS omogućava procesima pristup do fajlova na isti način, bez obzira na to gde su ti fajlovi smešteni:
 - ❖ na uređaju (disku) ugrađenom u isti računar
 - ❖ na uređaju spolja priključenom na računar: spoljašnji disk, telefon, kamera, foto aparat ili drugi uređaj povezan na računar na različite načine
 - ❖ na udaljenom računaru koji je povezan mrežom

- ❖ na Internetu

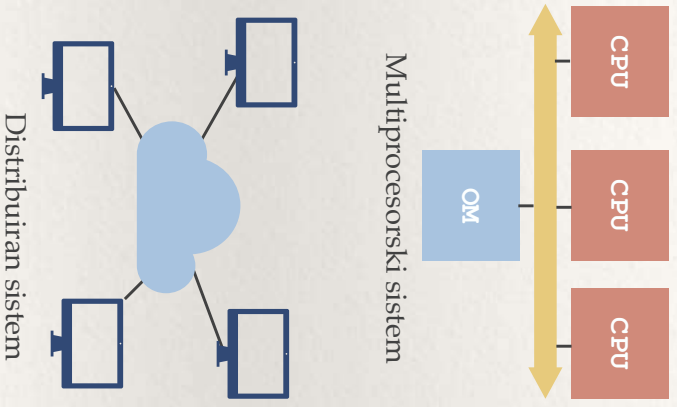
Mart 2020.

Copyright 2020 by Dragan Milićev

37

Arhitekturnalne paradigme

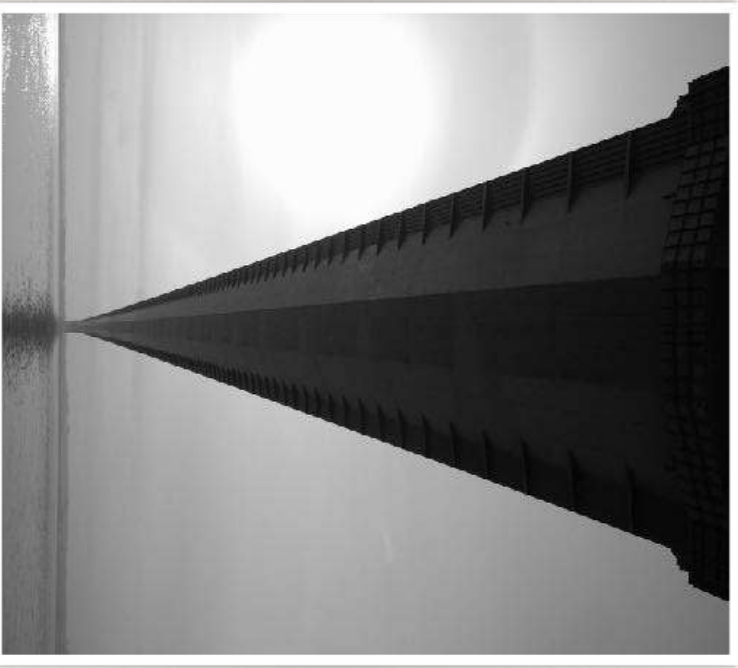
- ❖ Dve fundamentalne koncepcije, paradigme (hardverske) arhitekture računarskih sistema sa više procesora:
 - ❖ *Multiprocesorski sistem (multiprocessor)*: računarski sistem sa više procesora koji imaju zajedničku (deljenu) operativnu memoriju; procesori mogu da pristupaju deljenoj memoriji (čitaju iz nje ili u nju upisuju), npr. preko zajedničke magistrale na koju su svi povezani
 - ❖ *Distribuiran sistem (distributed system)*: sistem sa više procesora koji nemaju zajedničku operativnu memoriju (već svaki procesor ima svoju), a koji su povezani komunikacionom mrežom preko koje mogu razmenjivati poruke
 - ❖ Primeri distribuiranih sistema:
 - ❖ specijalizovan računar sa više procesora i brзом interkonekcijom mrežom između njih, obično neke specifične topologije (matrica, hiperkocka i slično)
 - ❖ lokalna računarska mreža (*local area network*, LAN): računarska mreža sa više povezanih računara na relativno malom prostoru i ograničenog pristupa (stan, zgrada, preduzeće, organizacija)
 - ❖ mreža šireg područja (*wide area network*, WAN): regionalna ili geografski distribuirana mreža
 - ❖ Internet: globalna, složena mreža najrazličitijih računara
- Mart 2020. Copyright 2020 by Dragan Milićev

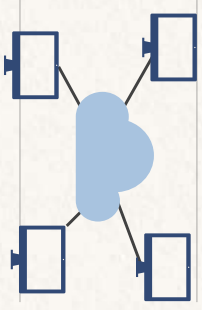


40

Glava 3: Vrste računarskih i operativnih sistema

- ❖ Arhitekturnalne paradigme
- ❖ Multiprocesorski sistemi
- ❖ Distribuiran sistemi
- ❖ Personalni računari
- ❖ Serverski sistemi
- ❖ Sistemi u oblaku
- ❖ Ugrađeni i sistemi za rad u realnom vremenu





Distribuiran sistemi

- ❖ Svi današnji operativni sistemi opšte namene podržavaju umrežavanje računara:
 - ❖ imaju module koji implementiraju komunikacione protokole (TCP/IP i druge)
 - ❖ omogućavaju (sistemske pozive za) razmenu poruka sa procesima na drugim računarima
 - ❖ omogućavaju pristup fajlovima koji se nalaze na uređajima drugih računara u mreži
 - ❖ omogućavaju udaljen pristup korisnicima (ne direktno preko tastature i monitora priključenog na računar):
 - ❖ *ssh* protokol (*secure shell*): protokol za kriptovanu komunikaciju sa drugim računarom na kom korisnik izvršava klijentski proces koji izvršava školjku sa interpreterom komandne linije, ali tako što taj interpreter pristupa udaljenom, serverskom računaru i stvara utisak neposrednog rada na njegovoj konzoli
 - ❖ *udaljena radna površina* (*remote desktop*): omogućava izvršavanje školjke sa grafičkim korisničkim interfejsom koja pristupa udaljenom, serverskom računaru i stvara utisak neposrednog rada na njegovom grafičkom interfejsu
- ❖ *Distribuiran operativni sistem* (*distributed OS*) je operativni sistem koji na skupu umreženih računara, tzv. klasteru (*cluster*), stvara utisak jedinstvenog prostora računarskih resursa, odnosno jedinstvenog "virtuelnog računara", i sakriva postojanje različitih računara: raspoređuje procese na procesore tih računara, fajlove raspoređuje po uređajima na tim računarima, ali stvara privid da su u istoj hijerarhiji direktorijuma, svim procesima stavlja na raspolaganje sve dostupne uređaje itd.

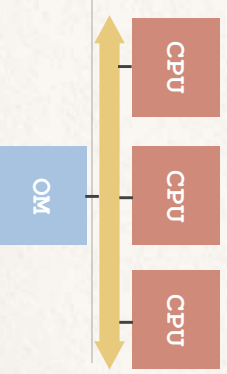
- ❖ Ovaj kurs se dalje bavi klasičnim operativnim sistemima koji upravljaju jednim računarom (ne distribuiranim OS)

Mart 2020.

Copyright 2020 by Dragan Milićev

42

Multiprocesorski sistemi



- ❖ Multiprocesorski sistem može biti, na hardverskom nivou:

- ❖ simetričan: svi procesori su opšte namene, jednaki, imaju isto vreme pristupa operativnoj memoriji

- ❖ asimetričan: neki procesori su specijalizovani za posebne namene (npr. *floating point unit* - *FPU*, *graphical processing unit* - *GPU*), ili imaju različito vreme pristupa operativnoj memoriji (*non-uniform memory access*, *NUMA*)

- ❖ Svi današnji računari su multiprocesorski, sa više procesora opšte namene i često koprocесорима za specijalne namene

- ❖ Različiti hardverski koncepti, poput *jezgra* (*core*) ili *niti* (*thread*), su hardverski elementi koji omogućavaju paralelno procesiranje, ali dele određene elemente procesora na različite načine (npr. keš memorije, podsistem za pristup magistrali i slično). OS ih u principu vidi kao *logičke procesore*

- ❖ OS za multiprocesorski sistem može biti:

- ❖ simetričan: svi procesori su ravnopravni, u smislu da svi mogu izvršavati i kod procesa i kod kernela
- ❖ asimetričan: jedan procesor je "gazda" (*master*) i on izvršava raspoređivanje procesa na druge procesore, kao i kernel kod za ostale sistemske usluge; ostali procesori su "robovi" (*slave*) i samo izvršavaju kod korisničkih procesa u skladu sa onim što im master dodeli

- ❖ OS za multiprocesorski sistem treba dodatno da obavlja sledeće radnje:

- ❖ raspoređuje procese na procesore, kako bi ih efikasno koristio; na primer, može da vodi računa o tzv. *afinitetu* (*affinity*): ako se proces u jednom naletu izvršava na jednom, a u narednom na drugom procesoru, generiše puno promašaja u procesorskom kešu, što je manje efikasno nego da nastavi na istom procesoru na kom se prethodno izvršavao - proces ima svojoj "afinitet" prema tom procesoru (bilo bi dobro, ali ne mora baš na njemu da se izvršava)
- ❖ rešava problem uporednog pristupa deljenim strukturama podataka kernela u zajedničkoj memoriji od strane instrukcija koje izvršavaju različiti procesori

Mart 2020.

Copyright 2020 by Dragan Milićev

41

Serverski sistemi

- ❖ *Serverski računar (server)* je računar namenjen za opsluživanje zahteva koji stižu komunikacionim protokolima preko računarske mreže sa udaljenih računara - *klijenata (client)*
- ❖ Da bi opsluživao što više takvih zahteva, serverski računar najčešće ima velike hardverske kapacitete:
 - ❖ više procesora (8, 16 i više)
 - ❖ mnogo operativne memorije (128 GB, 256 GB, pa i 512 GB i 1 TB)
 - ❖ više diskova na posebnim uređajima, tzv. *skladiština (storage)* velikog ukupnog kapaciteta (desetine i stotine TB), povezanih u redundantne strukture visokih performansi i povećane pouzdanosti (tzv. *RAID strukture*) - detalji u OS2
 - ❖ mrežnu vezu visoke propusnosti
- ❖ OS za serverski sistem mora da obezbedi uporedno izvršavanje mnogo procesa, efikasno korišćenje resursa, kao i razne druge usluge potrebne za korisnike na klijentima
- ❖ Serverski sistem najčešće i nema neposredno priključenu konzolu (tastaturu i ekran), ili ako je ima, ona služi samo za administraciju sistema; mnogo češće se serverskom računaru pristupa preko udaljenih školjki (*ssh, remote desktop*)
- ❖ Korisnici pristupaju serveru preko procesa koji se izvršavaju na njihovim računarima - klijentima, a ti procesi komuniciraju sa serverskim procesima razmenom poruka (uporediti ovo sa konceptom mejnfrejm računara)

Mart 2020.

Copyright 2020 by Dragan Milićev

44

Personalni računari

- ❖ *Personalni računari (personal computer, PC)* su računari za ličnu upotrebu - desktop, laptop (*laptop*), tablet, pametni telefon itd.
- ❖ Pojavljuju se krajem 1970-ih, u širu upotrebu ulaze 1980-ih
- ❖ Prvi personalni i stari kućni računari (*home computer*) bili su monoprocesni, sada su svi multiprocesni
- ❖ Umesto efikasnosti korišćenja procesora kao jednog od primarnih ciljeva (procesori na ličnim računarima su izuzetno malo iskorišćeni, ali to nikoga ne brine), operativni sistemi za personalne računare imaju druge prioritete:
 - ❖ lakoća i pogodnost upotrebe
 - ❖ efikasno izvršavanje više uporednih procesa (aplikacija)
 - ❖ mogućnost umrežavanja, pristupa Internetu, pristupa udaljenim fajlovima
 - ❖ povezivanje sa priključenim uređajima i drugim računarima
 - ❖ zaštita privatnosti i podataka
 - ❖ pouzdanost skladištenja podataka
 - ❖ dostupnost aplikacija
 - ❖ što manja potrošnja energije (baterije)

Mart 2020.

Copyright 2020 by Dragan Milićev

43

Ugrađeni sistemi i sistemi za rad u realnom vremenu

- ❖ *Ugrađen (embedded)* sistem je sistem koji služi za nadzor i upravljanje određenog većeg inženjerskog (hardverskog) okruženja i koji ispunjava svoj cilj obradom informacija, ali pri čemu obrada informacija jeste samo sredstvo, a ne njegov primarni cilj
- ❖ Veliki deo ovakvih sistema spada u kategoriju tzv. *sistema za rad u realnom vremenu (real-time system, RT)*: sistem koji obrađuje informacije i čije korektno funkcionisanje ne zavisi samo od logičke korektnosti rezultata, nego i od njihove pravovremenosti - rezultat isporučen neblagovremeno ili uopšte neisporučen je isto tako loš kao i pogrešan rezultat
- ❖ Kategorije RT sistema:
 - ❖ "Tvrđi" (*hard*): RT sistemi za koje je apsolutni imperativ da odziv stigne u zadatom vremenskom roku (*deadline*), jer prekoračenje roka ili potpuno neisporučenje rezultata može da dovede do katastrofalnih posledica po živote i zdravlje ljudi, materijalna sredstva ili životnu okolinu. Primeri: sistem za kontrolu nuklearne elektrane, sistem za upravljanje vozom ili letom aviona itd.
 - ❖ "Mek'i" (*soft*): RT sistemi kod kojih su vremenski rokovi važni i treba da budu poštovani, ali se povremeno mogu i prekoračiti, sve dok performanse sistema (propusnost i kašnjenje) statistički ulaze u zadate okvire. Primeri: telefonska centrala, uređaj za kablovsku TV, sistem za prikupljanje podataka u industriji itd.

Mart 2020.

Copyright 2020 by Dragan Milićev

46

Sistemi u oblaku

- ❖ *Sistemi u oblaku (cloud)* su distribuirani sistemi sa mnogo (stotine i hiljade) povezanih serverskih računara u jednom računskom centru (*data center*) ili regionalno ili globalno raspoređenim računskim centrima koji obebeđuju različite usluge korisnicima; usluge su dostupne preko Interneta, a o platformi brine pružalac usluge
- ❖ Sistemima u oblaku upravljaju posebni slojevi softvera, specifični distribuirani operativni sistemi (jer i oni upravljaju resursima) koji, korišćenjem usluga operativnog sistema na svakom serverskom računaru:
 - ❖ pružaju utisak jedinstvenog prostora resursa
 - ❖ raspoređuju procese i podatke na računare i diskove
 - ❖ obebeđuju skalabilnost: povećanje propusnosti u skladu sa potrebama korisnika i njihovog softvera, bez logičkih ograničenja
 - ❖ obebeđuju pouzdanost, tj. otpornost na otkaze (jednog ili više računara, diskova, mreže itd.)
- ❖ Više o ovome u predmetu OS2

Mart 2020.

Copyright 2020 by Dragan Milićev

45

dr Dragan Milićev

redovni profesor

Elektrotehnički fakultet u Beogradu

dmilicev@etf.rs, www.rcub.bg.ac.rs/~dmilicev

Operativni sistemi

Osnovni kurs

Slajdovi za predavanja

Deo II - Upravljanje memorijom

Za izbor slajda drži miša
uz levu ivicu ekrana

Mart 2020.

Copyright 2020 by Dragan Milićev

1

Ugrađeni sistemi i sistemi za rad u realnom vremenu

- ❖ Karakteristike tvrdih RT sistema i operativnih sistema za njih:
 - ❖ ne smeju da postoje hardverske i softverske komponente koje bi unosile neodređenost u vremenske karakteristike (kašnjenja, vreme odziva), kao što su diskovi, virtuelna memorija itd.
 - ❖ nema raspodele vremena
 - ❖ procesi su najčešće isključivo periodični ili sporadični
 - ❖ raspoređivanje je veoma strogo i karakteristično za ove sisteme (po prioritetima ili po vremenskom roku), ne primenjuju se algoritmi raspoređivanja kao u operativnim sistemima opšte namene
- OS opšte namene nemaju karakteristike potrebne za tvrde RT sisteme - u ovim sistemima koriste se specijalizovani, posebno projektovani RT operativni sistemi
- ❖ Karakteristike mekih RT OS:
 - ❖ mora postojati podrška za raspoređivanje po prioritetima i kontrolu vremenskih rokova
 - ❖ potrebne su sistemske usluge vezane za realno vreme (vremensko ograničenje, merenje protoka vremena, periodičnu aktivaciju procesa itd.)
- Mnogi moderni OS imaju ovakvu podršku i mogu se koristiti (i koriste se) u mekim RT sistemima (npr. Linux)
- ❖ Detalji u predmetu "Programiranje u realnom vremenu" / "Softver za rad u realnom vremenu" na master studijama

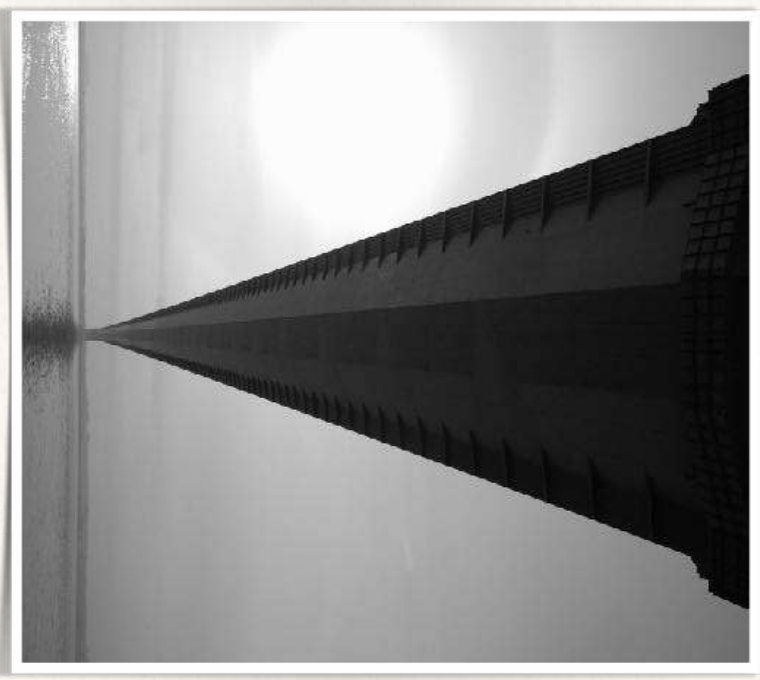
Mart 2020.

Copyright 2020 by Dragan Milićev

47

Glava 4: Adresiranje memorije

- ❖ Arhitektura procesora
- ❖ Operativna memorija
- ❖ Asembler
- ❖ Adresiranje podataka
- ❖ Adresiranje instrukcija
- ❖ Prevođenje
- ❖ Povezivanje



Mart 2020.

Copyright 2020 by Dragan Milićev

3

Deo II: Upravljanje memorijom

Mart 2020.

Copyright 2020 by Dragan Milićev

2

Arhitektura procesora

- ❖ *Programski dostupni registri* procesora:
 - Eksplicitno ili implicitno se referenciraju u instrukciji: instrukcija čita vrednost iz registra ili u njega upisuje vrednost
 - Semantika izvršavanja instrukcija garantuje “održavanje vrednosti” registra između dve izvršene instrukcije: ako je jedna instrukcija upisala vrednost u neki od ovih registara, prva sledeća izvršena instrukcija koja čita iz tog registra pročitace baš tu upisanu vrednost
 - Semantika svake instrukcije ili strogo definiše način na koji utiče na neki programski dostupan registar, ili garantuje da na njega ne utiče - nema nepredvidivih efekata
- ❖ Osim programski dostupnih, procesor poseduje mnogo internih registara koje koristi za implementaciju i koji nisu dostupni u instrukcijama: instrukcija ne može koristiti njihove vrednosti niti uticati na njih, a semantika instrukcije ih ne uključuje u svoju definiciju - za softver, oni kao i da ne postoje (potpuno su nevidljivi, transparentni)
- ❖ *Registri posebne namene*: imaju unapred definisanu i rezervisanu namenu i obično se ne pominju eksplicitno u instrukciji (već instrukcija podrazumeva njihovo korišćenje), npr. *PC, SP, PSW*
- ❖ *Registri opšte namene*: mogu se koristiti za čuvanje vrednosti koje se tumače kao *podaci* (operandi ili rezultati operacija) ili *adrese* lokacija u memoriji u kojima se nalaze podaci ili instrukcije

Mart 2020.

Copyright 2020 by Dragan Milićev

5

Arhitektura procesora

- ❖ *Instrukcija*: binarni zapis određene veličine (koja može da zavisi od vrste instrukcije) koji definiše elementarni zadatak za procesor (operaciju), a koji se može sastojati iz nekih od sledećih postupaka:

- Čitanja *operanda*: vrednosti *programski dostupnih registara* i /ili lokacija *operative memorije*
- Aritmetičke ili logičke *operacije* nad pročitanim vrednostima
- Upis *rezultata* - pročitane ili izračunate vrednosti u programski dostupne registre i /ili lokacije *operative memorije*
- Određivanja lokacije sledeće instrukcije

31	23	20	15	10	5	2
Op code	Addr mode	Reg 0	Reg 1	Reg 2	Type	Unused

- ❖ *Način adresiranja (address mode)*: specifikacija načina na koji se određuje mesto operanda / rezultata (registar ili lokacija u memoriji) ili adrese sledeće instrukcije (u memoriji)

Mart 2020.

Copyright 2020 by Dragan Milićev

4

Arhitektura procesora

- ❖ Fon Nojmanova arhitektura računara: procesor dohvata instrukcije iz operativne memorije i izvršava ih, jednu po jednu
- ❖ Programski brojač (*program counter*, *PC*) je specijalizovani programski dostupan registar procesora čija se vrednost koristi kao adresa memorijske lokacije čiji se sadržaj dohvata i tumači kao (deo) instrukcije koja je naredna za izvršavanje
- ❖ Naredna instrukcija je podrazumevano na memorijskoj lokaciji koja sledi odmah iza poslednje lokacije koju zauzima tekuća instrukcija - odmah po dohvatanju tekuće instrukcije, *PC* se podrazumevano inkrementira za veličinu te instrukcije, ali i sama instrukcija može definisati novu vrednost *PC* (*instrukcije skoka*)
- ❖ *picoRISC* (pRISC): izmišljeni, jednostavan RISC procesor sa 32-bitnom *load/store* arhitekturom koji se koristi za ilustraciju u primerima

Mart 2020.

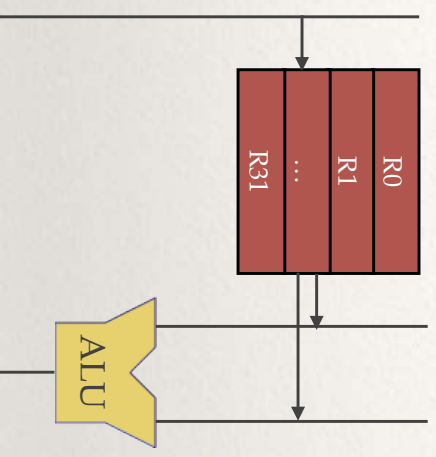
Copyright 2020 by Dragan Milićev



7

Arhitektura procesora

- ❖ *CISC procesori* često uvode ograničenja ili pravila korišćenja registara opšte namene:
 - Registri za podatke (*data registers*): registri koji se isključivo ili dominantno koriste za smeštanje operanada i rezultata aritmetičko-logičkih operacija
 - Registri za adrese (*address registers*): registri koji se isključivo ili dominantno koriste za smeštanje adresa memorijskih lokacija u kojima su operanadi ili instrukcije
- ❖ *RISC procesori* imaju jednostavnu, pravilnu, ortogonalnu arhitekturu:
 - nema ograničenja u načinu korišćenja registara opšte namene (svi se mogu koristiti i za adrese i za podatke) - *registrski fajl* (*register file*)
 - svi registri se mogu koristiti u svim dozvoljenim načinima adresiranja i u svim instrukcijama koji dozvoljavaju adresiranje registara (nema ograničenja ni razlike između registara)
 - *load/store arhitektura*: samo instrukcije prenosa podataka iz memorije u registar (*load*) ili iz registra u memoriju (*store*) mogu adresirati operativnu memoriju za izvorišni/odredišni operand; sve ostale instrukcije operišu isključivo nad programski dostupnim registrima



Staza za podatke (*data path*)

Mart 2020.

Copyright 2020 by Dragan Milićev

6

Arhitektura procesora

- ❖ *Programska statusna reč (program status word, PSW)*: specijalizovan programski dostupan registar procesora čiji se (neki) biti najčešće postavljaju implicitno, kao bočni efekti instrukcija, i služe kao indikatori statusa izvršene operacije ili rezultata
- ❖ Primeri: Z (*zero flag*) - postavlja se ako je rezultat jednak 0, C (*carry flag*) - postavlja se ako je u aritmetičkoj operaciji bilo prenosa, N (*negative flag*) - postavlja se ako je rezultat aritmetičke operacije negativan, V (*overflow flag*) - postavlja se ako je u aritmetičkoj operaciji bilo prekoračenja i slično

- ❖ Instrukcije uslovnog skoka kao uslov mogu da koriste vrednost nekog bita iz PSW

```
cmp r1, r2
jnz 0xA1
...
```

Instrukcija `cmp` (*compare*) oduzima vrednosti dva operanda i rezultat ne smešta nigde, ali implicitno postavlja bit Z (*zero flag*) u PSW na 1 ako je rezultat oduzimanja jednak 0 (vrednosti su jednake), u suprotnom ga postavlja na 0

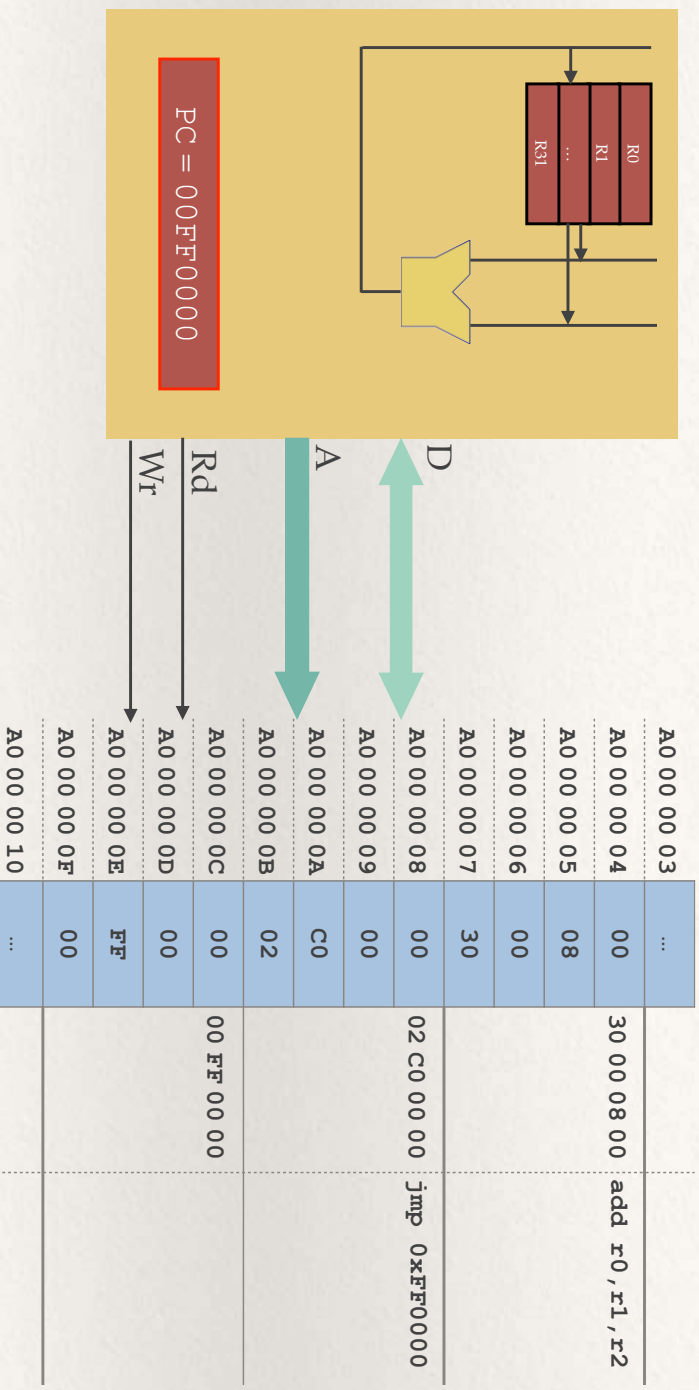
Instrukcija `jnz` (*jump on not zero*) ili, alternativno, `jne` (*jump on not equal*) vrši skok (upisuje adresu odredišta skoka u PC) ako je indikator Z u PSW postavljen na 1 (rezultat poslednje operacije koja ga je postavila je bio 0)

Mart 2020.

Copyright 2020 by Dragan Milićev

9

Arhitektura procesora



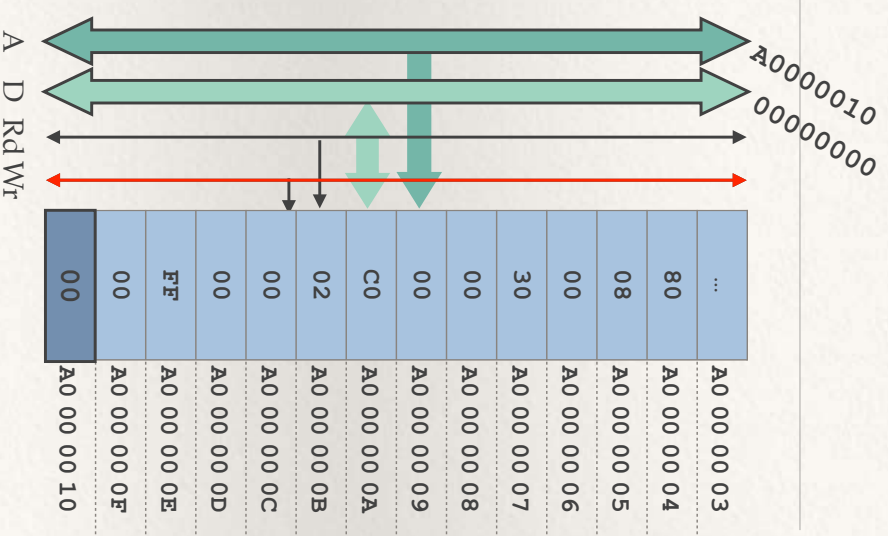
Mart 2020.

Copyright 2020 by Dragan Milićev

8

Operativna memorija

- ❖ *Operativna memorija* (*operating memory*, OM): linearno uređen skup ćelija sa pridruženim *adresama* iz skupa $0..2^n-1$, gde je n širina adrese u bitima (danas najčešće 32 ili 64, nekada 16, 20 ili neki drugi broj)
- ❖ U odnosu na registre procesora, značajno veći kapacitet (danas reda nekoliko desetina ili stotina GB), ali i duže vreme odziva (reda desetina ns)
- ❖ Procesor pristupa memoriji *ciklusima čitanja ili upisa* (*read/write memory cycle*), zadajući adresu, podatak (za ciklus upisa) i tip ciklusa (Rd / Wr)
- ❖ *Direktan pristup*: sukcesivni pristupi memoriji mogu biti bilo kojim lokacijama, u proizvoljnom redosledu
- ❖ *Adresibilna jedinica* (*addressible unit*): širina najmanje jedinice (memorijske ćelije) koja se može adresirati; danas najčešće bajt, ali je u prošlosti bilo i nešto drugo (16, 20, 40... bita)
- ❖ U zavisnosti od arhitekture, u jednom pristupu može se preneti sadržaj jedne ili više susednih adresibilnih jedinica, počev od adrese koja mora ili ne mora biti "poravnata" (vrednost koja daje ostatak 0 pri deljenju sa 2 ili 4)



Mart 2020.

Copyright 2020 by Dragan Milićev

11

Arhitektura procesora

- ❖ Umesto binarnog zapisa mašinskog programa, ljudski čitljiv, ekvivalentan zapis koji mu odgovara jedan-na-jedan je zapis na *simboličkom mašinskom jeziku* (*symbolic machine language*) ili *asembleru* (*assembly*): jedna asemblerka instrukcija odgovara jednoj mašinskoj instrukciji i obratno, s tim što asemblerki zapis koristi *mnemonike* (simboličke oznake ili skraćenice) za mašinske instrukcije, registre itd.

- ❖ Aritmetičko-logičke instrukcije: *add*, *subtract*, *compare*, *multiply*, *divide*, *and*, *or*, *not*, *xor*...

add r0, r1, r3
R0 := R1 + R3

31	23	20	15	10	5	
Op code	Addr mode	Reg dst	Reg src1	Reg src2	Unused	
0x30	0b000	0b00000	0b00001	0b00011	Unused	

- ❖ Instrukcije prenosa podataka: *load* (iz memorije u registar) i *store* (iz registra u memoriju), ili *move* (sa jednog na drugo mesto)...

- ❖ Instrukcije skokova - u PC smeštaju vrednost određene adrese skoka ukoliko je uslov ispunjen (inače PC zadržava vrednost koja ukazuje na lokaciju iza tekuće):

jmp 0xFF123456
bezuslovni skok: jump

Op code	Addr mode	Unused
0x02	0b110	
Destination address 0xFF123456		

- ❖ uslovni skokovi: *jump on zero* (ili *jump on equal*, znači isto), *jump on not zero* (ili *jump on not equal*), *jump on negative*, *jump on not negative*, *jump on overflow*, *jump on not overflow*...

- ❖ Mnoge ostale, specijalizovane instrukcije

Mart 2020.

Copyright 2020 by Dragan Milićev

10

Operativna memorija

- ❖ Različite oblasti (kontinualni i disjunktni delovi) fizičke memorije i fizičkog adresnog prostora mogu biti "pokrivene" različitim tipovima fizičke memorije ili ulazno-izlaznih uređaja:
 - ❖ Neperzistentni, nepostojani (*non-persistent, volatile*) RAM (*Dynamic / Random Access Memory, DRAM*): brza memorija sa mogućnošću i čitanja i upisa, ali gubi sadržaj gubitkom napajanja; za smeštanje kernela i procesa
 - ❖ Perzistentni, postojani (*persistent, non-volatile*) RAM: memorija sa mogućnošću i čitanja i upisa, ne gubi sadržaj gubitkom napajanja (npr. baterijski podržana ili fleš memorija), ali je sporija za čitanje nego DRAM; za smeštanje sistemskih parametara konfiguracije računara, npr. koji uređaj (disk) služi za podizanje OS (*boot disk*)
 - ❖ ROM (*Read-Only Memory*): perzistentna memorija bez mogućnosti upisa (samo čitanje); sadržaj upisan posebnim tehnološkim postupkom u samoj proizvodnji; u ovom delu su:
 - ❖ *bootstrap program*: program počev od fiksne, unapred definisane adrese za dati procesor (ili od adrese upisane na fiksnoj lokaciji, tzv. *reset pointer*) od koje procesor počinje dohvatanje i izvršavanje instrukcija po uključivanju; taj program ima zadatak da sa *boot* diska u memoriju učita unapred definisan blok (tipično blok broj 0) u kome se nalazi veći *bootstrap* program koji dalje sa tog diska učitava OS, inicijalizuje ga i stavlja ga u funkciju - podizanje sistema (*booting*)
 - ❖ BIOS (*Basic Input/Output System*): skup fiksnih, predefinisanih i ugrađenih rutina (procedura) koje obavljaju bazične, elementarne operacije sa delovima hardvera koji su uvek prisutni za datu arhitekturu računara i koje koristi OS
 - ❖ *Memorijski preslikani ulazno-izlazni uređaji (memory mapped I/O)*: različiti ulazno/ izlazni uređaji, odnosno njihovi registri ili memorijski moduli (npr. memorija grafičke kartice, registri uređaja itd.), kojima se pristupa preko određenih adresa raspoređenih po adresnom prostoru; procesor im pristupa isto kao i operativnoj memoriji

Mart 2020.

Copyright 2020 by Dragan Milićev

13

Operativna memorija

- ❖ Fizički adresni prostor (*physical address space*): skup svih adresa koje se mogu adresirati na adresnoj magistrali, u opsegu $0..2^n-1$ ($n = 32, 64, 16, 20\dots$)
- ❖ Instalirana fizička memorija: podskup fizičkog adresnog prostora (ne obavezno kontinualan) za koje stvarno postoje instalirani memorijski moduli (čipovi) ili drugi hardverski elementi i koji se mogu adresirati; adresiranje ostalih adresa generiše hardversku grešku; npr. za $n = 64$, adresibilna jedinica bajt, fizički adresni prostor je $2^{64} \text{ B} = 16 \text{ EB}$ (eksabajta, $1 \text{ E} = 2^{60}$), a instalirana memorija računara je po pravilu mnogo, mnogo manja (npr. 4 GB, 8 GB, 16 GB, 256 GB, 512GB, 1 TB - terabajt, $1 \text{ T} = 2^{40}$)

Mart 2020.

Copyright 2020 by Dragan Milićev

12

Operativna memorija

Primer: mapa fizičkog adresnog prostora neke izmišljene, jednostavne arhitekture



Mart 2020.

Copyright 2020 by Dragan Milićev

Operativna memorija

Primer: mapa fizičkog adresnog prostora neke izmišljene, jednostavne arhitekture



Mart 2020.

Copyright 2020 by Dragan Milićev

Asembler

- ❖ *Asembler (assembly)* je program koji prevodi sadržaj ulaznog, tekstualnog fajla sa kodom na simboličkom mašinskom jeziku u izlazni fajl sa binarnim zapisom mašinskih instrukcija i podataka
- ❖ Asembler prevodi jednu po jednu liniju teksta iz ulaznog fajla
- ❖ Prevodeći jednu po jednu liniju, asembler izračunava *tekuću adresu* za tu liniju, počev od neke unapred definisane vrednosti (0 ili neka druga vrednost)
- ❖ Ukoliko linija sadrži instrukciju (prepozna je se po mnemoniku instrukcije), asembler:
 - ❖ generiše binarni kod za tu mašinsku instrukciju na lokaciji tekuće adrese, uzimajući u obzir operande instrukcije, po potrebi koristeći vrednost tekuće adrese linije
 - ❖ uvećava tekuću adresu linije za veličinu binarnog zapisa prevedene instrukcije

Mart 2020.

Copyright 2020 by Dragan Milićev

17

```
; for (int i=0; i<n; i++) a[i]++
xor r1,r1,r1 ; i = 0
load r2,n
loop: cmp r1,r2 ; while (i<n)
      jnlt end_loop
      load r3,[r1+a] ; a[i]++
      inc r3
      store r3,[r1+a]
      inc r1 ; i++
      jmp loop
end_loop:
```

\$ je ovde simbol za tekuću adresu linije

Operativna memorija

Primer: mapa fizičkog adresnog prostora za 32-bitnu arhitekturu Intel IA32 (donekle pojednostavljeno)

4 GB	
00 10 00 00	RAM (upper parts reserved) 4 GB – 1 MB
00 0F 00 00	
00 0E 00 00	
00 0C 00 00	
00 0A 00 00	
1 MB	
00 00 00 00	RAM (DOS area) 640 KB

Mart 2020.

Copyright 2020 by Dragan Milićev

16

Assembler

- ❖ *Labela (label)* je identifikator (simbol) pridružen jednoj liniji asemblerskog koda. Assembler izgrađuje tabelu definisanih simbola (uključujući i labele). Svakoj labeli na koju naiđe pridružuje vrednost tekuće adrese te linije (ovde označena kao predefinisani simbol \$). Kada u nekom konstantnom izrazu naiđe na labelu ili drugi simbol, assembler je zamenjuje pridruženom brojnomoj vrednošću, na primer, kao određite instrukcije skoka:

```
Labela
xor    r1,r1,r1 ; i = 0
load  r2,n
loop: cmp r1,r2 ; while (i<n)
      jnlt end_loop
      ...
      jmp  loop
end_loop:
```

- ❖ Direktiva *org* koja eksplicitno podešava tekuću adresu linije - uzrokuje promenu adrese od koje se nastavlja (ili započinje) dalje generisanje koda; simbol \$ označava tekuću adresu, pre ove promene:

```
ORG constant-expression ; comment
org 0x8000

page def 0x1000 ; one 4K page size
org $-$%page+page ; align to the next page
```

- ❖ Direktiva *start* koja označava adresu početne instrukcije programa; ovu informaciju koja je na određeni način zapisana u izlaznom fajlu koristi OS kada pokreće proces nad ovim programom:

```
START constant-expression ; comment

start main
main: ...
Mart 2020.
```

Copyright 2020 by Dragan Milićev

19

Assembler

- ❖ *Direktiva (directive)* je linija asemblerskog teksta koja ne sadrži instrukciju, već neku drugu specifikaciju ili uputstvo assembleru

- ❖ Direktiva *def* koja definiše *simboličku konstantu (symbolic constant)*: definisanoj simboličkoj konstanti pridružuje vrednost konstantnog izraza naveden u direktivi; kada se u ostatku programa upotrebi ova simbolička konstanta, assembler je zamenjuje pridruženom brojnomoj vrednošću:

- ❖ program je razumljiviji, jer je lakše razumeti značenje simbola nego konkretne vrednosti
- ❖ program je lakši za održavanje i manje podložan greškama, jer se promena vrednosti obavlja na jednom mestu (a zamene u celom kodu obavlja assembler)

```
symbol DEF constant-expression ; comment

mask    def 0x80
qmark   def '?'
step    def 4
dstep   def step*2
```

Ova direktiva nema nikakvog efekta na generisani binarni zapis ili na vrednost tekuće adrese. Assembler jednostavno ubacuje nov simbol u svoju internu tabelu definisanih simbola

- ❖ *Konstantan izraz (constant expression)* je izraz koji assembler izračunava za vreme provođenja i sastoji se samo od (prethodno definisanih) simboličkih konstanti ili literala (brojne i znakovne konstante) i aritmetičkih i logičkih operacija sa njima

Mart 2020.

Copyright 2020 by Dragan Milićev

18

Adresiranje podataka

- ❖ Konstante koje se koriste u programu kao operandi operacija u mašinskim instrukcijama koriste se kao operandi specifikovani *neposrednim načinom adresiranja* (*immediate address mode*): operand je binarni sadržaj u odgovarajućem polju same instrukcije (samo za izvorište):

#constant-expression

označava neposredno adresiranje, a operand je zadat kao konstantan izraz (tipično literal ili simbolička konstanta)

load r1, #1

U nekim assemblerima za ovo se koristi notacija \$const ili samo const

and r0, #mask

; a >>= 4

load r0, a

load r1, #4

ssr r0, r0, r1 ; ssr - signed shift right

store r0, a

31	23	20	15	5	2
Op code	Addr mode	Reg dst	Unused	Type	Unused
0x20	0b100				
Immediate operand					

Mart 2020.

Copyright 2020 by Dragan Milićev

21

Assembler

- ❖ Direktive za definisanje podataka: za svaki navedeni specifikator jednog podatka, odvaја se prostor u generisanom binarnom zapisu za smeštаnje jednog podatka navedenog tipа (байt, dva байta, ...), na tekućој adresi, i u taj prostor upisuje binarni zapis vrednosti inicijalizatora koji je zadat konstantnim izrazom; tekuća adresa se uvećаva za veličinu aloциranih podataka:

label: DB|DW|DD data-spec, ... ; comment

hello: db 'H', 'e', 'l', 'l', 'o', '\0'

lookup: dw 16 dup 0

p: dd 0

16 puta (dup - duplicate) 16-bitna reč (dw - define word) inicijalizovana nulom

4 байта (dd - define double word) - 32-bitna reč

Mart 2020.

Copyright 2020 by Dragan Milićev

20

Adresiranje podataka

- ❖ Za indirektan pristup preko pokazivača koristi se *registarsko indirektno adresiranje* (*register indirect address mode*): operand je u memoriji, na lokaciji čija je adresa zadata vrednošću registra koji je specifikovan u određenom polju instrukcije:

[reg-mnemonic]

```
; *p += 4
load r1, [r0]
load r2, #4
add r1, r1, r2
store r1, [r0]
```

p je pokazivač tipa *int** čija je vrednost učitana u *r0*

- ❖ Pristup elementu niza:

```
; p[i]++
load r3, #2
shl r3, r2, r3 ; r3 = r2<<2 (offset=i*sizeof(int))
add r3, r1, r3
load r0, [r3]
inc r0
store r0, [r3]
```

p je pokazivač tipa *int** koji ukazuje na element niza tipa *int[]* i čija je vrednost učitana u *r1*, a *i* je promenljiva tipa *int* čija je vrednost učitana u *r2*

Mart 2020.

Copyright 2020 by Dragan Milićev

23

Adresiranje podataka

- ❖ Da bi se sukcesivne aritmetičko-logičke operacije u jednom izrazu nad programskim promenljivim uradile efikasnije (a kod RISC procesora sa *load/store* arhitekturom to je i jedini način), vrednosti tih promenljivih se iz memorije učitavaju u registre nad kojima se onda vrše operacije

- ❖ U tim slučajevima, operand se zadaje *registarskim direktnim adresiranjem* (*register direct address mode*): operand je u registru koji je specifikovan u odgovarajućem polju instrukcije:

reg-mnemonic

```
; a = (++b + c) * d
```

```
load r0, b
inc r0
store r0, b
load r1, c
add r0, r0, r1
load r1, d
mul r0, r0, r1
store r0, a
```

U nekim assemblerima za ovo se koristi notacija *%reg*

Mart 2020.

Copyright 2020 by Dragan Milićev

22

Adresiranje podataka

- ❖ *Statički podaci programa*: na jezicima C/C++ i svim drugim srodnim jezicima, semantika statičkog podatka je takva da za vreme izvršavanja programa uvek postoji jedna i samo jedna instanca podatka za svaku definiciju statičkog podatka
- ❖ Zbog toga prostor za smeštanje takvog podatka može da se obezbedi statički, za vreme prevođenja programa (odnosno pisanja asemblerskog programa), unutar samog binarnog zapisa izvršnog fajla koji je izlaz prevodioca ili asemblera
- ❖ Ako je ovakav podatak u programu inicijalizovan konstantnim izrazom, tj. izrazom čija se vrednost može izračunati za vreme prevođenja, onda prevodilac generiše binarni zapis te inicijalne vrednosti u prostoru alociranom za podatak:

```
; static int a=0, b=-1;
a dd 0
b dd 0xffffffff
; static char c='0', *p=nullptr;
c db 0x30
p dd 0
```

- ❖ Pošto se statički podatak alocira za vreme prevođenja, prevodilac/ asembler poznaje njegovu adresu, pa se pristup do ovih podataka može izvršiti *memorijskim direktnim adresiranjem (memory direct address mode)*: operand je u memoriji, na lokaciji čija je adresa zadata u samoj instrukciji:

constant-expression	
Op code	0b11
memory address	

Vrednost izraza (tipično labela) koristi se kao adresa operanda

U nekim asemblerima se za ovo koristi notacija *[const-expression]*

Mart 2020.

Copyright 2020 by Dragan Milićev

25

Adresiranje podataka

- ❖ *Registarsko indirektno adresiranje sa pomerajem (register indirect address mode with displacement)*: operand je u memoriji, na lokaciji čija se adresa dobija sabiranjem sadržaja registra specifičnog u instrukciji i neposredne (tipično označene) konstante (pomeraja, može biti i negativan) definisane u instrukciji:

```
[reg-mnemonic +|- const-expression]
[const-expression +|- reg-mnemonic]
```

- ❖ Pristup elementu niza:

; a[i]--	
load r2, #2	
shl r2, r1, r2	; r2 = r1<<2 (offset=i*si
load r0, [atr2]	
dec r0	
store r0, [atr2]	

a je niz tipa *int[]* čija je adresa označena labelom *a*,
 $a+i$ je promenljiva tipa *int* čija je vrednost učitana u *r1*

Zapravo odslilkava pointersku aritmetiku jezika C:
a[i] je isto što i $*(a+i)$

- ❖ Pristup članovima strukture:

```
; struct S {int a, b, c;}
S::a def 0
S::b def 4
S::c def 8

; struct S* p;
; p->a = p->b
load r2, [r1+S::b]
store r2, [r1+S::a]
```

Članovi strukture definišu se kao simboličke konstante koje predstavljaju pomeraj (*offset*) u odnosu na adresu početka strukture

p je pokazivač tipa *S** čija je vrednost učitana u *r1*

Mart 2020.

Copyright 2020 by Dragan Milićev

24

Adresiranje podataka

- ❖ *Rekurzija*: u nekom trenutku izvršavanja programa može biti više aktiviranih, a nezavršenih aktivacija istog potprograma (poziva, izvršavanja, inkarnacija), i svaka od njih mora da poseduje svoj skup instanci lokalnih podataka koje odgovaraju istim definicijama tih podataka
- ❖ Primer jednostavnog rekurzivnog potprograma na jeziku C:

```
unsigned factorial (unsigned n) {
    if (n==0) return 1;
    else return n*factorial (n-1);
}
```
- ❖ Kada ne bi bilo rekurzije, lokalni podaci, uključujući i argumenti, mogli bi se alocirati statički (npr. FORTRAN). Kako ovo rešiti ako se dozvoljava rekurzija?
- ❖ Ideja:
 - ❖ Prilikom poziva potprograma, formirati nov komplet lokalnih podataka (uključujući i argumente), tzv. *aktivacioni blok*, i staviti ga na kraj liste
 - ❖ Potprogram treba da referencira one instance lokalnih podataka koje su na kraju liste - tekući aktivacioni blok je novi blok
 - ❖ Pri povratku iz potprograma, izbaciti komplet lokalnih podataka sa kraja liste - tekući aktivacioni blok postaje onaj koji je sada na kraju liste
- ❖ Struktura koja podržava ovaj protokol - stek (*stack*):
 - ❖ linearno organizovana struktura sa dve operacije
 - ❖ operacija *push* smešta datu vrednost na "vrh" steka (na kraj liste)
 - ❖ operacija pop uzima vrednost sa "vrha" steka (sa kraja liste) i izbacuje je sa steka (LIFO struktura)

Mart 2020.

Copyright 2020 by Dragan Milićev

27

Adresiranje podataka

- ❖ *Lokalni podaci potprograma* (automatski na jezicima C/C++), uključujući i argumente (ulazne i izlazne / povratne): na proceduralnim i OO jezicima, svaka aktivacija (poziv) potprograma ima svoj, nov skup instanci lokalnih podataka

- ❖ Da li se i lokalni podaci mogu alocirati statički, i time adresirati direktnim adresiranjem?

- ❖ Primer jednostavnog potprograma - funkcije na jeziku C:

```
int max (int a, int b) {
    return (a>=b) ?a:b;
}
```

- ❖ Moguća implementacija:

```
max::ret dd ?
max::a dd ?
max::b dd ?
```

Znak ? označava proizvoljnu vrednost, neinicijalizovan alociran podatak

```
max:      load r0,max::a
          load r1,max::b
```

```
          cmp r0,r1
```

```
          jnge L001
```

```
          store r0,max::ret
```

```
          ...
```

```
L001:      store r1,max::ret
```

```
          ...
```

- ❖ Poziv potprograma:

```
; z=max(x,y)
store r0,max::a
store r1,max::b
```

x je u r0, y u r1, a z u r2

Poziv potprograma

```
...      load r2,max::ret
```

Povratak iz potprograma

Mart 2020.

Copyright 2020 by Dragan Milićev

26

Adresiranje podataka

Izuzetak su one rekurzije u kojima se rekurzivni poziv nalazi na kraju tela potprograma, iza kog taj potprogram više ne koristi svoje lokalne podatke, tzv. *repne rekurzije* (*tail recursion*). Kod ovakvih rekurzija, sve aktivacije mogu da koriste isti aktivacioni blok (na steku ili statički), pa se rekurzija pretvara u petlju (iteraciju). Prevodioci i primenjuju ovu tehniku kao optimizaciju, tzv. eliminaciju repne rekurzije (*tail recursion elimination*)

- ❖ Implementacija lokalnih podataka i argumenata na steku:

- ❖ Stvarne argumente na stek mora da stavi pozivalac, i isto tako da ih sa steka i skine nakon povratka iz potprograma

```
int max (int a, int b) {  
    return (a>=b) ?a:b;  
}
```

- ❖ Alokaciju i inicijalizaciju lokalnih podataka potprograma na vrhu steka mora da obavi sam potprogram, instrukcijama stavljanja inicijalnih vrednosti na stek ili prosto pomeranjem vrha steka ako te vrednosti nisu zadate, i isto tako da ih skine sa vrha steka pre povratka

SP+12 →	b
SP+8 →	a
SP+4 →	ret addr
SP →	

- ❖ Povratne vrednosti funkcije, kao i izlazni argumenti (na jezicima koji taj koncept podržavaju):

- ❖ mogu da se prenose slično kao i ulazni argumenti: pozivalac alokira prostor za ove argumente na steku, ali ih ne inicijalizuje; potprogram upisuje te vrednosti pre povratka, a pozivalac ih skida sa steka nakon povratka iz potprograma

- ❖ ukoliko povratne vrednosti mogu da stanu u registar ili nekoliko registara, mogu se prenositi preko određenog registra, što se najčešće i radi

```
max:    load r0, [SP+8] ; a  
        load r1, [SP+12] ; b  
        cmp r0, r1  
        jge L001  
        load r0, r1  
        ... ; return
```

- ❖ U svakom slučaju, instrukcije potprograma moraju da adresiraju lokalne podatke *relativnim adresiranjem u odnosu na vrh steka* (npr. registarskim indirektnim adresiranjem preko registra SP sa pomerajem), pri čemu su pomeraji poznati i konstatni za dati lokalni podatak i datu poziciju unutar koda potprograma

Mart 2020.

Copyright 2020 by Dragan Milićev

Zadatak: pokazati kako izgleda
asemblerški kod za poziv:
z=max (a, b)

29

Adresiranje podataka

- ❖ Implementacija steka na nivou arhitekture procesora: stek se alokira u memoriji, a na vrh steka može ukazivati vrednost nekog od programski dostupnih registara procesora - *pokazivač vrha steka* (*stack pointer*):

- ❖ Stek može rasti ka višim ili nižim adresama memorije

- ❖ Vrednost registra može da ukazuje na poslednju zauzetu ili prvu slobodnu lokaciju steka

- ❖ Ako ne postoji posebna procesorska podška za stek, jedan od registara opšte namene izdvaja se da služi kao pokazivač vrha steka. Npr. neka je to R15:

- ❖ Operacija *push*, npr. jednog bajta iz R0:

```
inc r15  
storeb r0, [r15]
```

Šifiks *b* u mnemonicu instrukcije ukazuje na tip operanda (*b* - *byte*, samo jedan bajt)

- ❖ Operacija *pop*, npr. 32 bita u R0:

```
load r0, [r15]  
dec r15  
dec r15  
dec r15  
dec r15
```

- ❖ Posebna podška procesora: postoji poseban, specijalizovan programski dostupan registar koji služi kao pokazivač steka SP i posebne instrukcije *push* i *pop* koje implicitno koriste i menjaju vrednost SP (on se ne referencira eksplicitno u instrukciji); tada arhitektura procesora dikтира to da li stek raste ka višim ili nižim adresama i na koju lokaciju ukazuje vrednost SP:

```
push r0  
pop r0
```

Mart 2020.

Copyright 2020 by Dragan Milićev

28

Adresiranje instrukcija

- ❖ Instrukcije skoka mogu da adresiraju određenu instrukciju *memorijskim direktnim adresiranjem*: adresa određene instrukcije je data u odgovarajućem polju same instrukcije skoka
- ❖ Prevodilac / assembler određuje tu adresu kao *apsolutnu* (vrednost tekuće adrese pridružene određenoj labeli), za vreme prevođenja:

```
; for (int i=0; i<n; i++) a[i]++
xor  r1,r1,r1 ; i = 0
load r2,n
loop: cmp  r1,r2 ; while (i<n)
      jnlt endloop
      loadb r3,[r1+a] ; a[i]++
      inc  r3
      storeb r3,[r1+a]
      inc  r1 ; i++
      jmp  loop
endloop:
```

Assembler ovde može da generiše mašinsku instrukciju koja koristi memorijsko direktno adresiranje određište skoka, odnosno apsolutni skok, zapisujući vrednost adrese pridružene labeli u polje u instrukciji

- ❖ U tom slučaju, adresa upisana u instrukciji skoka zavisi od lokacije (adrese) smeštanja određiškog koda: ako se ta lokacija promeni, mora se promeniti i adresa u instrukciji

31	23	20
Op code	Addr mode	...
	0b110	
Memory address		

Mart 2020.

Copyright 2020 by Dragan Milićev

31

Adresiranje podataka

- ❖ Neka su negde definišani sledeći podaci, kao statički ili lokalni:

```
int a[5];
struct S* p = ...;
```

Ovi podaci biće alocirani u statičkoj memoriji ili na steku kao susedni (na susednim lokacijama), npr: baš u navedenom poretku

- ❖ U nekom drugom delu programa napravljena je sledeća greška:

```
for (int i=0; i<=5; i++) a[i] = x*i;
```

Ovaj kod biće preveden u mašinske instrukcije koje upisuju vrednosti u 6 susednih reči u memoriji, počev od adrese početka niza *a*, tako da će upisati i u prostor koji zauzima pokazivač *p*: njegova vrednost postaje potpuno pogrešna, nevalidna, *korumpirana* (*corrupted*)

- ❖ Šta se dešava ako se sada negde druge u programu ovaj pokazivač *p* koristi npr. ovako:

```
p->a = p->b;
```

Prevod ovog izraza:

```
load r1,...; load p to r1
load r2,[r1+8::b]
store r2,[r1+8::a]
```

Ishod ovih instrukcija čitanja i upisa je u opštem slučaju potpuno nepoznat: nepoznate su i adrese sa kojih se čita i upisuje, kao ni vrednosti u lokacijama na tim adresama.

Efekat može biti različit: upis neke vrednosti u prostor za podatke, koji onda propagira grešku dalje kroz program jer taj podatak postaje korumpiran, ili upis preko instrukcije, što dovodi do neregularnosti u daljem izvršavanju - potrebna je zaštita (*protection*) od ovakvih pojava!

Mart 2020.

Copyright 2020 by Dragan Milićev

30

Adresiranje instrukcija

- ❖ Poziv potprograma, tj. prelazak izvršavanja na prvu instrukciju potprograma može da se realizuje kao jednostavan bezuslovan skok na tu instrukciju, sa apsolutnim ili relativnim adresiranjem
- ❖ Povratak iz potprograma ne može se izvršiti kao skok na adresu koja je unapred poznata za vreme prevođenja, jer se potprogram može pozvati sa različitim mesta u programu, pa se mora uvek vraćati na odgovarajuće mesto, na prvu instrukciju koja sledi iza instrukcije poziva potprograma. Prema tome, adresa povratka mora se odrediti *dinamički*, za vreme izvršavanja, i to za svaki poziv
- ❖ Povratak iz potprograma se mora izvršiti *indirektnim skokom*, tj. skokom na adresu koja je na neki način izračunata ili pročitana iz memorije, a koju je definisao pozivalac
- ❖ Zato se povratna adresa može se smatrati specifičnim argumentom potprograma, jer ga zadaje pozivalac, a koristi pozvani potprogram, pa stoga može da se implementira na jedan od sledećih načina:
 - ❖ ako nije podržana rekurzija, povratnu adresu pozivalac može da upiše na određenu statički alociranu lokaciju u memoriji za tu namenu i taj potprogram (ili čak u registar), iz koje će je onda pozvani potprogram pročitati prilikom povratka
 - ❖ u opštijem slučaju, zbog podrške rekurziji, povratna adresa se prenosi preko procesorskog steka

Mart 2020.

Copyright 2020 by Dragan Milićev

33

Adresiranje instrukcija

- ❖ Procesori po pravilu podržavaju, a ponekad i zahtevaju za neke instrukcije (npr. uslovne skokove) *relativno adresiranje*, i to po pravilu u odnosu na tekuću vrednost registra *PC* (koji tokom izvršavanja instrukcije skoka ukazuje na lokaciju iza te instrukcije):

```
loop : ...      jmp    loop
```

Bez obzira na istu notaciju (navođenje labela), assembler može da generiše mašinski instrukciju koja koristi relativan skok, odnosno registarsko indirektno adresiranje sa pomerajem, tako da se adresa određišta skoka izračunava dodavanjem pomeraja na trenutnu vrednost *PC*: `jmp lpc+loop-($+8)`

- ❖ Instrukcija skoka sada koristi registarsko indirektno adresiranje sa pomerajem u odnosu na vrednost *PC* tokom izvršavanja ove instrukcije: određište skoka izračunava se kao zbir trenutne vrednosti *PC* i pomeraja iz instrukcije. Pomeraj se izračunava za vreme prevođenja i jednak je (simbol *&* označava dužinu tekuće instrukcije skoka, a *dest* označava apsolutnu adresu određišta skoka, npr. vrednost pridruženu labeli):

dest - (\$ + &)

- ❖ Na ovaj način mašinski kod generisan za jedan blok koda, npr. kontrolnu strukturu (*if-then-else*, petlja) ili ceo potprogram postaje *relokabilan*, tj. ne zavisi od lokacije na kojoj je smešten

Mart 2020.

Copyright 2020 by Dragan Milićev

32

Adresiranje instrukcija

❖ Primer rekurzivne funkcije:

```
unsigned factorial (unsigned n) {  
    if (n==0) return 1;  
    else return n*factorial(n-1);  
}
```

SP+8 →	n
SP+4 →	ret addr
SP →	

❖ Prevod na assembler:

```
factorial: load    r1, [SP+n_disp]    ; r1=n  
           xor     r2, r2, r2        ; r2=0  
           cmp     r1, r2            ; if (n==0)  
           jne     else              ;  
           load    r0, #1            ; return 1  
           ret     r1                ;  
           dec     r1                ; r1=n-1  
           push    r1                ;  
           call    factorial         ;  
           pop     r1                ; pop the argument, clean the stack  
           load    r1, [SP+n_disp]   ; r1=n  
           mul     r0, r1, r0        ; return n*fact(n-1)  
           ret     r1                ; pop pc
```

ret - instrukcija povratka iz potprograma (ili *rti* - *return from subroutine*)
call - instrukcija skoka u potprogram; radi isto što i *push pc, jmp factorial*

Mart 2020.

Copyright 2020 by Dragan Milićev

35

Adresiranje instrukcija

❖ Zato procesori po pravilu podržavaju posebnu instrukciju *poziva potprograma* koja obavlja principijelno dve radnje:

- ❖ stavlja tekuću vrednost *PC* (ukazuje na instrukciju iza tekuće) na vrh steka
 - ❖ u *PC* upisuje adresu potprograma, definisanu načinom adresiranja (apsolutno ili relativno), i time skače na prvu instrukciju potprograma
- ❖ Instrukcija *povratka* iz *potprograma* obavlja sledeće: sa vrha steka skida vrednost veličine registra *PC* i tu vrednost smešta u *PC*
- ❖ Kako se *PSW* implicitno menja, i to kao bočni efekat mnogih instrukcija, velika je šansa da će instrukcije potprograma promeniti ovaj registar. Zato je potrebno da potprogram sačuva (na steku) zatečene vrednosti *PSW* i restaurira je pre povratka. Kako se ovo ne bi radilo svaki put posebnim instrukcijama potprograma, mnogi procesori implicitno čuvaju, osim *PC*, i *PSW* na steku instrukcijom skoka u potprogram, a restauriraju ga sa steka instrukcijom povratka iz potprograma

Mart 2020.

Copyright 2020 by Dragan Milićev

34

Adresiranje instrukcija

- ❖ Slično može da se desi ako se kao “mašinski program”, zlonamerno ili greškom, podmetne bilo kakav binaran sadržaj ili pogrešna adresa početne instrukcije (na kojoj se uopšte ne nalazi regularna instrukcija)
- ❖ U principu, instrukcija u sebi može da referencira bilo koju raspoloživu adresu u memoriji, i to i na čitanje i na upis, ili da sa nje dohvata sadržaj koji će tumačiti kao instrukciju
- ❖ Sveukupno, zbog slučajne greške u programu (poput navedenih) ili zbog loše namere, program može imati potpuno neregularne pojave:
 - ❖ izvršavanje neregularne instrukcije
 - ❖ pristup proizvodnjim delovima memorije, pa čak i onim koje zauzima sam kernel ili drugi procesi
- ❖ Zato je potrebno da OS, uz podršku hardvera (bez koje to ne može da uradi) obezbedi najviši mogući stepen zaštite od ovakvih pojava - *zaštita (protection)*

Mart 2020.

Copyright 2020 by Dragan Milićev

37

Adresiranje instrukcija

- ❖ Funkcije mogu da se pozivaju i indirektno, preko pokazivača; ovakvi su uvek polimorfni pozivi virtuelnih funkcija članica na jeziku C++, kada se objektima pristupa preko pokazivača:

```
unsigned (*p) (unsigned) = factorial;
```

```
...  
(*p) (m) ;
```

Prevod poziva funkcije na assembler:

```
load    r0,m  
push    r0  
load    r0,p  
call    [r0]  
pop     r1
```

Registarsko indirektno adresiranje: sadržaj registra je adresa odredišta skoka

- ❖ Ukoliko vrednost pokazivača postane korumpirana (nevalidna), ili je on *null*, poziv će imati potpuno nedefinisan efekat: u PC će biti upisana vrednost pokazivača, procesor će potom dohvatati binarni sadržaj sa lokacije na koju ukazuje pokazivač i tumačiti taj sadržaj kao instrukciju
- ❖ Pošto je taj sadržaj proizvoljan i nepredvidiv, efekti su nepredvidivi: taj binarni sadržaj može biti neregularan zapis (nepostojeći kod operacije ili način adresiranja), ili može biti zapis neke proizvoljne, ali regularne instrukcije sa potpuno nepredvidivim efektom - procesor počinje da “luta” memorijskim prostorom i izvršava neodređene stvari

Mart 2020.

Copyright 2020 by Dragan Milićev

36

Prevođenje

- ❖ Prevodilac učitava znak po znak iz ulaznog fajla sa izvornim kodom programa; iako tekst programa ljudi doživljavaju dvodimenzionalno (u ravni), pri čemu im te dve dimenzije pomažu u čitanju i razumevanju (prelom redova, proredi i uvlačenje, odnosno “nazubljivanje” koda), prevodilac kod tumači isključivo sekvencijalno
- ❖ Prevodilac najpre učitanе znakove grupiše u veće celine, tzv. *leksičke elemente* ili *lekseme* (*lexical element, lexem*), ili *žetone* (*token*), u skladu sa pravilima jezika; ova faza prevođenja naziva se *leksička analiza* (*lexical analysis*); na primer, u sledećem delu koda, različitim bojama označene su različite lekseme:
if (i++ +j>=0 && i<this->size())
- ❖ U daljem postupku prevodilac tretira lekseme kao integralne celine, odnosno kao elemente od kojih su izgrađeni krupniji jezički iskazi, tj. rečenice
- ❖ Prevodilac tokom prevođenja prepoznaje te veće jezičke celine (rečenice) na osnovu *gramatike* (*grammar*) jezika; ova faza prevođenja naziva se *parsiranje* (*parsing*); u slučaju prestupa nekog pravila gramatike, prevodilac prijavljuje grešku u prevođenju
- ❖ Za prepoznate rečenice i elemente u njima, prevodilac proverava ostala pravila jezika, tzv. *semantička pravila* (*semantic rules*), i opet prijavljuje greške u slučaju prestupa
- ❖ Konačno, za one elemente rečenica za koje je to definisano semantikom jezika, prevodilac generiše sadržaj u prevedenom objektom fajlu u kome se principijelno nalazi:
 - binarni mašinski kod za mašinske (procesorske) instrukcije naredbi tela funkcija (potprograma)
 - alociran prostor za statičke objekte (podatke), tj. sa tzv. *statičkim trajanjem skladištenja* (*static storage duration*)

Mart 2020.

Copyright 2020 by Dragan Milićev

39

Prevođenje

- ❖ *Prevodilac* (*compiler*) je program koji prevodi tekstualni zapis izvornog programa na višem programskom jeziku (ovde se razmatra samo C ili C++) u binarni, mašinski zapis
- ❖ Program na jeziku C/C++ sastoji se od jednog ili više modula, pri čemu se modulom smatra sadržaj jednog fajla za izvornim kodom (tipična ekstenzija imena je *.c* ili *.cpp*)
- ❖ Svaki izvorni fajl je *odvojena jedinica prevođenja* (*compilation unit*): svaki fajl se prevodi odvojeno i nezavisno: kada prevodilac prevodi jedan fajl, ne izlazi iz granica tog fajla, odnosno ne tretira druge fajlove programa
- ❖ Kada prevodi jedan fajl sa izvornim kodom (*.cpp*), prevodilac će generisati jedan fajl sa prevedenim kodom, fajl sa tzv. *objektnim kodom* (*object file*, tipična ekstenzija *.obj* ili *.o*; termin nema veze sa objektno orijentisanim programiranjem)
- ❖ Bilo koja greška u prevođenju uzrokuje da prevodilac ne proizvede izlazni *obj* fajl; bez obzira na to, prevodilac po pravilu nastavlja prevođenje, pokušavajući da prevaziđe svaku grešku i prijavljuje sve eventualne druge greške na koje naiđe
- ❖ Fajl sa izvornim kodom sastoji se isključivo od deklaracija: tipova (uključujući i klase), funkcija, objekata i drugog. *Deklaracija* (*declaration*) je iskaz koji uvodi identifikator u program
- ❖ Svako ime (identifikator) koje se koristi u programu mora najpre biti *deklarisano*, u suprotnom će prevodilac prijaviti grešku u prevođenju

Mart 2020.

Copyright 2020 by Dragan Milićev

38

Prevođenje

- ❖ Na primer, deklaracija globalnog statičkog objekta jeste i *definicija*, koja ima efekat pravljenja objekta:

```
int n = -16;
```

- ❖ Ova definicija ima:

- deklarativni deo (pre znaka =), koji deklarise ime (prevodilac uvodi ime u tabelu simbola)
- inicijalizator (izraz iza znaka =), kojim se inicijalizuje objekat

- ❖ Po pravilima jezika, ovakav objekat ima *statičko trajanje skladištenja* (*static storage duration*) i *statički životni vek*; prema semantičkim pravilima jezika C++, za ovakve objekte važi to da postoji jedna instanca objekta za svaku definiciju (za razliku od npr. definicija automatskih objekata, za koje se kreira nova instanca svaki put kada izvršavanje dođe do takve definicije)

- ❖ Zbog toga, za ovakve objekte prevodilac može (i to po pravilu radi) da alokira prostor *statički*, za vreme prevođenja; taj prostor održava se u prevedenom objektnom fajlu: za svaki takav objekat odvoji se prostor u prevedenom zapisu za smeštanje tog objekta

- ❖ U navedenom primeru, inicijalizator objekta je *konstantan izraz* (*constant expression*) — izraz čiji se rezultat može izračunati u vreme prevođenja

- ❖ Za ovakve statičke objekte fundamentalnog tipa, inicijalizovane konstantnim izrazom, prevodilac po pravilu inicijalizuje statički alocirani prostor vrednošću izraza još u vreme prevođenja

- ❖ Za funkcije, definicija je ona deklaracija koja daje i telo funkcije; za definiciju funkcije, prevodilac generiše binarni mašinski kod za instrukcije koje predstavljaju prevod naredbi iz tela funkcije

Mart 2020.

Copyright 2020 by Dragan Milićev

41

Prevođenje

- ❖ Kada naiđe na novu deklaraciju, prevodilac dodaje deklarisani identifikator u strukturu podataka koju izgrađuje tokom prevođenja i koja se tradicionalno naziva *tabela simbola* (*symbol table*); u ovoj strukturi prevodilac čuva informacije o svakom deklarisanom identifikatoru: o tome kojoj jezičkoj kategoriji pripada (tip, objekat, funkcija itd.), kog je tipa, kao i sva ostala svojstva deklarisanog entiteta definisana pravilima jezika

- ❖ Kada naiđe na neki upotrebljen identifikator, prevodilac:

- proverava da li je taj identifikator deklarisan i da li je dostupan, po pravilima jezika; ako nije, prijavljuje grešku;
- proverava da li je identifikator upotrebljen u skladu sa pravilima jezika i ako nije, prijavljuje grešku; na primer, ne može se vršiti operacija $f++$ ako je f funkcija, ili operacija $a()$ ako je a objekat tipa *int* i slično;
- ako je to definisano semantikom jezika, zna kako da generiše kod za upotrebu tog identifikatora u odgovarajućem kontekstu

A.cpp

```
int n = -16;

void f () {
    ++n;
}
```

A.obj

```
n: dd 0xfffffffff0
f: load r1,n
inc r1
store r1,n
ret
```


Prevođenje

- ❖ Sada definisane entitete želimo da koristimo u drugom fajlu *B.cpp*, ali tako da se odnose na entitete već definisane u *A.cpp*:

```
// B.cpp
```

```
void g () {  
  n++;  
  f();  
}
```

B.cpp

```
int n;  
  
void g () {  
  ++n;  
  f();  
}
```

B.obj

```
symbols  
↑"n": n  
↑"g": g  
end symbols
```

n: dd 0

```
int n;
```

- ❖ Ako se u ovom fajlu ne navede deklaracija objekta *n* i funkcije *f*, prevodilac će prijaviti grešku jer identifikator nije deklarisan

onda će prevodilac nju i dalje smatrati *definicijom*, i ponovo će alocirati prostor

za taj objekat, iako nije inicijalizovan; osim toga, mogao bi da operacije sa *n* u potpunosti prevede, koristeći adresiranje lokacije tog alociranog prostora

- ❖ Ovo nije željeno ponašanje, već želimo da se ove operacije odnose na *n* i *f* definisane u drugom fajlu *A.cpp*, a ne da se definišu novi entiteti

Mart 2020.

Copyright 2020 by Dragan Milićev

43

Prevođenje

- ❖ Međutim, osim toga, prevodilac u prevedenom fajlu ostavlja i informacije o svim imenima (simbolima) koja su definisana u datom fajlu, a mogu se koristiti u drugim fajlovima; ovakva imena nazivaju se imena sa *spoljašnjim vezivanjem* (*external linking*)

- ❖ U posebnom delu objektnog fajla, tipično u zaglavlju, prevodilac pravi tabelu takvih, *izvezanih* simbola, ostavljajući samo informaciju o:

- imenu (simbolu, prost niz znakova), bez ikakvih informacija o tome kakav je entitet predstavljalo to ime u programu (šta je, kog je tipa itd.)
- adresi, relativnoj u odnosu na početak binarnog prevoda (ili segmenta) u koji se to ime preslikava

- ❖ Na primer, po pravilima jezika C++, globalni objekat ili funkcija ima spoljašnje vezivanje, osim ako je eksplicitno deklarisana kao *static* (tada ima interno vezivanje)

- ❖ Imena koja imaju *interno vezivanje* (*internal linking*) ne mogu se koristiti u drugim fajlovima; prevodilac za ovakva imena ne ostavlja ovakve informacije u generisanoj tabeli simbola u objektnom fajlu

A.cpp

```
int n = -16;  
  
void f () {  
  ++n;  
}
```

A.obj

```
symbols  
↑"n": n  
↑"f": f  
end symbols  
  
n: dd 0xffffffff0
```

```
f: load r1,n  
inc r1  
store r1,n  
ret
```

Mart 2020.

Copyright 2020 by Dragan Milićev

42

Prevođenje

- ❖ Jezici C i C++ omogućavaju pisanje asemblerskog koda unutar C/C++ koda *asm* deklaracijom koja se može naći unutar bloka (uvek unutar tela funkcije):
asm (*string_literal*);
- ❖ Niz znakova unutar ove deklaracije je obično kratak deo asemblerskog programa na asembleru ciljnog procesora. Prevodilac će generisati mašinski kod unutar koda generisanog za okružujuću funkciju, na mestu ove deklaracije
- ❖ Sadržaj i uopšte podriška za ovu direktivu nije obavezna i krajnje je zavisna od prevodioca
- ❖ U svakom slučaju, ugrađeni asemblerski kod je zavisan od ciljnog procesora, ali može biti zavisan i od operativnog sistema ukoliko sadrži sistemske pozive ili druge zavisne elemente
- ❖ Prevodilac može (i to po pravilu prevodioci rade) omogućiti upotrebu identifikatora iz okružujućeg C/C++ programa; stepen i način ove podriške zavisi od prevodioca
- ❖ Na primer, prevodilac može podržati upotrebu identifikatora:
 - ❖ funkcija i statičkih podataka, koje može prevesti u (apsolutne) adrese ovih elemenata
 - ❖ lokalnih podataka, koje mora da razreši relativnim adresiranjem (registarskim indirektnim sa pomerajem)
- ❖ Za realizaciju poziva potprograma neophodno je znati konvenciju (protokol) prenosa argumenta na datom procesoru i prevodiocu, što se mora obezbediti u samom asemblerskom kodu

Mart 2020.

Copyright 2020 by Dragan Milićev

45

Referenciranje lokalnog podatka prevešće se u registarsko indirektno adresiranje sa pomerajem tog podatka

```
void f (int x, int y);  
extern static int x;  
  
void g (int y) {  
    asm (  
        load r0, y  
        push r0  
        load r0, x  
        push r0  
        call f  
    );  
}
```

Referenciranje statičkog podatka ili funkcije prevešće se u memorijsko direktno adresiranje sa adresom tog elementa

Prevođenje

- ❖ Da bismo napravili deklaraciju koja nije i definicija, za ovakav statički objekat potrebno je navesti ključnu reč *extern*:

extern int n;

Sada prevodilac neće alocirati prostor za ovaj objekat

- ❖ Za funkciju je dovoljna deklaracija bez tela funkcije, ona nije definicija (reč *extern* može da se piše, ali ne mora):

extern void f ();

- ❖ Međutim, pošto *n* i *f* nisu definisani, prevodilac ne može u potpunosti prevesti operacije sa njima; umesto toga, on će generisati binarni kod za mašinske instrukcije koje implementiraju potrebne operacije, ali sa adresnim poljima u kojima je potrebno definisati adrese tih entiteta postavljenim na proizvoljnu, nedefinisanu vrednost (npr. sve nule); ovakva polja tako ostaju *nerazrešena* (*unresolved*) u vreme prevođenja

- ❖ Zbog toga prevodilac u zaglavlju objektnog fajla, u spisku simbola, ostavlja informacije o svim takvim *uvezanim* simbolima, tj. simbolima koji se koriste, a nisu definisani u datom fajlu, kao i o mestima u binarnom kodu na kojima se nalaze nerazrešena adresna polja mašinskih instrukcija u koja treba naknadno upisati adrese
- ❖ Time prevodilac završava svoj posao; prema tome, objektni fajl nije izvršiv, između ostalog, zbog toga što mašinske instrukcije mogu da imaju nerazrešena adresna polja

Mart 2020.

Copyright 2020 by Dragan Milićev

44

B.cpp

```
extern int n;  
void f();  
  
void g () {  
    ++n;  
    f();  
}
```

B.obj

```
symbols  
↓"n": ...  
↓"f": ...  
↑"g": g  
end symbols  
  
g: load r1, ?  
inc r1  
store r1, ?  
call ?  
ret
```

Povezivanje

- ❖ Biblioteka (*library*) je fajl sa tipičnom ekstenzijom *.lib*, koja ima principijelno isti format kao i objektni fajl; kada povezuje fajlove, linker tretira ulazne *lib* i *obj* fajlove na isti način (zadaje mu se spisak ulaznih *obj* i *lib* fajlove koje treba povezati)
- ❖ Razlika je u tome što je *obj* fajl nastao prevođenjem jednog izvornog fajla, dok je *lib* nastao *povezivanjem* više *obj* (i moguće drugih *lib*) fajlova u jedan *lib* fajl
- ❖ Motiv je praktičan: kod za neku biblioteku (potprograma, struktura, tipova) za određenou namenu može da ima mnogo (na stotine) izvornih fajlova; nepraktično bi bilo koristiti to mnoštvo fajlova i sve pojedinačno ih davati na povezivanje u programe koji ih koriste; “pakovanjem” u biblioteku koja se isporučuje i povezuje kao jedan *lib* fajl olakšava se rukovanje
- ❖ Biblioteka svakako izvozi simbole koje definiše (upravo one koji predstavljaju njene usluge, ono za šta se ona i koristi), ali može i da ih uvozi, ukoliko zavisi od neke druge biblioteke (koristi simbole iz druge biblioteke)

Mart 2020.

Copyright 2020 by Dragan Milićev

47

Povezivanje

- ❖ Zadatak da od skupa objektnih fajlova napravi program, tj. izvršiv fajl, ima program koji se naziva *poveziivač*, tj. *linker* (*linker*): linkeru se zadaje spisak ulaznih objektnih (*.obj*) fajlova i zadatak da napravi *izvršivi* (*executable*, *.exe*) fajl kao svoj izlaz
- ❖ Linker taj zadatak obavlja u dva prolaza:

- u prvom prolazu analizira ulazne fajlove, veličinu njihovog binarnog sadržaja (prevoda), i pravi mapu *exe* fajla; osim toga, sakuplja informacije iz tabela simbola *obj* fajlova i izgrađuje svoju tabelu simbola; u tu tabelu simbola unosi izvezenne simbole iz *obj* fajlova, za koje odmah može da izračuna adresu u odnosu na ceo *exe* fajl
- u drugom prolazu generiše binarni kod, i ujedno razrešava nerazrešena adresna polja mašinskih instrukcija na osnovu informacija o adresama u koje se preslikavaju simboli iz njegove tabele simbola

A.obj

B.obj

C.obj

P.exe

```
symbols
↑"n": n
↑"f": f
end symbols
n: dd ffffffff0
f: ...
```

```
symbols
↑"g": g
↑"a": ...
↑"r": ...
end symbols
g: load r1,?
inc r1
store r1,?
call ?
ret
```

```
n: dd ffffffff0
f: ...
g: load r1,n
inc r1
store r1,n
call f
ret
...
```

Mart 2020.

Copyright 2020 by Dragan Milićev

46

Povezivanje

❖ U principu, prema tome, linker može da prijavljuje samo dve vrste grešaka:

1. *Simbol nije definisan*: kada napravi evidenciju (u svojoj tabeli simbola) o tome koji fajlovi izvoze (definišu) koje simbole, a koji fajlovi uvoze koje simbole, linker može da zaključiti da je neki fajl tražio (uvezao) neki simbol koji nijedan fajl nije definisao (uvezao); u informaciji o ovoj grešci linker ne može da kaže ništa o tome šta je simbol bio u izvornom programu niti gde je u izvornom kodu tražen (jer te informacije po pravilu i nema); tipični uzroci jesu:
 - zaboravljena definicija neke deklarirane funkcije ili objekta (retko, čista omaška)
 - zaboravljen neki *obj* ili *lib* fajl na spisku za linkovanje (omaška)
 - neka povezana biblioteka zavisi od (uvozi simbole iz) neke druge biblioteke, koja nije povezana
2. *Simbol višestruko definisan*: kada naide na simbol koji neki fajl izvozi, a isti taj simbol već postoji u tabeli simbola jer ga je neki drugi fajl već izvezao, linker prijavljuje grešku; ovo je situacija tzv. *sukoba imena* (*name clash*): dva izvorna fajla definisala su ista globalna imena sa eksternim vezivanjem (primetiti to da imena sa internim vezivanjem nisu ni vidljiva linkeru i ne mogu da naprave sukob); tipični uzroci jesu:
 - sukob imena u korisničkom programu
 - sukob imena iz korisničkog programa sa imenom koje je izvezla biblioteka

Ovakve pojave, tj. sukobe imena, značajno smanjuje korišćenje koncepta *prostora imena* (*namespace*) na jeziku C++, jer generisani simbol sadrži puno, kvalifikovano ime (uključuje i nazive svih ugnežđenih prostora imena), čime se izbegava potreba za globalnim imenima i obeshrabruje njihova upotreba

Mart 2020.

Copyright 2020 by Dragan Milićev

Povezivanje

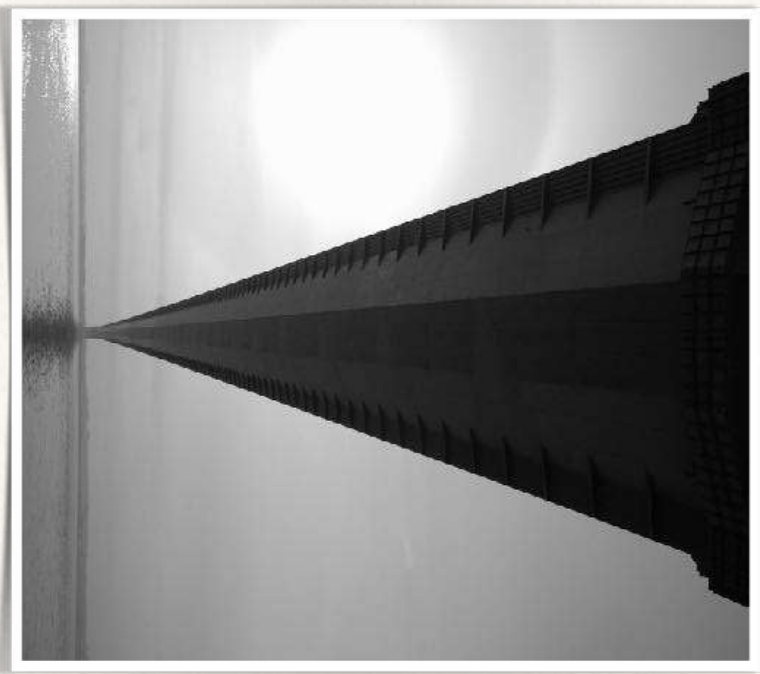
- ❖ Posao pravljenja biblioteke obavlja isti linker, koji može raditi u dva režima, za pravljenje *exe* i za pravljenje *lib* fajla (režim mu se zadaje parametrom prilikom pokretanja)
- ❖ Naravno, posao pravljenja *lib* fajla se razlikuje od posla pravljenja *exe* fajla iz nekoliko razloga:
 - *exe* može imati drugačiji format od *lib* (i *obj*) fajla
 - *exe* sadrži i informacije koje *lib* (i *obj*) fajlovi ne sadrže, a koje su potrebne operativnom sistemu za pokretanje programa; na primer, barem početnu adresu prve instrukcije od koje operativni sistem treba da počne izvršavanje programa
 - kada pravi *exe*, linker mora da završi posao bez nerazrešenih simbola; *lib* može da ima simbole koje uvozi i koji ostaju nerazrešeni
- ❖ Globalna funkcija *main* na jeziku C++ prevodi se kao najobičnija funkcija koju poziva kod koji se najpre izvršava prilikom pokretanja programa i koji onda poziva funkciju *main* kao bilo koju drugu funkciju; nakon povratka iz nje, poziva sistemski poziv za gašenje programa; ovaj okružujuć kod može biti u nekom *obj* fajlu koji se uvek podrazumevano povezuje

Mart 2020.

Copyright 2020 by Dragan Milićev

Glava 5: Organizacija i alokacija memorije

- ❖ Monoprocetni sistem
- ❖ Partitionisanje
- ❖ Kontinualna alokacija
- ❖ Segmentna organizacija
- ❖ Segmentno-stranična organizacija
- ❖ Stranična organizacija
- ❖ Zaštita



Mart 2020.

Copyright 2020 by Dragan Milićev

51

Povezivanje

- ❖ Objektni (*obj*) fajlovi, biblioteke (*lib*) i izvršivi fajlovi (*exe*) zapisani su u binarnom (ne tekstualnom) formatu, prema određenoj unapred definisanoj strukturi koja definiše sve elemente koji ovi fajlovi sadrže: tabele izvezenih i uvezenih simbola, početnu adresu izvršavanja programa, sam binaran zapis sa adresama u koje se smešta i drugo
- ❖ Jedan standardan format koji se danas široko koristi i koji definiše način zapisivanja *obj*, *lib*, *exe* i drugih fajlova slične namene naziva se *ELF (executable/linkable format)*
- ❖ Kada se pokreće proces nad programom zadatim u *exe* fajlu, OS mora da analizira sadržaj tog fajla, dekoduje ga, smesti binarni sadržaj sa statičkim podacima i mašinskim instrukcijama u memoriju odvojenu za proces koji se kreira nad tim programom i započne njegovo izvršavanje od početne adrese koja je definisana u *exe* fajlu

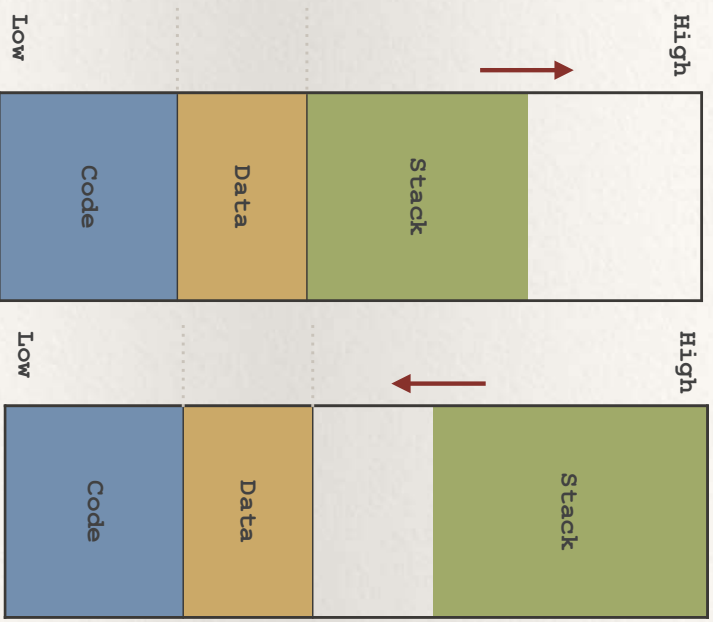
Mart 2020.

Copyright 2020 by Dragan Milićev

50

Monoprocesni sistem

- ❖ Unutar svog raspoloživog prostora, proces može da organizuje *logičke segmente* (programski kod, statički podaci, prostor za alokaciju dinamičkih podataka, stek) na proizvoljan način
- ❖ Pošto stek stalno menja svoju veličinu, vrlo često i intenzivno tokom izvršavanja (pozivi potprograma i povratak iz njih), pitanje je gde ga alocirati
- ❖ Ako se stek alocira iza segmenata za kod i podatke i raste na gore, prema slobodnom delu prostora, postavlja se pitanje kako proširiti prostor za podatke ako je to potrebno (npr. za potrebe dinamičkih struktura)
- ❖ Zato se često stek alocira na vrhu prostora procesa i raste “na dole”: processorske instrukcije *push* i *pop* rade tako da stek raste ka nižim adresama



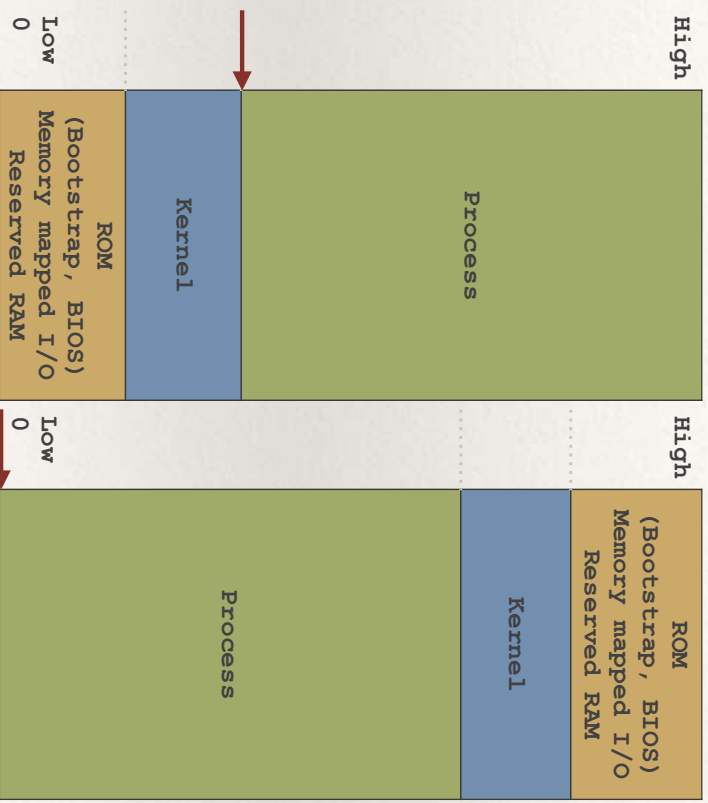
Mart 2020.

Copyright 2020 by Dragan Milićev

53

Monoprocesni sistem

- ❖ U monoprocesnom sistemu, u memoriji je samo jedan proces, i on može da koristi sav memorijski prostor koji mu je na raspolaganju - sav RAM osim onog koji zauzima kernel i delovi rezervisani arhitekturom (ROM, memorijski preslikan U/I, rezervisan RAM)
- ❖ Pošto je adresa početka dela prostora koji će proces zauzimati za vreme izvršavanja uvek ista i unapred poznata, prevodilac ili assembler (za ciljni OS) mogu da generišu kod koji počinje od te poznate početne adrese (0 ili neka druga):
`org process_start_address`
...
- ❖ Adrese koje program koristi u memorijskom direktnom adresiranju su i konačne - fizičke adrese memorije koje će instrukcije generisati tokom izvršavanja



Mart 2020.

Copyright 2020 by Dragan Milićev

52

Monoprocesni sistem

- ❖ Inicijalizaciju registra *SP* na odgovarajuću vrednost (u zavisnosti od toga gde je alocirani stek) može da izvrši (potpuno ravnopravno):
 - ❖ sam program, instrukcijama na početku svog izvršavanja
 - ❖ OS prilikom pokretanja procesa, na unapred definisanu adresu (npr. *vrh*) ili na adresu zapisanu u *exe* fajlu
- ❖ Prostor za dinamičke podatke organizuje sam proces, instrukcijama samog programa, u prostoru koji preostaje (u posebnom logičkom segmentu prostora procesa). Bibliotečne funkcije tipa *malloc/free* organizuju strukture podataka koje vode evidenciju o slobodnim i zauzetim delovima unutar ovog prostora i o tome OS ne mora da vodi računa, to je potpuno nevidljivo za OS - odgovornost je na samom programu i ovi potprogrami su deo koda samog procesa
- ❖ Prema tome, prilikom pokretanja procesa, OS treba da dekoduje sadržaj *exe* fajla programa, učitati binarni sadržaj (mašinski kod instrukcija i statički alocirane i inicijalizovane podatke) u prostor predviđen za proces, po potrebi inicijalizuje *SP* i započne izvršavanje procesa počev od adrese koja je definisana u *exe* fajlu
- ❖ Ostaje otvoreno pitanje kako sprečiti da proces, zbog mogućnosti da upotrebi proizvoljnu adresu, pristupi delu memorije koji zauzima kernel ili drugi sadržaj i neovlašćeno čita ili kumpira taj sadržaj - pitanje zaštite

Mart 2020.

Copyright 2020 by Dragan Milićev

55

Monoprocesni sistem

- ❖ Kako sprečiti da stek poraste toliko (recimo, dubokim ugnežđivanjem poziva potprograma, npr. u dubokim rekurzijama) da "pregazi" druge segmente (podatke, kod) i kumpira njihov sadržaj?
- ❖ Jedan pristup je sledeći:
 - ❖ u programu se obezbedi jedna statički alocirana promeljiva koja čuva adresu granice do koje stek može da raste; njena vrednost se inicijalizuje statički, na osnovu prostora koji zauzimaju prevedeni kod i statički alocirani podaci, a tokom izvršavanja programa može da se menja - granica se "pomera" za potrebe alokacije dinamičkih podataka
 - ❖ prevodilac (ili programer na asembleru) na početku koda svakog potprograma (ili pre poziva svakog potprograma) generiše instrukcije koje proveravaju da li za aktivacioni blok na steku ima dovoljno prostora i signaliziraju grešku (gase proces) ako ne postoji
 - ❖ ako program ovo ne radi, ili ne uradi kako treba na svim mestima, može doći do korupcije
- ❖ Drugi, efikasniji i robusniji pristup, koji ne zahteva odgovornost softvera za sprečavanje ovog problema, zahteva podršku hardvera i biće opisan kasnije

Mart 2020.

Copyright 2020 by Dragan Milićev

54

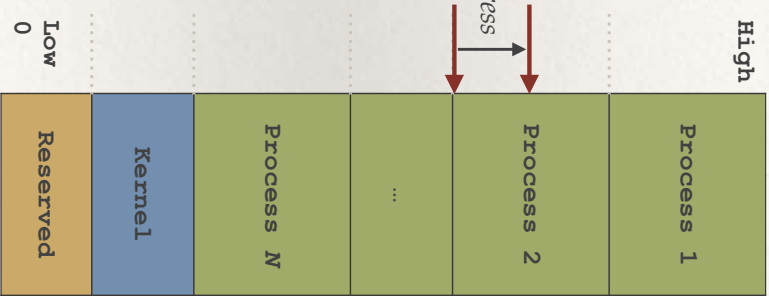
Partitionisanje

- ❖ Sada adresa od koje počinje prostor dodeljen procesoru nije više poznata unapred, za vreme prevođenja/ pisanja assemblerkog programa, pa adresa kojom se adresira fizička memorija (tzv. *fizička adresa*, *physical address*) nije ista kao ona koju je generisala instrukcija - *logička adresa* (*logical address*)
- ❖ Fizička adresa se dobija sabiranjem logičke adrese i *bazne adrese* (*base address*) procesa - adrese početka oblasti u memoriji koju zauzima proces
- ❖ Mašinske instrukcije programa generišu logičke adrese na osnovu načina *Physical address* adresiranja, i to za *svaku* adresiranje, i instrukcije i podatka (potencijalno više puta tokom izvršavanja instrukcije)
- ❖ Skup adresa koje instrukcije mogu da generišu čini *logički* (ili *virtuelni*, *virtual*) adresni prostor svakog procesa (može početi od 0)
- ❖ Pre nego što se adresa uputi memoriji preko magistrale, transformiše se u fizičku adresu kojom se adresira memorija i *fizički adresni prostor* (*physical address space*)
- ❖ Svaki proces ima svoj virtuelni adresni prostor: skup lokacija sa logičkim adresama i sadržajem koji je u opštem slučaju različit i nezavisan od sadržaja istih logičkih adresa drugih procesa
- ❖ Prevodilac i assembler mogu zato generisati mašinski zapis koji pretpostavlja smeštanje procesa počev od (logičke) adrese 0 i koristiti sve načine adresiranja, pa i memorijsko direktno (apsolutne adrese, ali u virtuelnom prostoru)

Mart 2020.

Copyright 2020 by Dragan Milićev

57



Partitionisanje

- ❖ U multiprocesnom sistemu treba smestiti više procesa u deo RAM-a raspoloživ za procese
- ❖ Jedan jednostavan način:
 - ❖ raspoloživi prostor podeliti na N jednakih i disjunktih delova (gde je N fiksiran broj), *particija* (*partition*)
 - ❖ svaku particiju može zauzimati samo jedan proces, ili ona može biti slobodna
- ❖ OS vodi jednostavnu evidenciju o tome koja je particija slobodna, a koja zauzeta. Za svaki kreiran proces OS vodi evidenciju o tome u kojoj particiji se proces izvršava
- ❖ Kada se proces ugasi, njegova particija se oslobađa. Proces se može pokrenuti samo ako postoji slobodna particija u koju se proces učitava. Tada se ta particija označava zauzetom
- ❖ Proces može biti smešten u bilo koju slobodnu particiju, sve su ravnopravne i jednake



Mart 2020.

Copyright 2020 by Dragan Milićev

56

Partitionisanje

- ❖ Semantika izvršavanja instrukcija procesa ne zavisi od njegove lokacije u fizičkoj memoriji, odnosno od particije u koju je smešten, pošto on "vidi" samo svoj logički prostor (virtuelne adrese). Zato proces može biti i premešten (prostom kopiranjem) u drugu particiju, tj. na drugo mesto - proces je *relokatable*
- ❖ Kernel mora da ima pristup do svih delova operativne memorije, kako bi ih održavao (učitavao procese, pristupao ROM-u, I/O prostoru itd.). Mogući pristupi:
 - ❖ tokom izvršavanja koda kernela, nema preslikavanja - svaka logička adresa je ujedno i fizička; kernel "vidi" ceo fizički adresni prostor kao svoj prostor, pod uslovom da može da adresira ceo fizički adresni prostor adresama koje generiše (tj. da procesor to omogućava)
 - ❖ kernel tokom izvršavanja svog koda u bazni registar može da postavi proizvoljnu vrednost (0 ili neku drugu), kako bi pristupio delu memorije kom je potrebno; prilikom prelaska u kernel kod treba upisati vrednost koja omogućava pristup delu memorije kernela u ovaj registar, a po povratku na izvršavanje procesa vratiti baznu adresu tog procesa; ukoliko treba da pristupi delovima memorije za procese, kernel upisuje odgovarajuće bazne adrese za potrebne pristupe
- ❖ Ostaju otvorena pitanja - problem zaštite:
 - ❖ kako zaštititi prostor koji zauzimaju drugi procesi od pristupa procesa koji prekorači granicu svoje particije, ukoliko je virtuelni adresni prostor veći od particije (u suprotnom instrukcija i ne može generisati adresu preko te veličine)?
 - ❖ kako sprečiti da proces upiše u bazni registar proizvoljnu vrednost i tako pristupi bilo kom delu memorije, pa i onom koji zauzima kernel ili neki drugi proces?

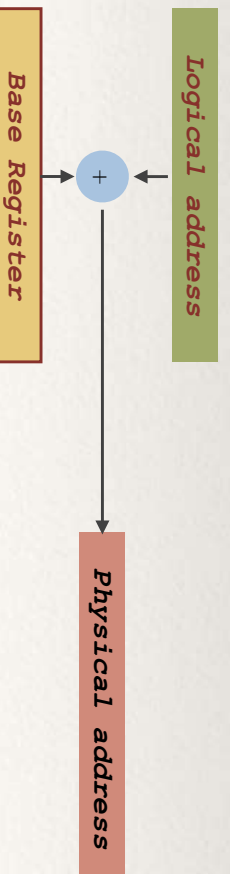
Mart 2020.

Copyright 2020 by Dragan Milićev

59

Partitionisanje

- ❖ Preslikavanje logičke (virtuelne) adrese u fizičku vrši se pri *svakom* adresiranju memorije tokom izvršavanja instrukcije, i za adresiranje instrukcije, i za adresiranje podataka - potencijalno više puta tokom iste instrukcije
- ❖ Zato ovo preslikavanje mora da radi hardver - poseban deo procesora koji se bavi preslikavanjem logičke u fizičku adresu (*memory management unit*, MMU)
- ❖ Preslikavanje je ovde jednostavno: na baznu adresu procesa dodaje se logička adresa i tako dobija fizička adresa
- ❖ Da bi MMU znao baznu adresu, mora je imati dostupnu u hardveru procesora - u posebnom, specijalizovanom programski dostupnom registru: procesor mora da obezbedi instrukciju za upis u ovaj registar
- ❖ OS čuva informaciju o baznoj adresi (ili broju particije) za svaki kreiran proces. Kada vrši promenu konteksta, OS upisuje vrednost bazne adrese u ovaj registar za proces kom dodeljuje procesor. OS baznu adresu može da izračuna i samo na osnovu broja particije u kojoj je proces i veličine particije



Mart 2020.

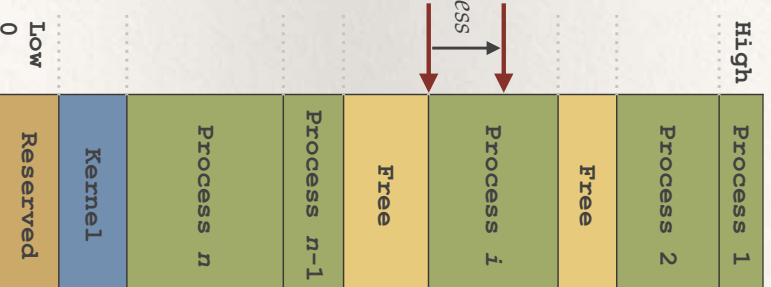
Copyright 2020 by Dragan Milićev

Kontinualna alokacija

- ❖ Delimično rešenje navedenih problema: proces zauzima samo onoliko memorije koliko mu je potrebno, u skladu sa *stvarnom veličinom* prostora koji mu je potreban
- ❖ Informaciju o potrebnom prostoru OS ima u *exe* fajlu programa, na osnovu "memorijskog otiska" (*footprint*) binarnog sadržaja (veličine segmenata za kod i statičke podatke)
- ❖ Odmah iza granice prostora koji zauzima jedan proces može biti alocirani drugi proces. Pošto su veličine tih zauzetih delova različite, *bazna dresa* procesa može biti bilo koja fizička adresa u prostoru namenjenom za procese iza koje ima dovoljno prostora za taj proces
- ❖ Kada se proces ugasi, prostor koji je zauzima se proglašava slobodnim. Pošto su veličine tih delova različite, i slobodni fragmenti su različite veličine i raspoređeni na proizvoljnim lokacijama memorije
- ❖ Proces ponovo može biti smešten ili premešten (prostirni kopiranjem sadržaja memorije i promenom bazne adrese) na bilo koje mesto u prostoru za procese, njegovo izvršavanje ne zavisi od toga - proces je inherentno *velokatilban*

Mart 2020.

Copyright 2020 by Dragan Milićev



61

Partitionisanje

- ❖ Partitionisanje je jednostavna tehnika, ali poseduje velike nedostatke:
 - ❖ stepen multiprogramiranja (broj aktivnih procesa koji su u memoriji i mogu se izvršavati) je ograničen brojem particija - ne može biti proizvoljan
 - ❖ veličina particije mora da bude onolika koliko je veliki logički adresni prostor procesa, a fizička memorija mora biti značajno veća od toga (broj particija mora biti dovoljno velik); obratno - veličina particija može biti relativno mala u odnosu na raspoloživu fizičku memoriju
 - ❖ za svaki proces se zauzima cela particija, iako proces možda koristi samo mali deo tog prostora; ostatak je potpuno neiskorišćen i bespotrebno "bačen", jer je dodeljen datom procesu (i ne može biti dodeljen nijednom drugom), a on ga ne koristi
- ❖ Ovakav deo memorije koji je neiskorišćen (slobodan), ali se ne može iskoristiti ni za šta drugo, jer je *unutar* prostora alocirano i rezervisanog samo za onog ko ga koristi, naziva se *interni fragment (internal fragment)*, a ova pojava *interni fragmentacija*; ova pojava je negativna i treba je suzbijati, a kod ove tehnike može biti značajno izražena
- ❖ Zbog ovih nedostataka, ova tehnika je zastarela i odavno se ne koristi, jer je iskorišćenje memorije slabo, a svi današnji OS opšte namene dozvoljavaju neograničen stepen multiprogramiranja

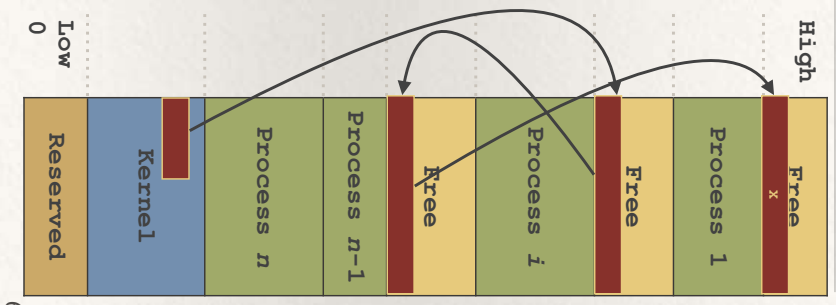
Mart 2020.

Copyright 2020 by Dragan Milićev

60

Kontinualna alokacija

- ❖ OS sada mora da vodi evidenciju, odgovarajućom strukturom podataka i procedurama koje je održavaju, slobodnih fragmenata memorije - njihovih pozicija (početnih adresa) i veličina
- ❖ Na primer, to može da uradi ulančavanjem slobodnih fragmenata u listu, pri čemu se pokazujući za ulančavanje smeštaju u same slobodne fragmente (npr. na početku), kako se za evidenciju slobodne memorije ne bi trošio poseban prostor kernela (kad je ta memorija već slobodna); glava te liste je pokazuivač unutar kernela
- ❖ Kada se proces ugasi, prostor koji je zauzimao se oslobađa: u listu se dodaje oslobođeni fragment, uz spajanje sa eventualno postojećim slobodnim fragmentom ispred i iza onog koji je zauzimao proces, kako bi se slobodni fragmenti ukupnili
- ❖ Kada treba alocirati prostor za nov proces, kernel mora da pronađe slobodan fragment dovoljne veličine koja je procesu potrebna (mora biti veći ili jednak), kada se takav odabere, deo u koji se alocira proces se proglašava zauzetim, a eventualno preostali deo (fragment) ostaje u spisku slobodnih (ali sada kao manji)
- ❖ Kako se delovi proizvoljnih veličina zauzimaju i oslobađaju dinamički, slobodni fragmenti mogu biti ulančani po redosledu smeštanja ili po nekom drugom kriterijumu (npr. prema početnoj adresi ili prema veličini): pogodnost strukture zavisi od vrste operacija koje se sa njom obavljaju
- ❖ Slobodni fragmenti su sada *eksterni*, jer se nalaze izvan prostora koji je nekomе dodeljen



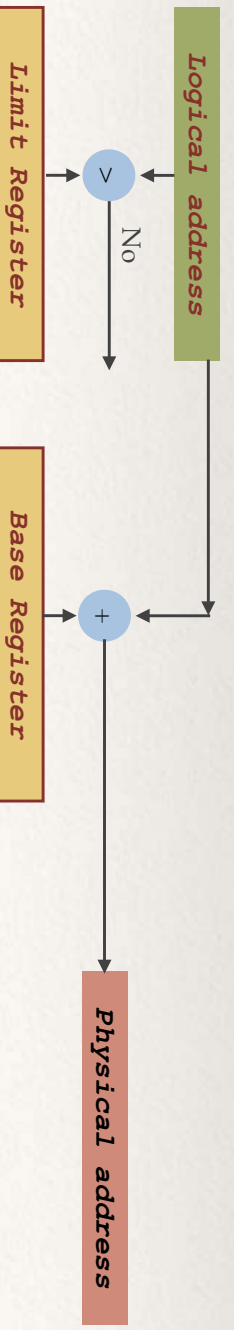
Mart 2020.

Copyright 2020 by Dragan Milićev

63

Kontinualna alokacija

- ❖ Preslikavanje logičke (virtuelne) adrese u fizičku izgleda isto kao i za partitionisanje: fizička adresa dobija se sabiranjem logičke adrese sa baznom adresom tekućeg procesa, za svako adresiranje tokom izvršavanja svake instrukcije procesa
- ❖ Baznu adresu procesor uzima iz za to namenjenog registra, u koji vrednost upisuje OS prilikom promene konteksta
- ❖ Sada svaki proces svakako može da generiše logičku adresu koja je veća od stvarne veličine prostora dodeljenog procesu, jer je logički prostor u opštem slučaju veći od prostora dodeljenog procesu (zato što proces zauzima samo onoliko koliko stvarno koristi). Zato se ne sme dozvoliti ovakvo preslikavanje za adrese preko granice prostora dodeljenog procesu
- ❖ Zato MMU mora da proveriti, pre samog preslikavanja, da li je generisana logička adresa veća od stvarne veličine, opet za svako adresiranje. Informaciju o stvarnoj veličini prostora tekućeg procesa opet mora imati u drugom specijalizovanom, programski dostupnom registru procesora, *registru granice (limit register)* ili *veličine (size)*, u koji vrednost upisuje OS prilikom promene konteksta, isto kao i za baznu adresu (ista tehnika može da se koristi i kod partitionisanja)
- ❖ Ukoliko MMU detektuje prekoračenje, procesor će generisati *izuzetak (exception)*: signal da je instrukcija napravila *prestup u pristupu memoriji (memory access violation)* i da ne može da se izvrši. Ovaj izuzetak obrađuje OS i podrazumevano gasi tekući proces, uz eventualno obaveštenje korisniku (portuku) - detalji kasnije



Mart 2020.

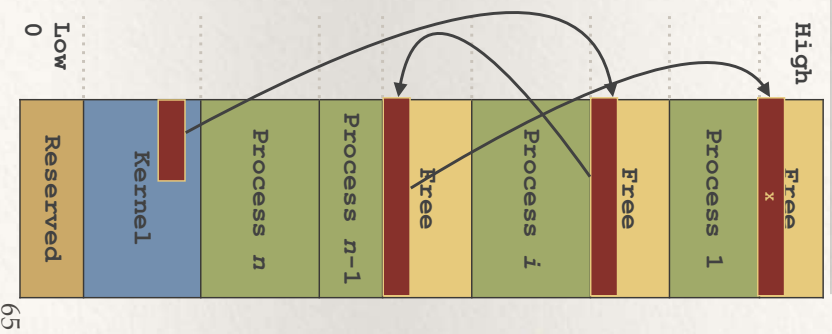
Copyright 2020 by Dragan Milićev

Kontinualna alokacija

- ❖ Kada treba da pronađe mesto za smeštanje procesa, OS treba da pronađe slobodan fragment dovoljne veličine (veći ili jednak veličini koja se alokira) - *algoritmom dinamičke alokacije prostora (dynamic allocation)*
- ❖ Mogući pristupi:
 - ❖ uzeti prvi dovoljno velik segment na koji se naiđe (npr. u ulančanoj, neuređenoj listi) - *prvi koji odgovara (first fit)*
 - ❖ s ciljem da nakon alokacije ostane što manji slobodan fragment, kako bi se smanjila eksterna fragmentacija (količina neupotrebljive slobodne memorije), izabrati onaj koji *najbolje odgovara (best fit)*: od svih slobodnih koji su dovoljno veliki, odabrati onaj najmanji; struktura u koju su organizovani slobodni fragmenti treba da omogući efikasnu pretragu (npr. uređena lista ili stablo)
 - ❖ s ciljem da preostali slobodan fragment bude što upotrebljiviji, odnosno takav da se poveća šansa da se on može upotrebiti za daju alokaciju, odabrati *najveći* slobodan fragment, odnosno onaj koji *najlošije odgovara (worst fit)*
- ❖ Neka iskustva pokazuju da algoritmi *first fit* i *best fit* daju generalno bolji učinak nego algoritam *worst fit*, s tim da nijedan od ta dva nije generalno efikasniji (s tim da je algoritam *first fit* jednostavniji, tj. operacije održavanja strukture su manje složene)

Mart 2020.

Copyright 2020 by Dragan Milićev



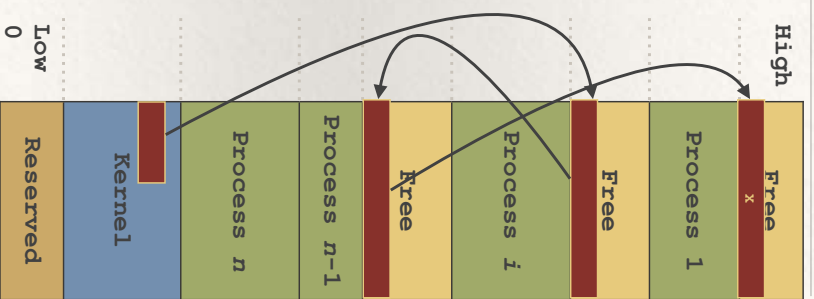
65

Kontinualna alokacija

- ❖ Problem: kada se slobodan fragment zauzme za nov proces, preostaje manji fragment. Taj fragment može biti suviše mali da bi njegovo evidentiranje bilo svrishodno, jer se ne može iskoristiti ni za šta (suviše je mali za bilo kakav proces). Štaviše, može da bude suviše mali da se u njega upšte može smestiti struktura za ulančavanje (npr. svega nekoliko bajtova) - problem *eksterne fragmentacije (external fragmentation)*

- ❖ Moguća rešenja:

- ❖ takve male ostatke upšte ne evidentirati, već ih pridružiti procesu (dati procesu malo više nego što mu treba)
- ❖ alocirati uvek delove zaokružene na neku veličinu (blok), npr. 512B
- ❖ Oba ova pristupa samo smanjuju problem, ne rešavaju ga u potpunosti, jer:
 - ❖ pojavljuje se interna fragmentacija, jer se alociranom delu pridružuje deo koji on ne koristi (slobodan fragment unutar alociranog bloka)
 - ❖ ako se koristi zaokruživanje na cele blokove, slobodan fragment sada može biti veličine samo jednog bloka ili nekoliko susednih blokova, što je opet nedovoljno za alokaciju procesa, jer je blok mali; ako se blok poveća, povećava se interna fragmentacija, pa je iskorišćenje memorije slabije



64

Mart 2020.

Copyright 2020 by Dragan Milićev

Kontinualna alokacija

- ❖ Problem: nakon dužeg rada sistema, slobodna memorija može da postane jako fragmentirana - puno suviše malih slobodnih fragmenata, tako da nijedan ne može da se iskoristi za novu alokaciju, iako je ukupna količina slobodne memorije sasvim dovoljna - *eksterna fragmentacija*
- ❖ Uzrok problema: kontinualni segmenti različite veličine alociraju se u isti linearan prostor; kada se oslobodi deo neke veličine, u taj prostor se može smestiti nov segment koji je ili iste veličine (vrlo retko) ili manji, što ostavlja eksterni fragment
- ❖ Moguće rešenje je *kompakcija* (*compaction*) slobodnog prostora: kernel relocira sve procese tako da ih slaže jedan iza drugog, redom, tako da sav slobodan prostor fuzionise u samo jedan slobodan fragment na samom kraju
- ❖ Problem ovog rešenja: izuzetno zahtevna operacija jer podrazumeva kopiranje najvećeg dela memorije, intenzivno zauzima procesor i može da traje dugo, dok su za to vreme *svi* procesi suspendovani, ne mogu da se izvršavaju: OS zaustavlja rad celog sistema zbog režijske operacije, da bi "počistio svoje dvorište"

Mart 2020.

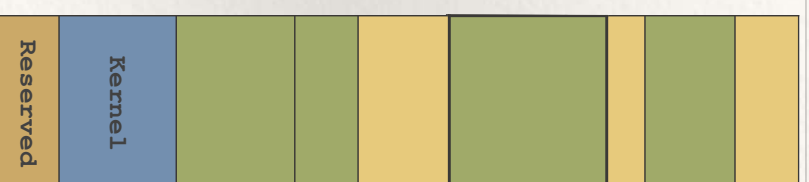
Copyright 2020 by Dragan Milićev



67

Kontinualna alokacija

- ❖ Usluga koju OS može pružiti procesima: sistemski poziv kojim proces traži povećanje (ili smanjenje) svog prostora na zadatu veličinu ili za zadatu promenu veličine (parametar sistemskog poziva)
- ❖ Ako proces traži smanjenje svog prostora, OS taj zahtev uvek može da ispuní - deo kog se proces odriče proglašava slobodnim i dodaje eventualnom slobodnom fragmentu iza procesa, a smanjuje vrednost *limit* tog procesa
- ❖ Ako proces traži povećanje svog prostora:
 - ❖ ako iza procesa postoji slobodan fragment dovoljne veličine, OS može prosto povećati prostor dodeljen procesu (povećanje vrednosti *limit*), a slobodnom fragmentu smanjiti veličinu
 - ❖ ako to nije zadovoljeno, OS može ili da odbije zahtev (vraći grešku iz sistemskog poziva, negativan status), ili da pronađe drugo, dovoljno veliko slobodno mesto u memoriji i relocira proces kopiranjem sadržaja i promenom njegove baze adrese, uz odgovarajuće ažuriranje struktura za evidenciju slobodnih fragmenata (dva susedna oslobođena dela se uvek fuzionišu u jedan veći)

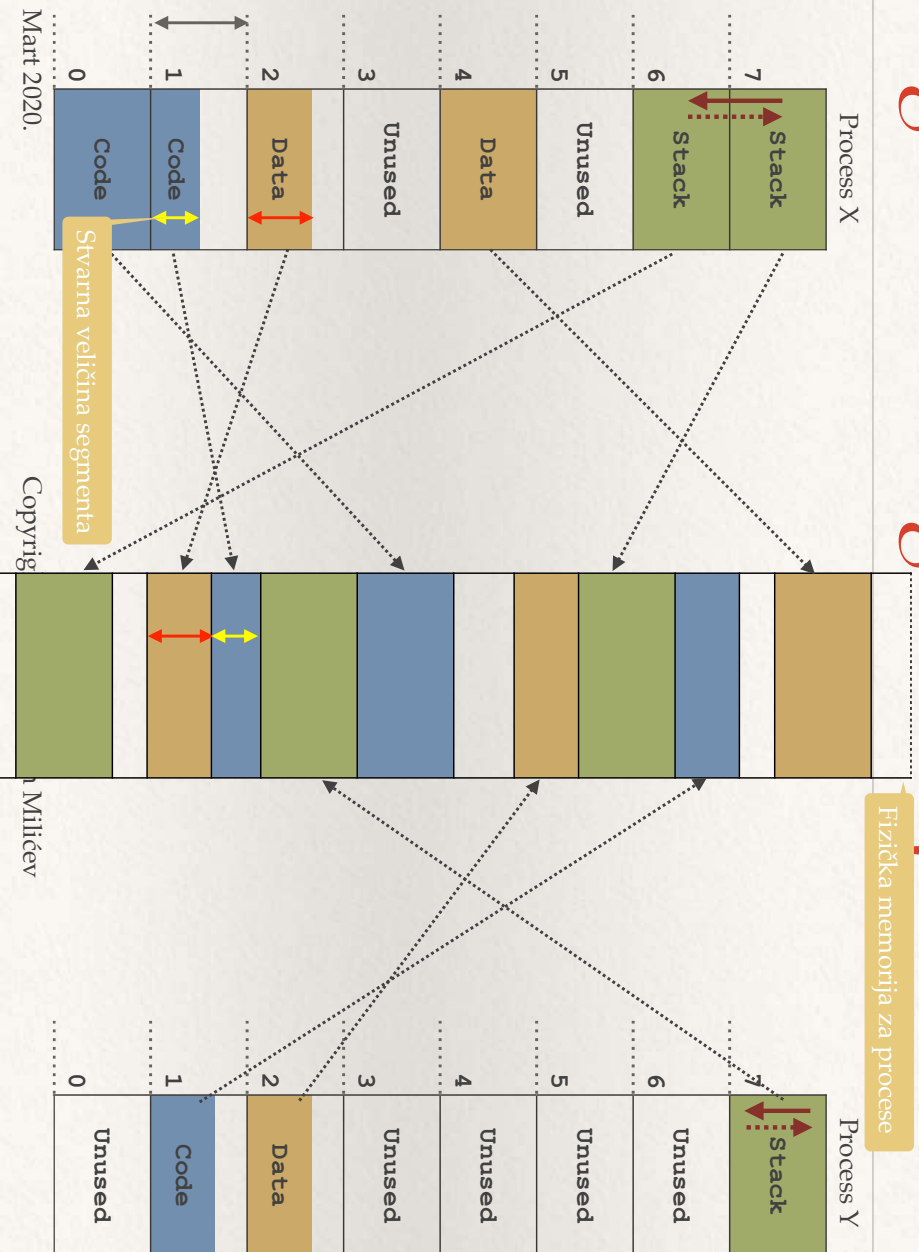


66

Mart 2020.

Copyright 2020 by Dragan Milićev

Segmentna organizacija



Segmentna organizacija

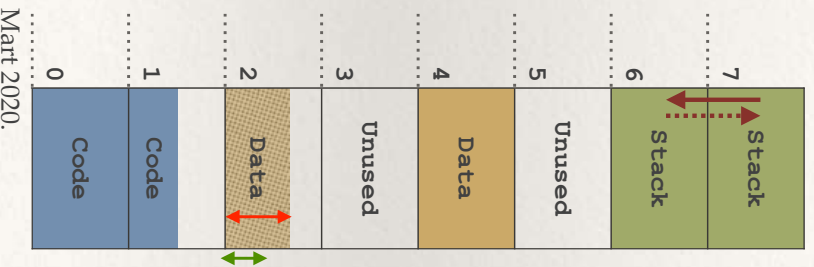
- ❖ Još neki problemi kontinualne alokacije celog prostora procesa:
- ❖ Ako proces organizuje stek na vrhu svog alociranog prostora (stek raste nadole), a kod i podatke na dnu (kao na slici), a između njih ostavi prostor koji može (a ne mora) biti potreban za dinamičku alokaciju podataka tokom izvršavanja, proces zauzima ceo prostor u kome je potencijalno puno neiskorišćenog (interni fragment)
- ❖ Ako više procesa izvršava isti program, u memoriji se nalazi više kopija istog sadržaja (istog programskog koda) koji bespotrebno zauzimaju memoriju - kako ih zajednički deliti između procesa i u memoriji držati samo jednu kopiju istog koda?
- ❖ Ideja za rešenje:
- ❖ logički adresni prostor procesa podeli se na *segmente*, tako da prvih nekoliko bita logičke (virtuelne) adrese određuje broj segmenta u adresnom prostoru - svi segmenti su iste *maksimalne veličine* (uvek stepen dvojke) određene brojem preostalih bita u logičkoj adresi
- ❖ sadržaj procesa se podeli na logičke celine - *segmente*, prema sadržaju: segment za kod, za podatke, za stek itd; segmenti se u virtuelnom adresnom prostoru mogu, ali ne moraju grančiti, između njih može, ali ne mora da bude neiskorišćenih segmenta
- ❖ svaki segment se može smestiti u fizičku memoriju na proizvoljno mesto, od proizvoljne baze adrese - svaki segment ima svoju bazu adresu; segmenti se smeštaju nezavisno
- ❖ svaki segment zauzima u memoriji samo prostor koji mu je potreban, u skladu sa svojom stvarnom veličinom (manje ili jednako maksimalnoj), pa ima svoju *granicu (limit)*



Segmentna organizacija

Physical Address Space

Process X Virtual Address Space



Mart 2020.

Virtual Address
15 : 12

Segment	Offset
0b010	0xd08

Primer: adresibilna jedinica je bajt, virtualna adresa je 16-bita, a fizička adresa 32-bita. Maksimalna veličina segmenta je 8 KB = 2^{13} B. Virtualni adresni prostor ima $2^{16-13} = 8 = 2^3$ segmenata. Zato polje u VA za broj segmenta ima 3, a offset 13 bita.

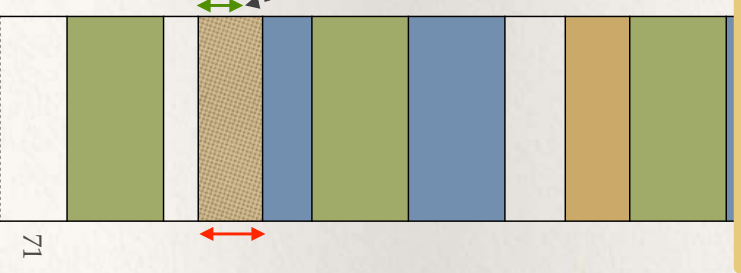
Process X Segment Map Table

	Base Address	Limit
0	0xa0402830	0x1fff
1	0x27134580	0x808
2	0x10145120	0xdff0
3	0	0
4	0xc027f800	0x1fff
5	0	0
6	0x52de00d0	0x1fff
7	0x2a800080	0x1fff

Physical Address
31

$$0x10145120 + 0xd08 = 0x10145e28$$

Copyright 2020 by Dragan Milićev



71

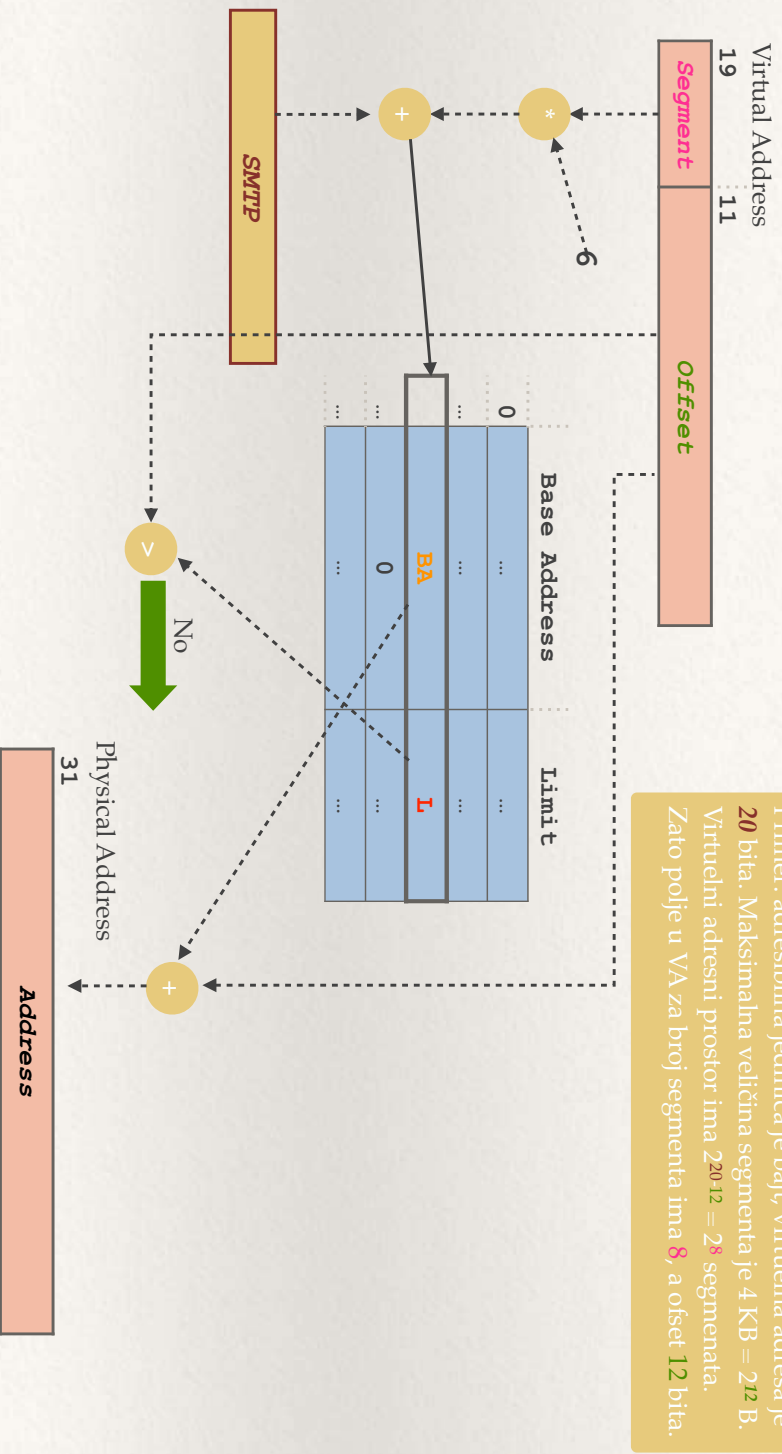
Segmentna organizacija

- ❖ Svaki segment je sada nezavisna jedinica kontinualne alokacije i ima svoju baznu adresu i granicu (stvarnu veličinu)
- ❖ Proces i dalje vidi svoj kontinualni virtualni adresni prostor (od adrese 0 do neke maksimalne), ali je fizički taj prostor diskontinualan u operativnoj memoriji
- ❖ Da bi MMU vršio preslikavanje svake virtualne adrese, najpre izdvađa onoliko viših bita virtualne adrese koji definišu broj segmenta. Ostatak bita u virtualnoj adresi određuju *pomeraj* (*offset, displacement*) adresirane lokacije u odnosu na početak segmenta
- ❖ Zatim za segment sa tim brojem mora da odredi baznu adresu i granicu. Da bi to uradio, mora posedovati ove informacije za svaki segment svakog procesa
- ❖ Zato OS za svaki proces organizuje posebnu strukturu podataka, *tabelu preslikavanja segmenata* (*segment map table, SMT*) koju koristi MMU pri svakom preslikavanju
- ❖ SMT sadrži po jedan ulaz - *descriptor* za svaki segment u virtualnom adresnom prostoru. Deskriptor je određene veličine koja je potrebna da se smeste navedene informacije (bazna adresa i veličina segmenta), određen broj adresibilnih jedinica, često zaokružen na ceo, mali stepen dvojke, npr. 4, 8, 16
- ❖ SMT se nalazi u memoriji, u memorijskom prostoru kernela, koji kernel dodeljuje evidenciji svakog procesa
- ❖ Za segmente koje proces ne koristi, deskriptor segmenta u SMT sadrži neku specijalnu *null* oznaku koja označava to, npr. poseban bit ili, da se ne bi odvajao ceo bit (polovina opsega vrednosti), posebna vrednost za baznu adresu (npr. 0 ili sve jedinice, jer proces ne može biti smešten počev od takve fizičke adrese)

Mart 2020.

Copyright 2020 by Dragan Milićev

Segmentna organizacija



Mart 2020.

Copyright 2020 by Dragan Milićev

73

Segmentna organizacija

- ❖ Da bi MMU znao gde da pronađe tabelu za tekući proces (u čijem kontekstu se izvršava instrukcija i viši preslikavnije), mora posedovati tu adresu u specijalizovanom, programski dostupnom registru *SMTP* (*segment map table pointer*)
- ❖ Na osnovu vrednosti u *SMTP*, broja segmenta i veličine deskriptora, MMU izračunava (fizičku) adresu deskriptora:
 $descr_addr = SMTP + segment_no * descr_size$
- ❖ Ako je *descr_size* stepen dvojke, množenje se svodi na prosto pomeranje ulevo za onoliko bita koliki je taj stepen
- ❖ MMU dovlači sadržaj deskriptora iz fizičke memorije, u jednom ili više ciklusa čitanja, u zavisnosti od veličine deskriptora i veličine reči koja se može preneti u jednom ciklusu
- ❖ Ako je u deskriptoru *null* (segment nije u upotrebi), virtualna adresa ne može da se preslika, izvršavanje instrukcije se prekida, procesor generiše izuzetak koji onda obrađuje OS - prestup u adresiranju memorije (*memory access violation*): proces je generisao adresu *izvan segmenta koji je deklarisan (alocirao)*
- ❖ U suprotnom, proverava se pomeraj u odnosu na granicu (stvarnu veličinu segmenta) dobijenu iz deskriptora: ukoliko pomeraj prekoračuje granicu stvarne veličine segmenta, izvršavanje instrukcije se prekida, procesor signalizira izuzetak koji obrađuje OS - prekoračenje granice segmenta, proces je opet adresirao virtualnu adresu izvan segmenta koji je deklarisan
- ❖ U suprotnom, izračunava se fizička adresa kao zbir bazne adrese dobijene iz deskriptora i pomeraja iz virtualne adrese:
 $p_addr = base_addr + offset$

Mart 2020.

Copyright 2020 by Dragan Milićev

72

Segmentna organizacija

- ❖ Za neke vrste (logičkih) segmentata prevodilac ne mora da generiše nikakav binaran sadržaj u *exe* fajlu, već samo da ostavi zapis o definiciji segmenta, pa tako ni OS ne mora da učitava taj binarni zapis prilikom učitavanja segmentata iz *exe* fajla u memoriju:
- ❖ segment sa statičkim podacima inicijalizovanim nulama, tzv. *bss* segment (*bss* je skraćenica od arhaičnog naziva ovog koncepta u sistemu u kom je prvi put upotrebljen): dovoljno je da prevodilac ostavi informaciju u *exe* fajlu o poziciji i veličini tog segmenta; prilikom kreiranja memorijskog konteksta procesa, OS će u prostor u memoriji alociran za ovaj segment samo upisati sve nule; zbog ovoga neki jezici, poput C/C++ definišu posebnu semantiku za statičke objekte sa podrazumevanom inicijalizacijom na 0 (tzv. *zero initialization*), a takve statičke objekte grupišu u isti segment, odvojen od statičkih objekata koji su statički inicijalizovani drugim vrednostima
- ❖ segment koji može imati proizvoljan (nedefinisan) početni sadržaj, jer je samo prostor za podatke koje će inicijalizovati sam proces tokom izvršavanja: prostor za stek, za dinamičke podatke, za statičke podatke koji se inicijalizuju nekonstantnim izrazima (izrazima koji se izračunavaju izvršavanjem instrukcija tokom izvršavanja programa); ovakve segmente OS treba samo da alocira, ne mora ničim da ih inicijalizuje (nema učitavanja niti upisa nula pri inicijalizaciji)



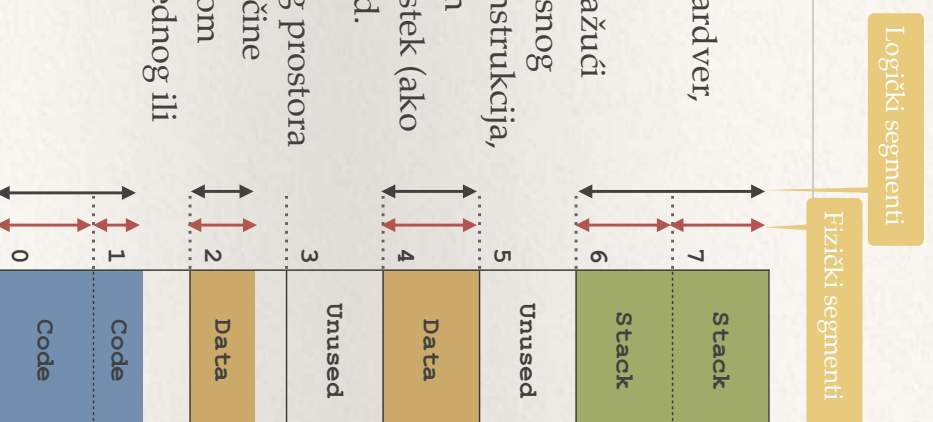
Mart 2020.

Copyright 2020 by Dragan Milićev

75

Segmentna organizacija

- ❖ Do sada opisani segmenti su *fizički* segmenti, jer ih tretira hardver, odnosno MMU procesora na mašinskom nivou
- ❖ Prevodilac prevodi izvorni kod i generiše binarni sadržaj slažući istorodne stvari u *logičke segmente* ili *regione* virtuelnog adresnog prostora: jedan ili više njih sa binarnim kodom mašinskih instrukcija, tzv. *text* segment(i), jedan ili više njih sa binarnim sadržajem inicijalizovanih statičkih podataka, eventualno segment sa stek (ako njega alocira sam proces, a ne OS pri pokretanju procesa) itd.
- ❖ Logički segmenti su kontinualni delovi virtuelnog adresnog prostora proizvoljne dužine (ne obavezno manji od maksimalne veličine fizičkog segmenta) sa istorodnim sadržajem (sa istim načinom korišćenja). Zato se logički segment može protezati preko jednog ili više susednih fizičkih segmentata



Mart 2020.

Copyright 2020 by Dragan Milićev

74

Segmentna organizacija

- ❖ OS može (i to sistemi i rade) pružiti uslugu *dinamičke alokacije* logičkog segmenta (regiona) u virtuelnom adresnom prostoru, koju onda proces može tražiti sistemskim pozivom, tokom svog izvršavanja
- ❖ Na primer, proces može tražiti (alocirati) prostor za dinamičke podatke, u skladu sa potrebom izvršavanja, ili proširiti prostor za stek
- ❖ Efekat ove operacije, pa i njena implementacija, isti su kao i kada se logički segment kreira *statički*, prilikom formiranja memorijskog konteksta, na osnovu definicije u *exe* fajlu
- ❖ U slučaju da ne može da izvrši zahtev, npr. zato što nema odgovarajućeg slobodnog dela u koji može smestiti alocirani fizički segment ili segmente, OS će vratiti grešku (npr. negativnu celobroju vrednost ili *null* vrednost pokazivača)
- ❖ U svakom slučaju, pre nego što pristupa nekom delu svog virtuelnog adresnog prostora, proces *mora* deklarirati (alocirati) taj prostor na neki od ova dva načina, statički ili dinamički (u suprotnom će izazvati izuzetak prilikom tog pristupa)



Mart 2020.

Copyright 2020 by Dragan Milićev

77

Segmentna organizacija

- ❖ U assembleru se segmenti mogu definisati posebnim direktivama, a assembler to onda prevodi u odgovarajući format zapisa u *obj/exe* fajlu:

```
seg  
org ...  
end
```

Direktiva *seg* kojom započinje definicija segmenta, a koja se završava direktivom *end*. Direktivom *org* na početku zadaje se početna adresa segmenta

- ❖ Često i logično ograničenje jeste to da logički segment mora biti poravnat (*aligned*) na početak fizičkog segmenta - polje za pomeraj početne adrese logičkog segmenta mora biti 0
- ❖ Ukoliko takvo ograničenje ne postoji, sistem svakako mora alocirati fizički segment od početne adrese, pa zapravo ignoriše pomeraj u adresi početka logičkog segmenta, a ispred stvarne adrese zadate direktivom tipa *org*, na početku segmenta, ostaje neiskorišćen prostor (interni fragment)
- ❖ U *exe* fajlu su zapisi sa definicijama logičkih segmentata definisani u odgovarajućem binarnom formatu
- ❖ Pri formiranju inicijalnog memorijskog konteksta procesa, OS treba da učitava sadržaj iz *exe* fajla programa, dekoduje ove zapise za definicije segmentata i radi sledeće:

- alokira prostor za smeštanje SMT
- za svaki definisani logički segment formira jedan ili više susednih fizičkih segmentata, u zavisnosti od veličine logičkog segmenta
- alokira prostor za smeštanje sadržaja fizičkih segmentata u memoriji predviđenoj za procese i u taj prostor učitava sadržaj iz *exe* fajla, ako je potrebno
- inicijalizuje ulaze SMT u skladu sa definicijama segmentata u *exe* fajlu i baznim adresama alociranih segmentata



Mart 2020.

Copyright 2020 by Dragan Milićev

76

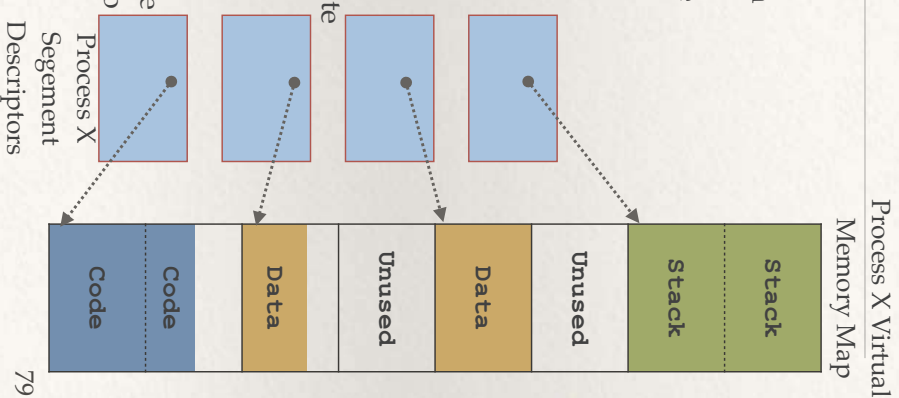
Segmentna Organizacija

- ❖ Da bi realizovao ovu uslugu, OS mora da:
 - ❖ vodi evidenciju o zauzetim logičkim segmentima (regionima) i slobodnim fragmentima, tj. o njihovim pozicijama i veličinama, ali sada unutar virtuelnog adresnog prostora svakog procesa; za te potrebe OS organizuje odgovarajuću strukturu deskriptora logičkih segmenata za svaki proces
 - ❖ sprovodi algoritam dinamičke kontinualne alokacije prostora, ali unutar virtuelnog adresnog prostora svakog procesa
- ❖ Po pravilu postoji i usluga dealokacije (oslobađanja) alocirano^g logičkog segmenta:

```
int mmap (void* base_addr, size_t size);
```
- ❖ Bibliotečne funkcije *malloc*, *calloc* i *free* služe za alokaciju manjih dinamičkih delova memorije i mogu biti implementirane u sistemskoj biblioteci na različite načine:
 - ❖ direktno prosleđuju zahtev sistemskom pozivu
 - ❖ alociraju po potrebi veće komade memorije sistemskim pozivom, a onda internim strukturama podataka i algoritmima alociraju i dealociraju manje komade; njihova organizacija je u isključivoj nadležnosti procesa, odnosno programa koji on izvršava (kod biblioteke koja implementira ove funkcije)

Mart 2020.

Copyright 2020 by Dragan Milićev



79

Segmentna Organizacija

Ovo je pojednostavljen prikaz primera sistemskog poziva u sistemu Linux

- ❖ Na primer, ovaj sistemski poziv može izgledati ovako:

```
void* mmap (void* base_addr, size_t size);
```

Parametar *base_addr* predstavlja početnu logičku (virtuelnu) adresu logičkog segmenta (obično mora biti poravnata na početak fizičkog segmenta), a parametar *size* veličinu logičkog segmenta

- ❖ Međutim, da bi korisnički program odredio adresu u virtuelnom memorijskom adresnom prostoru na kojoj želi alokaciju, mora sam da vodi računa (evidenciju) o delovima svog prostora koji su zauzeti, što nije praktično

- ❖ Zato obično postoje varijante u kojima se ostavlja da sam OS pronađe dovoljno veliki slobodan prostor u virtuelnom adresnom prostoru procesa u kom se može alocirati logički segment tražene veličine, a sistemski poziv će vratiti (virtuelnu) adresu početka alocirano^g prostora; tada prvi argument ne postoji, ili može da bude *null*:

```
void* addr = mmap(NULL, size);
```



Mart 2020.

Copyright 2020 by Dragan Milićev

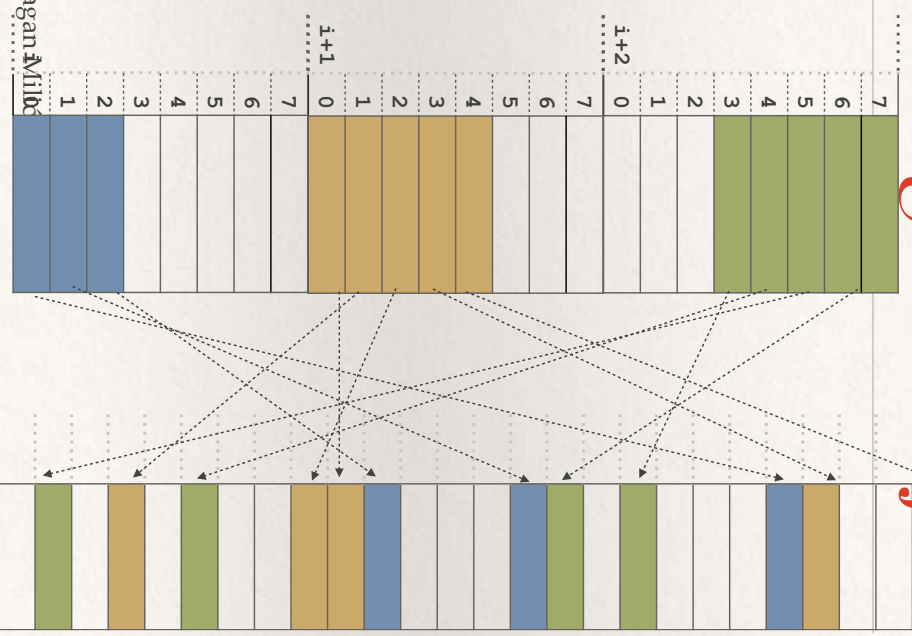
78

Segmentno-stranična organizacija

- ❖ Rešenje problema kontinualne alokacije:
- ❖ logički (virtualni) adresni prostor procesa podeli se na segmente iste maksimalne veličine (uvek stepen dvojke), kao kod segmentne organizacije; segment je logička celina, sadrži istovodni sadržaj
- ❖ svaki segment se logički deli na *stranice* (*page*) iste veličine (uvek stepen dvojke)
- ❖ segment može imati različitu stvarnu veličinu, ali uvek zaokruženu na cele stranice
- ❖ fizička memorija logički je podeljena na *okvire* (*frame*) veličine jednake veličini stranice
- ❖ jedinica alokacije je sada jedna stranica: svaka stranica može se alocirati u bilo koji okvir i uvek se alocira ceo okvir za smeštanje stranice - stranice su uvek poravnate na okvire

Mart 2020.

Copyright 2020 by Dragan Milić



81

Segmentna organizacija

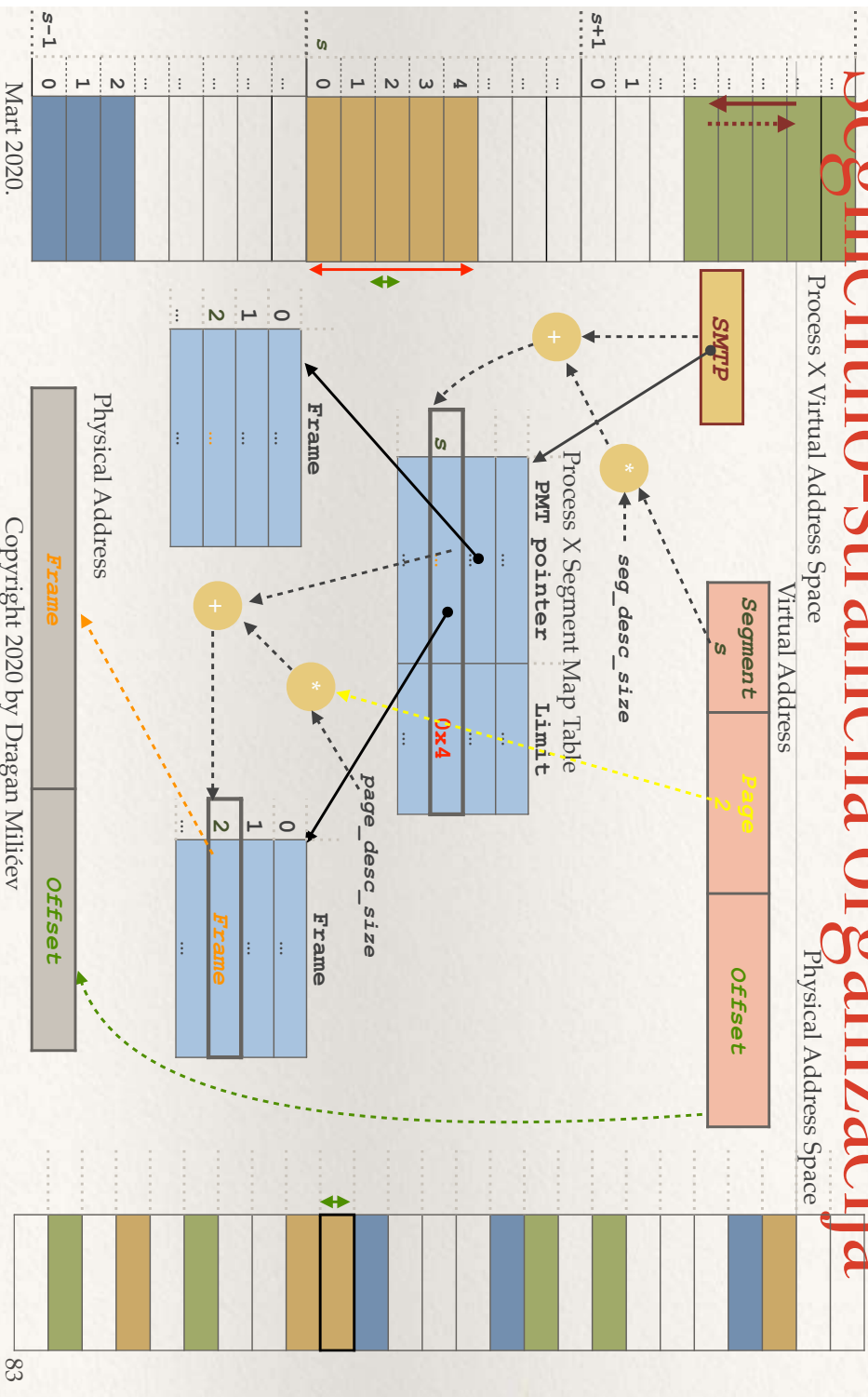
- ❖ Jedna značajna posledica ovakve organizacije jeste ta da virtualni adresni prostor sada može biti isti ili veći, pa i značajno veći od fizičkog, kao i od veličine fizičke memorije: sve dok procesi koriste mali deo svog virtuelnog adresnog prostora i njihovi alocirani segmenti mogu da stanu u prostor predviđen za procese, procesi se mogu izvršavati potpuno transparentno
- ❖ Segmentna organizacija podrazumeva kontinualnu alokaciju svakog pojedinačnog fizičkog segmenta, nezavisno, pa je ova organizacija opštiji slučaj kontinualne alokacije: kontinualna alokacija je specijalan slučaj u kom proces ima samo jedan segment
- ❖ Zbog toga ova organizacija ima sve karakteristike, pa i nedostatke kontinualne alokacije:
 - ❖ OS mora da organizuje i održava (netrivijalnu) strukturu podataka za evidenciju slobodnih fragmenata, prilagođenu operacijama alokacije i dealokacije
 - ❖ OS mora da sprovodi neki algoritam dinamičke alokacije memorije (npr. *first fit* ili *best fit*) za segmente
 - ❖ postoji problem eksterne fragmentacije
- ❖ Problem eksterne fragmentacije je i najozbiljniji problem ove organizacije, jer nakon dužeg rada sistema može dovesti do situacije u kojoj nova alokacija nije moguća bez kompakcije. Kompakcija je dugotrajna operacija koja suspenduje izvršavanje svih procesa
- ❖ Iako se ovaj problem može ublažiti opisanim tehnikama, može se rešiti samo suštinskim uklanjanjem uzroka: odustajanjem od kontinualne alokacije delova različite veličine

Mart 2020.

Copyright 2020 by Dragan Milić

80

Segmentno-stranična organizacija



83

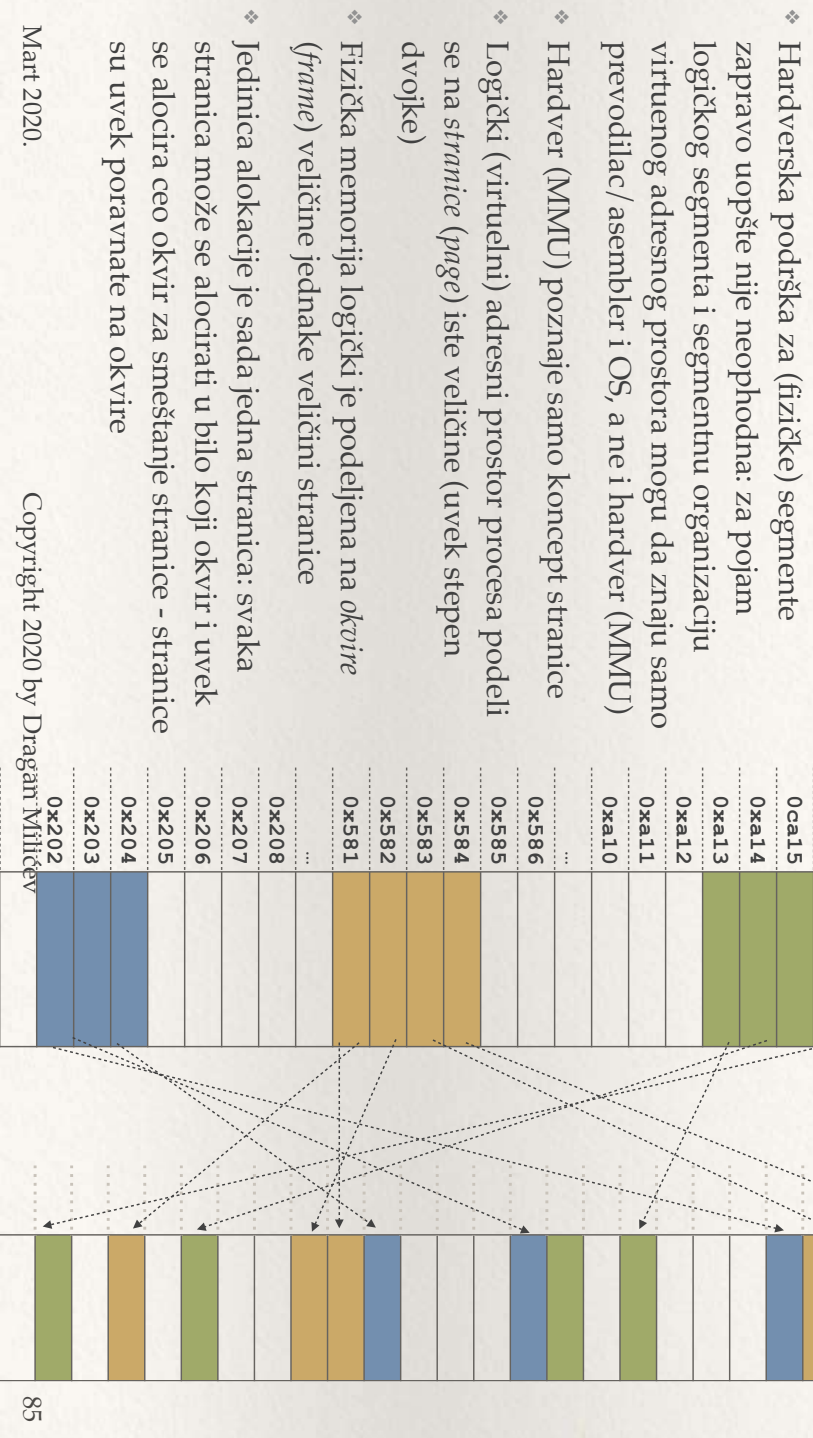
Segmentno-stranična organizacija

Preslikavanje je sada složenije:

- ❖ Virtuelna adresa ima tri polja: broj segmenta, broj stranice unutar segmenta i pomeraj (*offset*, *offset*) unutar stranice
- ❖ Registar *SMTP* procesora ukazuje na SMT tekućeg procesa; sadržaj mu definiše OS pri promeni konteksta
- ❖ SMT svakog procesa sadrži po jedan ulaz (deskriptor) za svaki segment u virtuelnom prostoru, ali taj deskriptor ne sadrži bazu adresu, već samo granicu (*limit*), i to izraženu u broju korišćenih stranica tog segmenta; ukoliko segment nije alocirani, ovaj deskriptor sadrži *null*
- ❖ Za svaki segment svakog procesa postoji *tabela preslikavanja stranica* (*page map table*, PMT). Deskriptor alociranog segmenta u SMT ukazuje na početak PMT za taj segment tog procesa
- ❖ PMT ima po jedan ulaz za svaku stranicu unutar jednog segmenta (deskriptor stranice). Taj ulaz sadrži broj okvira fizičke memorije u koji je stranica smeštena, ili *null* ako ta stranica nije alocirana (izvan je granice segmenta)
- ❖ MMU izračunava adresu deskriptora segmenta na osnovu broja segmenta i vrednosti registra SMTP, dovlači deskriptor segmenta sa te adrese i proverava da li je broj stranice iz virtuelne adrese prekoračio granicu segmenta; ako je granica segmenta prekoračena (broj adresirane stranice je iznad granice), procesor signalizira izuzetak - prekoračenje segmenta
- ❖ Ako je broj stranice unutar granice segmenta, iz deskriptora segmenta dobija se adresa početka PMT. Na osnovu broja stranice i te adrese, izračunava se adresa deskriptora stranice i on dohvata iz memorije
- ❖ Ako je u deskriptoru stranice *null*, procesor signalizira poseban izuzetak, tzv. *straničnu grešku* (*page fault*) - upotreba objašnjena kasnije (inače za ovu konkretnu organizaciju ova provera ne bi morala da se vrši)
- ❖ Iz deskriptora stranice uzima se broj okvira u koji je stranica smeštena. Pošto je stranica iste veličine kao i okvir, pa je pomeraj unutar stranice isti kao i pomeraj u odnosu na početak okvira, pomeraj se samo konkatirira s desne strane na broj okvira da bi se dobila fizička adresa

Stranična organizacija

Process X



Mart 2020.

Copyright 2020 by Dragan Milićev

85

Segmentno-stranična organizacija

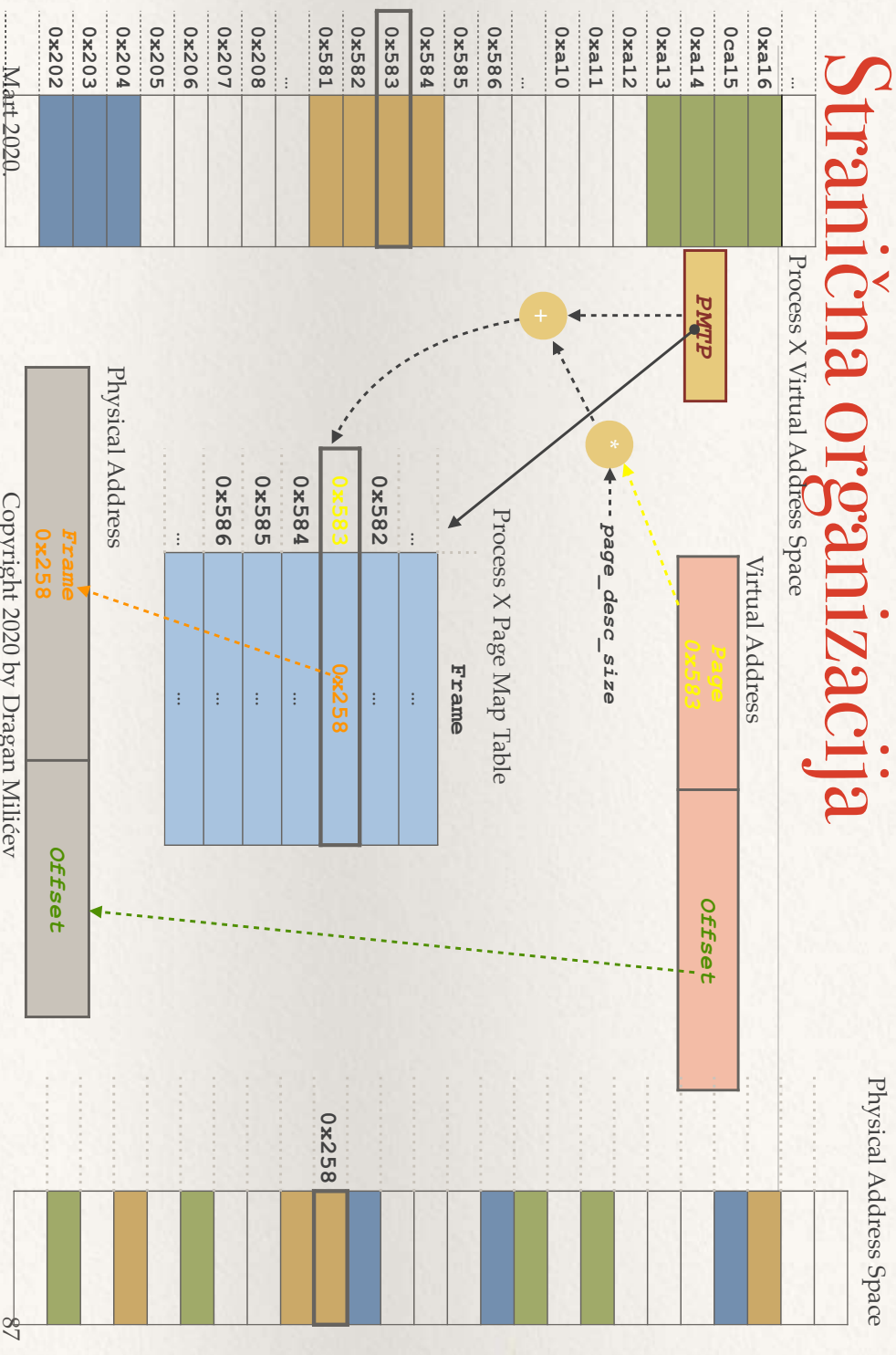
- ❖ OS treba da vodi evidenciju o slobodnim i zauzetim okvirima u memoriji nekom jednostavnom strukturom, npr.:
 - ❖ bit-vektor: po jedan bit za svaki okvir fizičke memorije koji ukazuje na to da li je taj okvir zauzet ili slobodan
 - ❖ ulančana, neuređena lista slobodnih okvira, sa pokazivačima za ulančavanje u samim slobodnim okvirima
- ❖ Kako su stranice iste veličine kao i okviri, a svi okviri ravnopravni u fizičkoj memoriji, bilo koja stranica može se smestiti u bilo koji okvir. Zato je zadatak koji OS ima u pogledu odabira okvira krajnje jednostavan, jer može da izabere bilo koji slobodan okvir (prvi na koji naiđe) - nema algoritma alokacije
- ❖ Nema eksternje fragmentacije, jer se alociraju blokovi iste veličine (stranice, odnosno okviri)
- ❖ Postoji interna fragmentacija: unutar stranice može postojati neiskorišćen deo, ali se taj deo fizičke memorije ne može dodeliti nekom drugom, jer je ceo okvir dodeljen samo jednoj stranici
- ❖ Velika složenost preslikavanja, zahteva složen hardver i ima veliko trajanje:
 - ❖ više pristupa tabelama preslikavanja u memoriji, u dva nivoa (SMT i PMT)
 - ❖ više sabiranja

Mart 2020.

Copyright 2020 by Dragan Milićev

84

Stranična organizacija



87

Stranična organizacija

Preslikavanje je sada jednostavno:

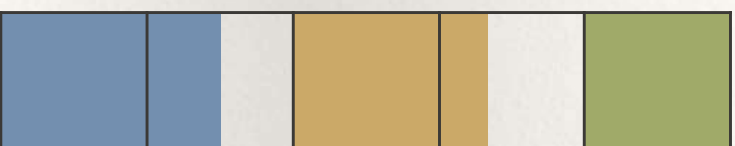
- ❖ Virtuelna adresa ima dva polja: broj stranice i pomeraj (offset, *offset*) unutar stranice
- ❖ Za svaki proces OS organizuje samo *tabelu preslikovanja stranica* (*page map table*, PMT)
- ❖ Registar *PMTP* (*PMT pointer*) procesora ukazuje na PMT tekućeg procesa; sadržaj mu definiše OS pri promeni konteksta
- ❖ PMT ima po jedan ulaz za svaku stranicu celog virtuelnog adresnog prostora (deskriptor stranice). Taj ulaz sadrži *broj okvira* fizičke memorije u koji je stranica smeštena, ili *null* ako ta stranica nije alocirana, npr. zato što je van opsega nekog logičkog segmenta
- ❖ MMU izračunava adresu deskriptora stranice na osnovu broja stranice i vrednosti registra PMTP i dovlači deskriptor stranice sa te adrese operativne memorije
- ❖ Ako je u deskriptoru stranice *null*, procesor signalizira poseban izuzetak, tzv. *straničnu grešku* (*page fault*); OS obrađuje taj izuzetak i podrazumevano gasi proces zbog prestupa u pristupu memoriji
- ❖ Iz deskriptora stranice uzima se broj okvira u koji je stranica smeštena. Pošto je stranica iste veličine kao i okvir, pa je pomeraj unutar stranice isti kao i pomeraj u odnosu na početak okvira, pomeraj se samo konkatenuje s desne strane na broj okvira da bi se dobila fizička adresa

Stranična organizacija

- ❖ I dalje postoji interna fragmentacija: unutar stranice može postojati neiskorišćen deo, ali se taj deo fizičke memorije ne može dodeliti nekom drugom, jer je ceo okvir dodeljen samo jednoj stranici
- ❖ Neiskorišćen deo stranice može biti veličine od nule do (skoro) veličine stranice. Ako se pretpostavi uniformna raspodela pozicije granice segmenta unutar stranice, prosečna veličina internog fragmenta za svaki logički segment je $1/2$ veličine jedne stranice
- ❖ Da bi se povećao stepen iskorišćenosti memorije, treba smanjiti ukupnu veličinu internih fragmenata. To se može postići smanjivanjem veličine stranice (u graničnom slučaju stranica je veličine jedne reči). Međutim, smanjivanje stranice donosi druge negativne posledice
- ❖ Manje stranice uzrokuju:
 - ❖ veće tabele preslikavanja (PMT): ako je virtuelna adresa iste širine, manja stranica znači manje bita za ofset, više za broj stranice, što određuje broj ulaza i veličinu PMT; u graničnom slučaju, veličina jedne stranice je samo jedna memorijska reč - svaka virtuelna adresa preslikava se u posebnu fizičku lokaciju, pa je PMT iste veličine kao i virtuelni adresni prostor, što je potpuno besmisleno
 - ❖ kod učitalavanja (a i snimanja, kasnije) manjih stranica, ulazno-izlazni podsystem je manje efikasan, jer je udeo režijskih troškova u odnosu na količinu prenetog korisnog sadržaja (*payload*) veći (lošiji)
- ❖ Veće stranice uzrokuju:
 - ❖ manje tabele preslikavanja
 - ❖ ulazno-izlazne operacije (učitalvanje i snimanje stranica) su uvek generalno efikasnije za veće blokove prenosa
- ❖ Prema tome, određivanje veličine stranice je pitanje inženjerskog kompromisa (*trade-off*)

Mart 2020.

Copyright 2020 by Dragan Milićev



89

Stranična organizacija

- ❖ Sve dobre karakteristike segmentno-stranične organizacije i dalje ostaju:
 - ❖ OS treba da vodi evidenciju o slobodnim i zauzetim okvirima u memoriji nekom jednostavnom strukturom, npr. bit vektorom ili ulančanom listom
 - ❖ zadatak koji OS ima u pogledu odabira okvira je krajnje jednostavan, jer može da izabere bilo koji slobodan okvir (prvi na koji naiđe), svi su isti i ravnopravni
 - nema algoritma alokacije
 - ❖ nema eksterne fragmentacije, jer se alociraju blokovi iste veličine (stranice, odnosno okviri)
 - ❖ Preslikavanje je sada znatno jednostavnije i efikasnije, hardver je jednostavniji i brži, nema višestrukih pristupa tabelama, nema upotrebe sabirača za izračunavanje fizičke adrese kao kod segmentne i segmentno-stranične organizacije
- ❖ Zbog svega toga se stranična organizacija najčešće primenjuje: podrška hardvera je jednostavna i efikasna, a za OS sasvim dovoljna za sve što mu je potrebno

Mart 2020.

Copyright 2020 by Dragan Milićev

88

Stranična organizacija

- ❖ Odgovor (i rešenje) leži u srečnoj okolnosti da procesi po pravilu koriste vrlo, vrlo mali deo svog virtuelnog adresnog prostora, daleko manji od ukupno raspoloživog
- ❖ Ovo posebno važi za 64-bitne arhitekture: adresni prostor od 2^{64} B = 16 EB (reda 10^{19} bajtova) je ogroman, teško zamisliv i za veliku većinu realnih aplikacija sasvim nepotreban
- ❖ Jedno moguće rešenje:
 - ❖ PMT zauzima samo onoliko prostora, prvih n ulaza, koliko je potrebno za adresiranje prvog dela adresnog prostora (niži deo), onog u kom je proces išta alocirao
 - ❖ MMU treba da zna gde je granica PMT, odnosno koliko je stvarna veličina PMT, ovu informaciju mora da ima u specijalizovanom, programski dostupnom registru koji postavlja OS prilikom promene konteksta (deo memorijskog konteksta procesa), zajedno sa *PMTT*
 - ❖ PMT mora da bude velika toliko da pokrije sve alocirane logičke segmente, tj. njihove stranice, pa može imati puno neiskorišćenih (*null*) ulaza ako su segmenti procesa na velikom rastojanju (sa velikim praznim prostorom između) - PMT bespotrebno zauzima memoriju za te delove
 - ❖ problem je što ovaj pristup ima efekta samo kada proces alocira prvi (niži) deo virtuelnog adresnog prostora, ako alocira makar i jednu jedinu stranicu na vrhu tog prostora, PMT mora da se alocira cela

Mart 2020.

Copyright 2020 by Dragan Milićev

91

Stranična organizacija

1 K - kilo, 2^{10} , približno ali veće od 10^3
1 M - mega, 2^{20} , približno ali veće od 10^6
1 G - giga, 2^{30} , približno ali veće od 10^9
1 T - tera, 2^{40} , približno ali veće od 10^{12}
1 P - peta, 2^{50} , približno ali veće od 10^{15}
1 E - eksa, 2^{60} , približno ali veće od 10^{18}

- ❖ Koliko zapravo može biti velika PMT?
- ❖ Današnje arhitekture imaju virtuelne adrese veličine 32 ili 64 bita; izvedimo računicu za obe
- ❖ Neka je adresibilna jedinica bajt, a veličina stranice 4 KB = 2^{12} B; polje za ofset ima 12 bita
- ❖ Polje za broj stranice ima tako $32 - 12 = 20$, odnosno $64 - 12 = 52$ bita
- ❖ Zbog toga PMT treba da ima $2^{20} = 1$ M ulaza, odnosno čak $2^{52} = 4$ P ulaza, što je *ogroman broj*
- ❖ Ako jedan ulaz u PMT ima samo 32 bita, odnosno 4 B, što omogućava adresiranje (najviše) $2^{32} = 4$ G okvira, onda PMT za *jedan (svaki) proces* treba da zauzima:
 - ❖ $2^{20} \cdot 4$ B = 4 MB za 32-bitnu arhitekturu, što je već mnogo, imajući u vidu to da OS treba da izvršava mnogo procesa (na desetine i stotine)
 - ❖ $2^{52} \cdot 4$ B = 2^{54} B = 16 PB (reda više od 10^{16} bajtova) za 64-bitnu arhitekturu, što je ogroman prostor, daleko veći nego što će tehnologija u skorije vreme uopšte omogućiti
- ❖ Ovo izgleda nepraktično za 32-bitnu arhitekturu, a postaje fizički neizvodljivo za 64-bitnu!?
- ❖ Međutim, današnji sistemi (čak i oni 64-bitni) sasvim dobro funkcionišu. Kako?!

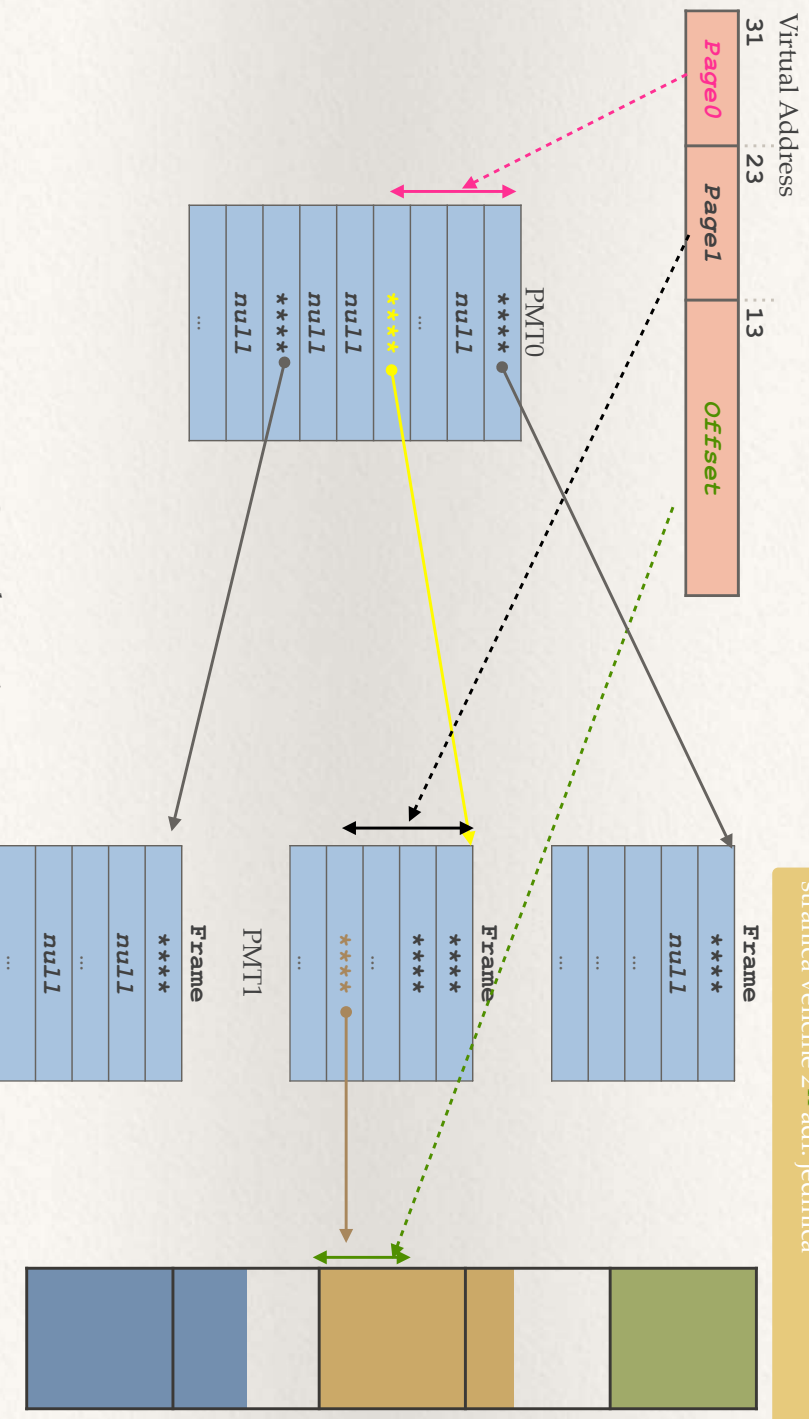
Mart 2020.

Copyright 2020 by Dragan Milićev

90

Stranična organizacija

Primer: PMT u dva nivoa, virtuelna adresa 32-bitna, PMT prvog nivoa ima $256 = 2^8$ ulaza, a PMT drugog nivoa ima 1 K = 2^{10} ulaza. Polje za ofset ima tako $32 - 8 - 10 = 14$ bita, pa je stranica veličine 2^{14} adr. jedinica



Mart 2020.

Copyright 2020 by Dragan Milčević

93

Stranična organizacija

- ❖ Drugo rešenje: upotreba heš tabele (*hash table*); heš tabela je upravo struktura koja rešava problem preslikavanja ključeva u vrednosti (ovde broja stranice u broj okvira), u kom je domen ključeva ogroman (ovde je domen ukupan skup stranica u celom virtuelnom adresnom prostoru i veličine je 2^{20} odnosno 2^{52}), ali je podskup ključeva koji se pojavljuju u stvarnom preslikavanju relativno mali u odnosu na ceo domen (ovde je to skup alociranih stranica)
- ❖ Problem je u tome što ovu heš tabelu treba da organizuje i koristi hardver, tj. MMU, jer je on taj koji vrši preslikavanje, a to nije jednostavan hardverski zadatak i može da bude zahtevan u pogledu efikasnosti i zauzeća prostora na čipu procesora
- ❖ Ideja za rešenje: samu PMT podeliti logički na "stranice", pa je organizovati indeksno, u više nivoa (dva, tri ili više), kao stablo, tako da PMT jednog nivoa predstavlja indeks tabela narednog nivoa itd.
- ❖ Efekat je u sledećem: za oblasti virtuelnog adresnog prostora koje proces uošte nije alocirao, a koje pokriva jedan ceo ulaz u PMT jednog nivoa, PMT narednog nivoa za taj ulaz uopšte ne treba alocirati, taj ulaz ima vrednost *null*
- ❖ Kada MMU preslikava adrese, dohvata najpre odgovarajući ulaz u PMT prvog nivoa; ako je u njemu *null*, generiše izuzetak, jer taj deo virtuelnog adresnog prostora uopšte nije alocirao i PMT sledećeg nivoa ne postoji (kao ni one dalje); ako vrednost nije *null*, sa adrese definisane njom dohvata adresu PMT narednog nivoa, itd.
- ❖ Polje sa brojem stranice u virtuelnoj adresi je sada podeljeno na više polja koja određuju ulaz u PMT svakog nivoa



Slika

Mart 2020.

Copyright 2020 by Dragan Milčević

92

Stranična organizacija

- ❖ Rešenje: u procesoru, kao deo MMU, organizuje se memorija koja je relativno mala po kapacitetu (mnogo manja od operativne memorije), ali brza po vremenu pristupa (jer je deo procesora), koja služi kao keš za deskriptore - tzv. *Translation Lookaside Buffer* (TLB):
 - ❖ sadrži podskup deskriptora koji su nedavno već korišćeni
 - ❖ kada treba da preslika virtuelnu adresu, MMU najpre traži deskriptor u TLB; ako je taj deskriptor već dovučen u TLB kod ranijeg pristupa, odziv će biti brz jer se preslikavanje završava bez dohvatanja deskriptora iz operativne memorije
 - ❖ ako deskriptor nije u TLB, MMU će ga dohvatiti iz operativne memorije, ali i sačuvati u TLB, kako bi sledeći pristupi istom deskriptoru, koji su vrlo verovatni zbog pristupa istoj stranici, biti ubrzani

Mart 2020.

Copyright 2020 by Dragan Milićev

95

Stranična organizacija

- ❖ Problem: prilikom *svakog* preslikavanja virtuelne u fizičku adresu, što se potencijalno dešava više puta tokom izvršavanja jedne instrukcije, procesor mora da dohvati (ciklusom čitanja) deskriptor stranice iz PMT, i to čak iz više ciklusa čitanja, ako deskriptor ne može da se prenese u jednom ciklusu ili ako je PMT organizovana u više nivoa
- ❖ Ovo znači da se za jedan *efektivan* pristup virtuelnoj memoriji vrši dva ili više pristupa fizičkoj memoriji, što višestruko usporava rad procesora sa memorijom - efektivno vreme pristupa memoriji je višestruko veće od realnog vremena odziva fizičke memorije. Ovo je cena koja ne bi bila prihvatljiva čak ni za tako koristan mehanizam kao što je virtuelna memorija
- ❖ Da li bi PMT mogla da se smesti u sam procesor, kako bi joj pristup bio znatno brži (bez pristupa operativnoj memoriji)? Neizvodljivo ili nepraktično i neefikasno, jer:
 - ❖ prostor za smeštanje PMT u procesoru bi morao da bude dovoljno velik (reda megabajta za 32-bitne arhitekture), što može biti neizvodljivo čak i za 32-bitne arhitekture
 - ❖ prilikom promene konteksta, sadržaj ovog "registarskog fajla" bi morao da se učita iz memorije, što zbog njegove veličine može da bude neprihvatljivo dugotrajno, dok se zapravo većina tog sadržaja možda neće uopšte koristiti u sledećem naletu izvršavanja tog procesa

Mart 2020.

Copyright 2020 by Dragan Milićev

94

Stranična organizacija

- ❖ TLB ni na koji način ne menja semantiku preslikavanja, već ga samo ubrzava, pa bi zato, u principu, bio potpuno transparentan za softver (i jeste za procese), osim jednog detalja
- ❖ TLB može da realizuje preslikavanja potrebnog ključa iz virtuelne adrese u deskriptor; taj ključ je:
 - ❖ broj segmenta kod segmentne organizacije
 - ❖ broj segmenta i broj stranice kod segmentno-stranične organizacije
- ❖ Međutim, ovo preslikavanje definisano je za *jedan* proces i izgleda potpuno drugačije za neki drugi proces, jer se isti broj segmenta ili stranice različitih procesa preslikava u različite deskriptore
- ❖ Zbog toga ceo sadržaj TLB-a ima samo opseg važenja tekućeg procesa, pa se mora *invalidovati* (proglasiti nepostojećim, nevalidnim, "obrisati") prilikom promene konteksta
- ❖ Ovo mora da uradi OS kada vrši promenu konteksta posebnom instrukcijom koju procesor mora da obezbedi za tu svrhu
- ❖ Druga mogućnost jeste ta da TLB sadrži deskriptore različitih procesa, uz koje pamti i informaciju o tome kom preslikavanju (tabeli preslikavanja) pripada svaki deskriptor
- ❖ Tada se kao deo ključa mora koristiti i *SMT/PMT/PMT*, kao identifikator procesa, odnosno identifikator samog preslikavanja za koje se traži deskriptor. U ovom slučaju je TLB potpuno transparentan i za OS

Mart 2020.

Copyright 2020 by Dragan Milicev

97

Vreme pristupa u slučaju pogotka: jedan pristup TLB-u (20 ns) i jedan pristup adresiranoj lokaciji (100 ns)

Stranična organizacija

Vreme pristupa u slučaju promašaja: jedan pristup TLB-u (20 ns), jedan pristup PMT (100 ns) i jedan pristup adresiranoj lokaciji (100 ns)

- ❖ Kako TLB rešava problem trajanja preslikavanja? Izvodimo u pored. proračun za slučaj kada TLB ne postoji i kada postoji, i to za slučaj PMT organizovane u jednom i u tri nivoa
- ❖ Neka je vreme pristupa operativnoj memoriji 100 ns, a TLB-u 20 ns; za zaključak proračuna zapravo nisu bitne ove apsolutne vrednosti, već samo njihov relativan odnos, koji zaista jeste takav da je pristup TLB-u više puta brži nego pristup operativnoj memoriji
- ❖ Neka je *učestvost pogotka* u TLB 95% (udeo broja slučajeva kada traženi deskriptor jeste u TLB u odnosu na ukupan broj pretraga); ovo je realna pretpostavka, jer se keševi prave tako da imaju visok procenat pogotka (preko 90%)
- ❖ Efektivno vreme pristupa za slučaj bez TLB-a:
 - ❖ za PMT u jednom nivou: $2 \cdot 100 \text{ ns} = 200 \text{ ns}$, jedan za pristup PMT i jedan za pristup adresiranoj lokaciji u memoriji
 - ❖ za PMT u tri nivoa: $4 \cdot 100 \text{ ns} = 400 \text{ ns}$, tri za pristup tabelama u tri nivoa i jedan za pristup adresiranoj lokaciji u memoriji
- ❖ Zaključak: efektivno vreme raste brzo, proporcionalno broju nivoa i višestruko je veće od vremena pristupa fizičkoj memoriji
- ❖ Efektivno vreme pristupa za slučaj sa TLB-om:
 - ❖ za PMT u jednom nivou: $0,95 \cdot 120 \text{ ns} + 0,05 \cdot 220 \text{ ns} = 125 \text{ ns}$, što je povećanje od 25%
 - ❖ za PMT u tri nivoa: $0,95 \cdot 120 \text{ ns} + 0,05 \cdot 420 \text{ ns} = 135 \text{ ns}$, što je povećanje od 35%
- ❖ Zaključak: efektivno vreme pristupa je prihvatljivo i za straničenje u više nivoa jer je "penal" relativno mali u odnosu na realno vreme pristupa memoriji, i raste sporo sa porastom broja nivoa straničenja; ovo je posledica toga što je ukupno vreme pristupa za slučaj pogotka relativno malo povećano u odnosu na realno vreme pristupa memoriji (za trajanje pristupa TLB-u) ponderisano velikim težinskim faktorom (procenat pogotka), dok je trajanje koje je veliko i koje brzo raste sa brojem nivoa u slučaju promašaja ponderisano vrlo malim faktorom (procenat promašaja)

Opet samo jedan pristup TLB-u, jer TLB odmah vraća deskriptor tražene stranice

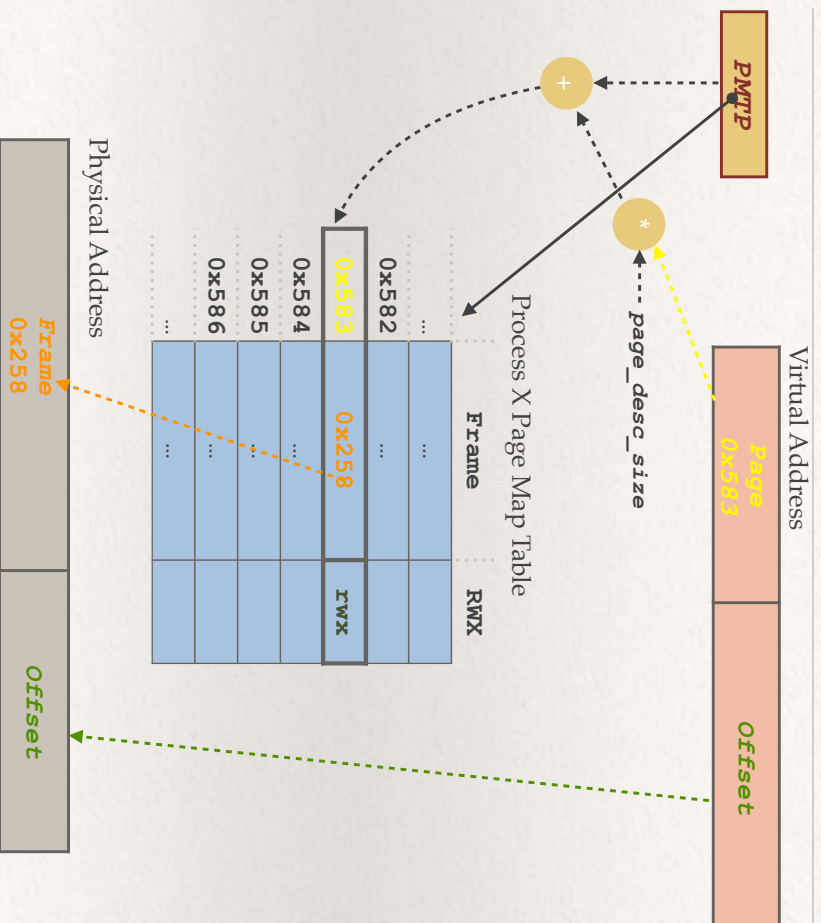
- ❖ Zaključak: tehnika preslikavanja virtuelne u realnu adresu preko tabela preslikavanja (segmentata ili stranica, svedeno) *nije nikako* efikasna bez TLB-a, a postaje sasvim praktična, izvodljiva i efikasna sa TLB-om. Zato je TLB obavezan element procesora

Mart 2020.

Copyright 2020 by Dragan Milicev

96

Zaštita



Mart 2020.

Copyright 2020 by Dragan Milicev

99

Zaštita

- ❖ Postoji potreba da se proces zaštiti od samog sebe, tj. od sopstvenih instrukcija, odnosno od sopstvenih grešaka zbog korupcije:
 - ❖ da ne pristupa nealociranim delovima virtuelnog adresnog prostora - podiška je već opisana
 - ❖ da ne menja sopstvene instrukcije ili podatke koji su namenjeni samo za čitanje
 - ❖ da stek ne prekorači svoju granicu i ne "pregazi" ostale podatke ili instrukcije
- ❖ Kako je proveru na navedene prestupe potrebno vršiti pri svakom adresiranju, neophodna je podiška hardvera:
 - ❖ u deskriptoru fizičkog segmenta (kod segmentne i segmentno-stranične organizacije), odnosno stranice (kod stranične organizacije, nalazi se i informacija o *pravinima pristupa* do datog fizičkog segmenta (ili svih stranica u njemu), odnosno stranice:
 - ❖ X (*execute*): dozvoljeno je "izvršavanje" sadržaja, tj. pristup u ciklusima čitanja, ali samo tokom faze dohvaćanja instrukcije u procesoru - važi za programski kod, odnosno logičke segmente sa instrukcijama
 - ❖ R (*read*): dozvoljeno je čitanje sadržaja, tj. pristup u ciklusima čitanja, ali samo tokom faze izvršavanja instrukcije u procesoru (dohvaćanja operanada) - važi za podatke
 - ❖ W (*write*): dozvoljen je upis sadržaja, tj. pristup u ciklusima upisa, ali samo tokom faze izvršavanja instrukcije u procesoru (upis rezultata) - važi za podatke
 - ❖ pri svakom preslikavanju virtuelne u fizičku adresu, MMU najpre proverava da li je tip ciklusa koji se traži (čitanje ili upis), u fazi izvršavanja instrukcije u kom se taj ciklus traži (MMU je deo procesora, pa to zna) u skladu sa dozvolama navedenim u deskriptoru; ako nije, generiše izuzetak - pristup kod pristupa memoriji (*memory access violation*); ako jeste, obavlja preslikavanje kako je opisano

Mart 2020.

Copyright 2020 by Dragan Milicev

98



Zaštita

- ❖ Zaštita procesa od međusobnog uticaja, tj. od toga da jedan proces neovlašćeno pristupi memorijskom sadržaju drugog procesa, je inherentno podržana svim tehnikama virtuelne memorije: osim ako procesi s namerom ne dele memoriju, njihovi virtuelni adresni prostori su preslikani u različite delove fizičke memorije, pa jedan proces ni na koji način ne može da pristupi memorijskom sadržaju koji pripada drugom procesu
- ❖ Da bi sam kernel pristupio svim potrebnim delovima fizičke memorije kako bi u njima obavljao neophodne operacije, kernel može da upiše bilo koju vrednost u bazni registar (kod kontinualne organizacije), odnosno da organizuje svoju SMT / PMT koju koristi za pristup memoriji, a čije ulaze može da upiše bilo koje bazne adrese / brojeve okvira po potrebi, kako bi pristupao potrebnim delovima fizičke memorije
- ❖ Međutim, kako sprečiti da neki proces uradi to isto? Šta ako neki proces (greškom ili zlonamerno) pristupi tabelama preslikavanja, ili izvrši instrukciju kojom u bazni odnosno registar *SMTT / PMTP* upiše neku proizvoljnu adresu, recimo fizičku adresu koja ukazuje na neku tabelu koju je on sam organizovao u memoriji? Na taj način bi taj proces mogao potpuno da preuzme kontrolu nad memorijom i da pristupi tuđim delovima memorije, onima koji pripadaju drugim procesima ili samom kernelu
- ❖ Jasno je da instrukcija upisa u *SMTT / PMTP* ne sme biti dozvoljena za izvršavanje od strane korisničkog procesa, odnosno kada procesor izvršava instrukciju u kontekstu nekog procesa (tačnije, ovaj registar ne sme biti dostupan)
- ❖ Ali kako da onda ta ista instrukcija bude dozvoljena tokom izvršavanja koda kernela? Kako da procesor zna šta trenutno izvršava?

Mart 2020.

Copyright 2020 by Dragan Miličev

101

Zaštita

prot je ceo broj čiji biti definišu prava pristupa. Radi lakšeg korišćenja, za maske odgovarajućih bita definisane su simboličke konstante (makroi):
PROT_EXEC - stranice se mogu izvršavati;
PROT_READ - stranice se mogu čitati;
PROT_WRITE - stranice se mogu upisivati;
PROT_NONE - stranice nisu dostupne.
Argument *prot* može biti *PROT_NONE* ili rezultat bit-po-bit OR operacije ostalih konstanti (npr. *PROT_READ | PROT_WRITE*)

- ❖ Informaciju o pravima pristupa za svaki logički segment, na osnovu koje upisuje vrednosti u deskriptor fizičkog segmenta odnosno stranice, OS dobija iz *exe* fajla (tu) je upisao prevodilac ili assembler, na osnovu vrste sadržaja logičkog segmenta) kod statičke alokacije segmenta, odnosno iz posebnog parametra sistemskog poziva za dinamičku alokaciju segmenta. Sistemski poziv za dinamičku alokaciju logičkog segmenta sada izgleda ovako:
`void* mmap (void* base_addr, size_t size, int prot);`
- ❖ Kako bi opisani mehanizam imao efekta i kod straničnog preslikavanja, cela stranica mora imati istorodan sadržaj, sa istim pravima pristupa; zato se logički segmenti koje formira prevodilac, definiše programer na assembleru, ili alocira proces, uvek definišu sa istim pravima pristupa, odnosno sa istom vrstom sadržaja (npr. nikada se ne stavljaju podaci i kod u isti logički segment), i uvek počinju od početka nove stranice (kao poglavlje u knjizi); nikada se ne počinje nov segment na istoj stranici nekog drugog segmenta
- ❖ Šta ako je odmah iza kraja segmenta sa stekom (u pravcu rasta steka) alociran segment za podatke? Oba segmenta su dozvoljena za upis i MMU neće detektovati situaciju u kojoj stek prekoračuje svoju granicu (jer generisana virtuelna adresa na koju ukazuje *SP* prelazi u susedan segment sa podacima)
- ❖ Ovo se rešava jednostavno tako što se u memorijskoj mapi virtuelnog adresnog prostora procesa, iza granice segmenta odvojenog za stek, uvek odvoji jedan mali prostor (jedna stranica) koji je nealociran i do kog nije dozvoljen nikakav pristup. Ako proces prekorači stek, odnosno generiše adresu preko granice segmenta steka, ta adresa upašće u ovaj prostor i MMU će generisati izuzetak

Mart 2020.

Copyright 2020 by Dragan Miličev

100

Zaštita

- ❖ Prelazak iz korisničkog (neprivilegovanog) u sistemski (privilegovani) režim vrši se kod sistemskog poziva i obrade hardverskog izuzetka; tada procesor treba da pređe na izvršavanje instrukcija koda kernela koje izvršavaju zahtev zatražen tim sistemskim pozivom ili obrađuju taj izuzetak
- ❖ Prema tome, instrukcija koja menja režim procesora u privilegovani mora ujedno i da izvrši skok na neku adresu na kojoj se sigurno nalazi kernel kod koji izvršava uslugu koji je ta instrukcija tražila, ili obrađuje izuzetak koji je ona izazvala
- ❖ Da li adresa na koju instrukcija koja vrši sistemski poziv skače može biti definisana eksplicitno u samoj instrukciji, kao operand te instrukcije, nekim od navedenih načina adresiranja, kao što je to slučaj kod uobičajenog poziva potprograma? Na primer ovako:
syscall address
Pretpostavka je da instrukcija sistemskog poziva radi isto što i instrukcija poziva potprograma, samo još menja režim rada procesora u privilegovani
- ❖ Svakako ne, jer bi onda proces u svojoj instrukciji mogao da navede bilo koju adresu na kojoj se nalazi proizvoljan sadržaj, i tako ponovo pređe u privilegovani režim i preuzme kontrolu nad sistemom - opet nije nikakva zaštita
- ❖ Osim toga, za izuzetke adresa skoka svakako mora biti implicitno određena, a ne u samoj instrukciji

Mart 2020.

Copyright 2020 by Dragan Milićev

103

Zaštita

Današnji procesori podržavaju više režima rada, odnosno različitih nivoa zaštite i pravila prelaska izvršavanja iz jednog u drugi. Međutim, operativnom sistemu su dovoljna dva. Kako bi bili lakše prenosivi, operativni sistemi uglavnom i koriste samo dva režima rada, kako je ovde opisano

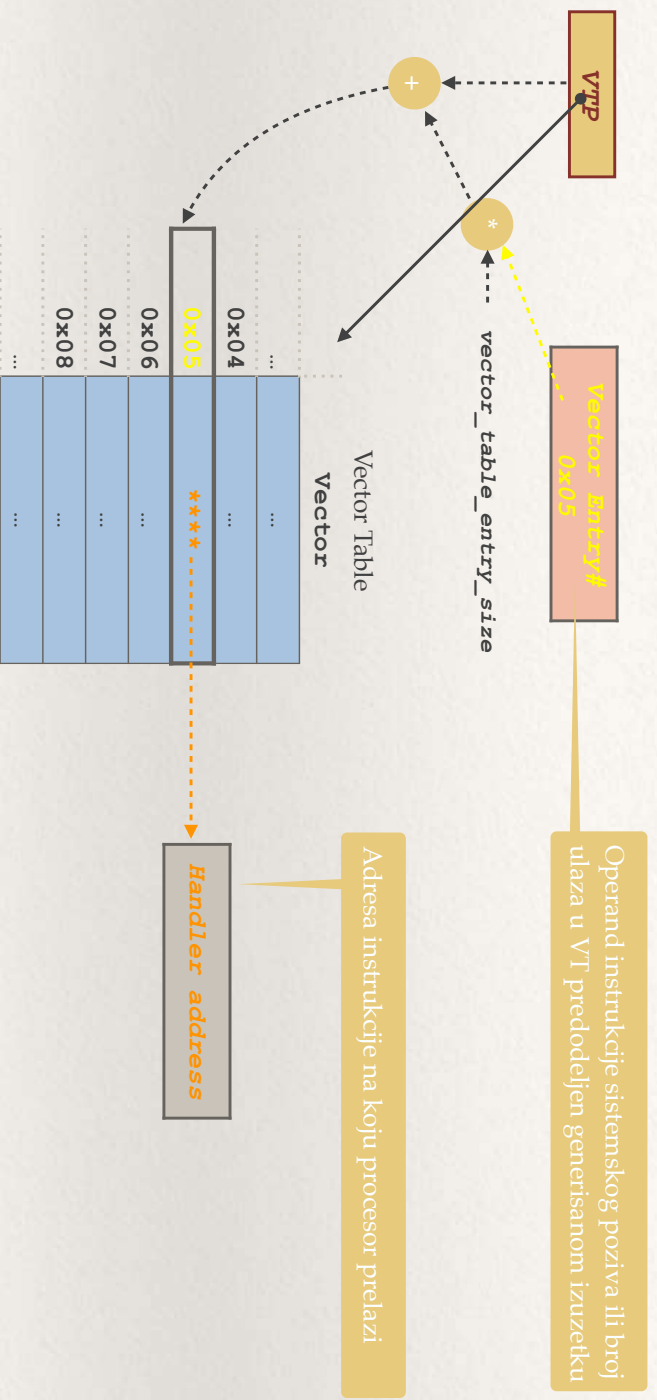
- ❖ Rešenje: procesor mora da podrži (najmanje) dva režima rada:
 - ❖ *privilegovani (privileged)* ili *sistemski, kernel režim (mode)*: kada je u ovom režimu, procesor dozvoljava sve instrukcije i pristup do svih programski dostupnih registara; kernel kod se izvršava u ovom režimu
 - ❖ *neprivilegovani (non-privileged)* ili *korisnički (user)* režim: u ovom režimu neki programski dostupni registri nisu dostupni instrukcijama, tj. instrukcije ne smeju da im pristupe, kao da ti registri ne postoje (ako procesor naiđe na instrukciju koja u svom zapisu ima kod operacije ili kod registra koji nije dozvoljen, generisaće izuzetak zbog nedozvoljene instrukcije); u ovom režimu izvršava se kod korisničkih procesa
- ❖ Kada procesor prelazi iz neprivilegovanog (korisničkog) u privilegovani (kernel) režim?
 - ❖ pri sistemskom pozivu, kada korisnički proces traži neku uslugu kernela
 - ❖ kada instrukcija koju procesor izvršava generiše bilo koji izuzetak
- ❖ Kako procesor prelazi iz jednog režima u drugi?
- ❖ Prelazak iz sistemskog (privilegovanog) u korisnički (neprivilegovani) režim je benignan i može ga izvršiti i kod procesa (bez efekta), kao i kernel, zato se može vršiti nekom instrukcijom koja je dostupna barem u sistemskom režimu
- ❖ Da li se prelazak iz korisničkog u kernel režim može vršiti instrukcijom, koja svakako mora biti dostupna u korisničkom režimu (jer procesor treba da pređe iz korisničkog režima u kernel režim), a koja samo menja režim i ne radi ništa drugo (već procesor prelazi na instrukciju odmah iza nje)?
- ❖ Ne, naravno, jer bi onda svaki proces mogao jednostavno da promeni režim i onda izvršava proizvoljne instrukcije - nije nikakva zaštita

Mart 2020.

Copyright 2020 by Dragan Milićev

102

Zaštita



Mart 2020.

Copyright 2020 by Dragan Milićev

105

Zaštita

- ❖ Zato instrukcija kojom se izvršava sistemski poziv mora da izvrši *implicitni skok*, *memorijskim indirektnim adresiranjem* (*memory indirect address mode*), preko adrese koja je sadržana negde u memoriji, zapravo preko pokazivača na kod potprograma na koji se skače, a ne direktnim adresiranjem određištá skoka
- ❖ Zbog toga procesori koriste sledeći mehanizam izvršavanja sistemskih poziva, ali isto tako i reakcije na izuzetke koje procesor generiše, pošto i njih treba da obradi kod kernela:
 - ❖ negde u memoriji kernel organizuje posebnu strukturu, *vektor tabelu* (*vector table*), VT, koja ima po jednu adresu (vektor, pokazivač na lokaciju) koda kernela koji obavlja određenu operaciju, npr. sistemski poziv ili obradu izuzetka
 - ❖ za svaki tip izuzetka koji može generisati (npr. nelegalan kod operacije, nelegalan način adresiranja, aritmetičko prekoracenje, pristup u pristupu memoriji, stranična greška itd.), hardver procesora pridružuje (predefinisano, nepromenljivo) po jedan broj ulaza u vektor tabeli
 - ❖ instrukcija sistemskog poziva kao svoj operand takode ima broj ulaza u VT, različiti sistemski pozivi mogu koristiti različite ulaze u VT, a mogu i svi koristiti isti, pa se tip poziva može zadati nekim obaveznim parametrom poziva:
syscall #entry_number — **Operand je neposredan, u samoj instrukciji - broj ulaza u VT**
 - ❖ da bi mogao da pronađe adresu koda koji obrađuje izuzetak ili sistemski poziv, procesor mora imati informaciju o adresi početka VT u memoriji, kao i ranije, nju ima u posebnom programskom dostupnom registru, *pokazivač na tabelu vektora* (*vector table pointer, VTP*); vrednost u ovaj registar upisuje OS prilikom inicijalizacije sistema, kada kreira i inicijalizuje VT, naravno, instrukcija za upis u VTP mora biti dostupna samo u privilegovanom režimu, kako neki proces ne bi mogao da uradi isto što i OS i preotme kontrolu nad sistemom
 - ❖ kada izvrši instrukciju sistemskog poziva ili generiše izuzetak sa određenim brojem, hardver procesora dohvata vektor iz memorije, iz ulaza u VT sa tim brojem, i dohvaćenu adresu koristi kao adresu određenog potprograma u koji viši skok

Mart 2020.

Copyright 2020 by Dragan Milićev

104

Zaštita

- ❖ Zbog ovoga procesori po pravilu kod ovakvih skokova, pri obradi izuzetka ili sistemskog poziva, vrednosti registara čuvaju (i kasnije odatele restauriraju) na nekom drugom mestu, koje je garantovano dostupno i ne može napraviti ovakav problem. Neki mogući pristupi su sledeći:
 - ❖ kada prelazi u kernel režim, procesor uopšte ne menja registre koje koristi u nepriviligovanom režimu (ostavlja njihove vrednosti nepromenjenim), već prelazi na korišćenje registara koje koristi u priviligovanom režimu ("sistemski" *PC*, *SP*, *PSW*) i koji nisu dostupni u korisničkom režimu; instrukcije koje se izvršavaju u priviligovanom režimu implicitno koriste ove registre, dok instrukcije koje rade u korisničkom režimu koriste "korisničke" registre; naravno, ovi "sistemski" su potpuno nevidljivi (nedostupni) u korisničkom režimu; tako "korisnički" registri ostaju nepromenjeni tokom obrade instrukcije izuzetka ili sistemskog poziva, pa ih kernel potom može sačuvati gde je potrebno (oni su svakako dostupni u kernel režimu)
 - ❖ kada prelazi u kernel režim, procesor sačuva registre ne na steku, već u posebnim, za to specijalizovanim svojim registrima koji su dostupni samo u kernel režimu, pa ih kernel može dalje sačuvati gde je potrebno
 - ❖ kada prelazi u kernel režim, procesor sačuva ove registre na posebnom, sistemskom steku, tj. na vrhu steka na koji ukazuje poseban registar, "sistemski" *SP*, koji je dostupan i koristi se samo u kernel režimu; kernel alocira ovaj stek u svom prostoru, koji samo on koristi i koji je dovoljno velik za sve što kernel može da radi (dovoljan je za najdublje ugnežđivanje poziva potprograma u kernelu) i garantovano neće biti prekoračen, jer kernel nema neograničene rekurzije u svom kodu

Mart 2020.

Copyright 2020 by Dragan Milićev

107

Zaštita

- ❖ Semantika ovakve instrukcije, kao i obrade izuzetka je po pravilu takva da, osim implicitnog prelaska u priviligovani režim, procesor skače u potprogram kao i kod instrukcije skoka u potprogram: čuva određene programski dostupne registre procesora na steku (svakako *PC*, najčešće *PSW*, a u nekim slučajevima i neke druge) i u registar *PC* smešta vrednost dohvaćenog vektora
- ❖ Povratak iz ovakve obrade vrši se po pravilu posebnom instrukcijom povratka iz sistemskog poziva, koja radi isto što i instrukcija povratka iz potprograma, osim što još i prebacuje procesor u nepriviligovani režim
- ❖ Problem: šta ako je registar *SP* dostigao vrednost koja je na samoj granici prostora alociranog za stek procesa, pa prilikom obrade izuzetka ili sistemskog poziva vrednosti registara koji se ovako implicitno čuvaju ne mogu da se smeste u memoriju, jer preslikavanje u fizičke adrese nije moguće (generisalo bi izuzetak)?

Mart 2020.

Copyright 2020 by Dragan Milićev

106

Zaštita

- ❖ Preostaje još sledeće pitanje: kako sprečiti da neki proces pristupi vektor tabeli i u nju upiše proizvoljne vrednosti, pa time ponovo probije sistem zaštite i preotme kontrolu nad sistemom tako što će promeniti neki vektor, preusmeriti ga na neki svoj (potencijalno maliciozan) kod i onda izazvati sistemski poziv i prelazak procesora u privilegovani režim?
- ❖ Vektor tabelu kernel formira u posebnom delu operativne memorije koji je samo pod njegovom kontrolom i samo njemu dostupan, a nedostupan korisničkim procesima - potrebna je zaštita prostora kernela od korisničkih procesa
- ❖ Kako inače zaštititi celu oblast koju koristi kernel od pristupa procesa, kako proces ne bi poremetio sadržaj memorije kernela?
- ❖ Jedan mogući pristup:
 - ❖ deo fizičke memorije koju koristi kernel, kao i drugi rezervisani i zaštićeni delovi memorije (ROM, memorijski preslikan U/I itd.) se uopšte ne mapiraju u virtuelne adresene prostore procesa: zapravo proces ne može nikako pristupiti tom delu, jer se nijedna njegova virtuelna adresa ne preslikava u taj deo
 - ❖ da bi sam kernel pristupio svom prostoru, prilikom prelaska u kernel režim, kernel menja memorijski kontekst menjajući *SMTP/PMTP* tako da ukazuje na tabelu koja preslikava virtuelnu adresu u identičnu fizičku adresu, kako bi kernel zapravo neposredno video fizičku memoriju i nju adresirao direktno; alternativa je da sam procesor ne viši preslikavanje adresa u kernel režimu
- ❖ Problem ovog rešenja je u tome što kernel mora da preslikava svaki parametar sistemskog poziva koji predstavlja pokazivač (adresu), kao i sve druge pokazivače koje proces koristi, a kernelu su potrebni, u fizičke adrese, jer ti pokazivači predstavljaju virtuelne adrese samo tog procesa

Mart 2020.

Copyright 2020 by Dragan Milićev

109

Zaštita

- ❖ Kako se prenose parametri u sistemski poziv? Mogući pristupi:
 - ❖ kroz registre opšte namene procesora: svaki sistemski poziv očekuje određene parametre u određenim registrima; jedan od njih može da nosi identifikaciju tipa sistemskog poziva, ukoliko svi sistemski pozivi koriste isti ulaz u VT
 - ❖ preko steka korisničkog procesa, odakle ih kernel može pročitati
 - ❖ u nekoj strukturi u memoriji, u kojoj kernel očekuje parametre složene po određenom formatu; adresa te strukture prenosi se u određenom registru (parametar sistemskog poziva je zapravo pokazivač na strukturu sa stvarnim argumentima)

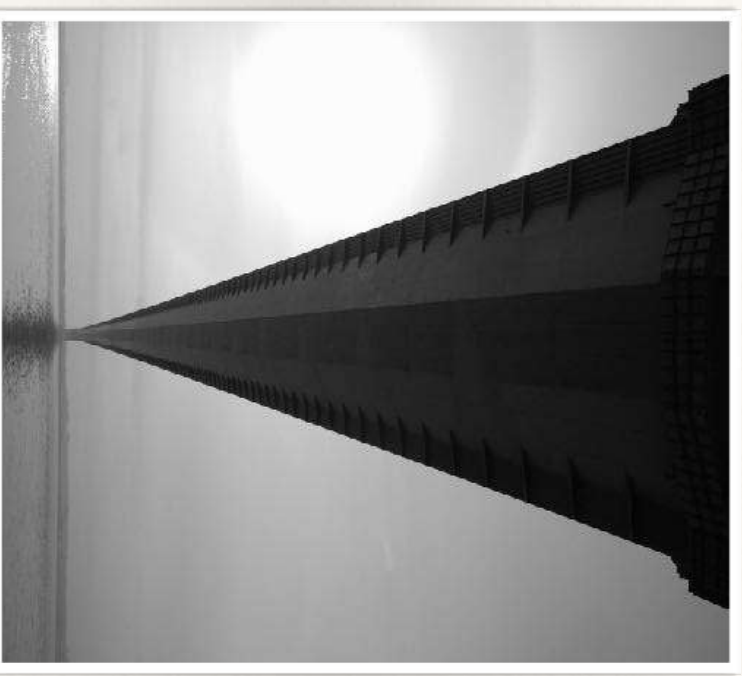
Mart 2020.

Copyright 2020 by Dragan Milićev

108

Glava 6: Deljenje memorije

- ❖ Tehnike deljenja memorije
- ❖ Dinamičko učitavanje
- ❖ Preklopi
- ❖ Logičko deljenje memorije
- ❖ Deljene biblioteke
- ❖ Zamena procesa
- ❖ Učitavanje stranica na zahtev
- ❖ Zamena stranica



Mart 2020.

Copyright 2020 by Dragan Miličev

111

Zaštita

Kod organizacije prikazane na slici, kernel i drugi rezervisani delovi su na najnižim adresama fizičkog adresnog prostora, ali su u virtuelnom adresnom prostoru svakog prostora na njegovom vrhu, dok proces vidi svoj deo prostora počev od adrese 0, što olakšava korišćenje

❖ Drugo rešenje je sledeće:

- ❖ kernel mapira svoj deo memorije u virtuelni adresni prostor svakog procesa, na isto mesto, ali označava taj prostor (stranice ili segmente) kao nedozvoljene za bilo kakav pristup (čitanje, upis ili izvršavanje) u nepriviligovanom režimu rada procesora
- ❖ procesor mora da podrži ovakav koncept tako što u deskriptoru stranice/segmenta ima informaciju o pravu pristupa u privilegovanom režimu (dovoljan je samo jedan bit) i ne dozvoljava pristup (generiše izuzetak) ako pristup nije dozvoljen, a instrukcija pokušava pristup toj stranici u nepriviligovanom režimu

❖ Na ovaj način:

- ❖ svaki proces zapravo ima kernel prostor na istom mestu u svom virtuelnom adresnom prostoru, ali ne može nikako da mu pristupi, "ne vidi ga" (zapravo "vidi" kao zauzet deo svog virtuelnog prostora koji ne može nikako da koristi)
- ❖ kernel ne menja memorijski kontekst (vrednost *PMTT*/*SMTP*) sve dok to ne mora: sve dok izvršava delove koda kojima to ne smeta, koristi isti kontekst u kom se izvršavao proces koji je izvršio sistemski poziv, i tako pristupa celom virtuelnom prostoru procesa
- ❖ tek kada želi da pristupi delovima fizičke memorije koji pripadaju drugim procesima ili su slobodni, prebaci memorijski kontekst na preslikavanje koje to omogućava
- ❖ kada predje procesor drugim procesu, u *PMTT*/*SMTP* upiše vrednost koja odgovara tom procesu

Mapa fizičkog adresnog prostora
Mapa virtuelnog adresnog prostora svakog procesa



Mart 2020.

Copyright 2020 by Dragan Miličev

110

Dinamičko učitavanje

- ❖ Programi vrlo često ispoljavaju sledeće ponašanje: neki delovi koda programa ili podataka, npr. neki potprogrami ili strukture podataka, koriste se vrlo retko, samo u nekim posebnim situacijama koje se retko dešavaju, npr. neki retki specijalni slučajevi obrade, obrada grešaka itd.
- ❖ Ako se tokom nekog izvršavanja programa (kao procesa) takva situacija uopšte ne desi, ovakvi delovi koda ili podataka se uopšte ne koriste; zašto bi onda uopšte zauzimali memoriju bespotrebno, tj. zašto za te delove procesa uopšte alocirati memoriju?
- ❖ Tehnika *dinamičkog učitavanja* (*dynamic loading*) podrazumeva da proces alocira i iz fajla učitava ovakve delove samo ako su stvarno potrebni, i onda kada su potrebni, odnosno kada se takva situacija zaista i dogodi
- ❖ Odgovornost za pokretanje alokacije prostora i učitavanja iz fajla je isključivo na procesu, tj. na programu koji se izvršava, dok OS o tome ne vodi računa
- ❖ Obaveza OS-a je samo da obezbedi uslugu (sistemski poziv) koji alocira deo (virtuelnog) adresnog prostora procesa, kao i uslugu kojom u dati prostor procesa učitava sadržaj iz nekog binarnog fajla (uobičajena sistemka usluga za rad sa fajlovima); OS zapravo ne zna za šta se te usluge koriste, tj. ne zna da se one upotrebljavaju baš za dinamičko učitavanje delova procesa

Mart 2020.

Copyright 2020 by Dragan Milićev

113

Tehnike deljenja memorije

- ❖ U multiprocesnom sistemu u operativnu memoriju treba smestiti više procesa, poželjno je što više, pa i neograničen broj
- ❖ Da bi to bilo moguće, potrebno je koristiti tehnike koje omogućavaju da:
 - ❖ proces koristi što je moguće manje memorije
 - ❖ procesi dele memoriju
- ❖ Deljenje memorije, kao hardverskog resursa, moguće je na sledeće načine:
 - ❖ prostorno: jedan deo resursa (memorije) koristi jedan proces, drugi deo drugi procesi, ili isti deo koristi više procesa
 - ❖ vremenski: u jednom intervalu vremena jedan isti deo memorije koristi jedan proces (ili deo jednog procesa), a u drugom intervalu neki drugi proces (ili deo istog ili drugog procesa)
 - ❖ kombinovano, i vremenski i prostorno

Od sada će se, osim ako se posebno ne naglasi drugačije, zbog jednostavnosti pretpostaviti da se koristi stranična organizacija memorije

Tehnika	Smanjenje memorijskog zauzeća procesa	Prostorno deljenje	Vremensko deljenje
Dinamičko učitavanje	✓		✓
Preklopi	✓	✓	
Logičko deljenje memorije	✓	✓	
Deljene biblioteke	✓	✓	
Zamena procesa			✓
Učitavanje stranica na zahtev	✓	✓	
Zamena stranica (virtuelna memorija)	✓	✓	✓

Mart 2020.

Copyright 2020 by Dragan Milićev

112

Dinamičko učitavanje

Ilustracija - primer pristupa nekom potprogramu u modulu koji se dinamički učitava:

```
#define handle_error(msg) \
do { perror(msg) ; exit(EXIT_FAILURE) ; } while (0)

#define MODULE_NAME "dyn_module.bin"
#define MODULE_SIZE 0x100 // 64K
#define PROC_OFFSET 0
typedef int (*PROC_TP) (int, int) ;

int proc_stub (int a, int b) {
    static PROC_TP pProc = nullptr ;
    if (!pProc) {
        int fd = open(MODULE_NAME, O_RDONLY) ;
        if (fd == -1) handle_error("open") ;
        void* addr = mmap(NULL, MODULE_SIZE, PROT_EXEC, MAP_PRIVATE, fd, 0) ;
        if (addr == MAP_FAILED) handle_error("mmap") ;
        pProc = (PROC_TP) ((char*)addr + PROC_OFFSET) ;
    }
    return pProc(a, b) ;
}

...
int c = proc_stub(a, b) ;
```

Potprogram *proc* se uvek poziva posredno, preko *proc_stub*

Mart 2020.

Copyright 2020 by Dragan Milićev

115

Dinamičko učitavanje

Engleska reč *stub* označava ostatak, otpadak od olovke, cigarete (opušak) ili sličnog predmeta nakon upotrebe, ili odsečak od računala, ulaznice ili sličnog koji se otepi i čuva kao potvrda

Princip implementacije:

- ❖ Delovi programskog koda ili statički alocirane i inicijalizovane strukture podataka prevode se u odvojene binarne fajlove - module; prevodilac i linker mogu pružiti ovakvu podršku, pri čemu prevodilac može zahtevati podršku programera, u smislu davanja sugestija (*hints*) za to kako da organizuje potprogramme i podatke u module koji se mogu dinamički učitavati
- ❖ Program je u obavezi da organizuje vođenje evidencije o tome da li je neki modul učitán ili ne, npr. pomoću odgovarajućih pokazivača na potprogramme / podatke ili nekih struktura podataka (npr. tabele pokazivača na potprogramme); ako je pokazivač *null*, modul u kome je potprogram / podatak još nije učitán
- ❖ Program je zadužen za to da "presretne" svaki pristup datom potprogramu ili strukturi podataka, npr. u nekom potprogramu "omotaču" za pristup; npr. funkcija za pristup podatku tipa *getter/setter*, ili tzv. *patrljak (stub)* za potprogram
- ❖ Kada zaključí da se potprogramu / podatku pristupa prvi put i da modul nije učitán, npr. tako što je dati pokazivač jednak *null*, sistemskim pozivom zahteva uslugu OS kojom alocira prostor (u svom virtuelnom adresnom prostoru) za učitavanje potrebnog modula, a potom i uslugu kojom učitava taj modul na to mesto
- ❖ Program je dužan da sam zna gde se u odnosu na početak učitánog sadržaja nalazi željeni podatak / potprogram, odnosno njegov pomeraj (*offset*). Alternativno, modul može biti zapisan u standardnom formatu *obj* fajla (ali bez nerazrešenih adresa, samo sa izveženim simbolima), a OS može pružiti uslugu očitavanja pomeraja traženih simbola iz tog fajla, kao što to inače radi linker, tako da program može potreban pomeraj, ili apsolutnu adresu u koji se simbol preslikava dobiti direktno sistemskim pozivom
- ❖ Kada je ovako izračunao adresu traženog potprograma / podatka, upisaje tu adresu u odgovarajući pokazivač i svaki dalje pristup tom potprogramu / podatku obavlja odmah preko njega, jer je modul sada učitán

Mart 2020.

Copyright 2020 by Dragan Milićev



114

Dinamičko učitavanje

- ❖ Kada modul sadrži više stvari, program mora da organizuje tabelu pokazivača na potprograme/podatke u svakom modulu i da ih sve postavi kada modul učita, na primer, neka struktura i potprogrami koji njom operišu:

```
struct Module_Table {
    int (pProc*)(int,int);
    Data* pData;
} m_table;

void load_module () {
    static bool isloaded = false;
    if (!isloaded) {
        int fd = open(MODULE_NAME,O_RDONLY);
        if (fd == -1) handle_error("open");
        void* addr = mmap(NULL,MODULE_SIZE,PROT_READ|PROT_WRITE|PROT_EXEC,MAP_PRIVATE,fd,0);
        if (addr == MAP_FAILED) handle_error("mmap");
        m_table.pProc = (Data*)((char*)addr + PROC_OFFSET);
        m_table.pData = (Data*)((char*)addr + DATA_OFFSET);
        isloaded = true;
    }
}

...
load_module();
Data* p = m_table.pData; p->... = ...; ... = p->data;
```

Pre svakog pristupa, program mora da obezbedi da je modul učitán

- ❖ Ova tehnika se koristila samo kod starih računara sa malo memorije i ne koristi se više, ali su iste osnovne ideje i motivi iskorišćeni za tehnike koje su sada u odgovornosti OS-a i hardvera, pa proces o tome ne mora da brine i programiranje postaje znatno lakše (program ne mora da vodi računa o tome da li je neki njegov deo učitán)

Mart 2020.

Copyright 2020 by Dragan Milićev

117

Dinamičko učitavanje

Ilustracija - primer pristupa nekoj strukturi podataka u modulu koji se dinamički učitava:

```
#define handle_error(msg) \
do { perror(msg); exit(EXIT_FAILURE); } while (0)
```

```
#define MODULE_NAME "dyn_module.bin"
#define MODULE_SIZE 0x100 // 64K
#define DATA_OFFSET 0
struct Data ...;
```

```
Data* getData () {
    static Data* pData = nullptr;
    if (!pData) {
        int fd = open(MODULE_NAME,O_RDONLY);
        if (fd == -1) handle_error("open");
        void* addr = mmap(NULL,MODULE_SIZE,PROT_READ|PROT_WRITE,MAP_PRIVATE,fd,0);
        if (addr == MAP_FAILED) handle_error("mmap");
        pData = (Data*)((char*)addr + DATA_OFFSET);
    }
    return pData;
}

...
Data* p = getData(); p->... = ...; ... = p->data;
```

Podatku *Data* se uvek pristupa posredno, preko pokazivača koji vraća *getData*

Mart 2020.

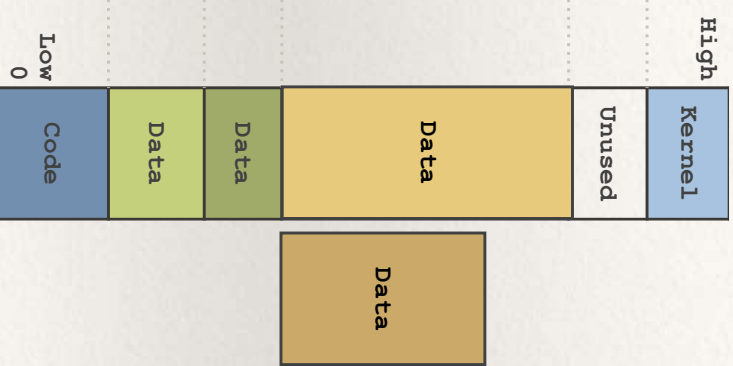
Copyright 2020 by Dragan Milićev

116

Preklopi

Mapa virtuelnog
adresnog prostora
procesa

- ❖ Svrha ove tehnike je u tome da se potrebe procesa za fizičkom memorijom još više smanje, u stepenu u kom je to moguće
- ❖ Na primer, neki složen program obrađuje neku veliku strukturu podataka, a onda ispisuje rezultate te obrade koristeći neke druge podatke, npr. stringove iz nekog vektora, pri čemu su rezultati prethodne obrade indeksi stringova u tom vektoru koje treba ispisati; ove dve faze se smenjuju naizmenično
- ❖ Kada ovaj program ne bi koristio preklape, u memoriji bi neprekidno držao obe ove velike strukture, i onu za obradu u prvoj fazi, i vektor stringova potreban u drugoj fazi
- ❖ Korišćenjem preklapa, program može da ih drži na istom mestu i koristi daleko manje memorije; zapravo, ukoliko je virtuelni adresni prostor relativno mali, manji od ukupno potrebnog prostora za ovaj program i obe strukture, bez korišćenja preklapa bi izvršavanje programa bilo i nemoguće



Mart 2020.

Copyright 2020 by Dragan Milićev

119

Preklopi

- ❖ Neki programi imaju sledeću karakteristiku: neki potprogrami ili grupe potprograma, ili neke strukture podataka se nikada ne koriste istovremeno sa nekim drugim potprogramima/podacima, već se koriste naizmenično
- ❖ Tehnika *preklapa (overlays)* podrazumeva da se moduli u kojima su grupisani potprogrami i/ili podaci koji se koriste zajedno, a u alternaciji sa drugim takvim modulima, dinamički učitavaju u memoriju (i izbacuju iz nje) *na isto mesto*, preklapajući se, pošto nikada nisu potrebni istovremeno
- ❖ Tehnika se implementira principijelno na isti način kao i za dinamičko učitavanje, uz sledeće razlike:

- ❖ više modula se učitava (naizmenično) na isto mesto u virtuelnom adresnom prostoru procesa, preklapajući se
- ❖ modul može biti nepristupan u memoriji ne samo kada mu se prvi put pristupi, nego i kasnije, jer je na njegovo mesto učitán neki drugi modul (preklóp), pa program o tome mora da vodi računa; program zapravo mora da vodi računa o tome koji je od preklópljenih modula trenutno učitán na datom mestu, kao i o tome da je pre pristupa određenom potprogramu/podaku modul u kome se on nalazi sigurno učitán (isto kao i kod dinamičkog učitavanja)
- ❖ ako neki modul koji se preklapa sadrži podatke koji se menjaju, pre nego što na njegovo mesto program učitá neki drugi modul, mora da sačuva sadržaj izbačenog modula njegovim upisom u neki fajl, ako će taj modul biti ponovo kasnije potreban
- ❖ Odgovornost za ceo ovaj posao, isto kao i kod dinamičkog učitavanja, je i dalje isključivo na procesu, tj. na programu koji se izvršava, dok OS o tome ne vodi računa
- ❖ Obaveza OS-a je i dalje samo to da obezbedi uslugu (sistemski poziv) koji alocira deo (virtuelnog) adresnog prostora procesa, kao i uslugu kojom u dati prostor procesa učitava sadržaj iz nekog binarnog fajla i sadržaj iz memorije upisuje u neki fajl (ubojčajene sistemske usluge za rad sa fajlovima); OS i dalje ne zna za šta se te usluge koriste, tj. ne zna da se one upotrebljavaju baš za dinamičko učitavanje i zamenu delova procesa

Mart 2020.

Copyright 2020 by Dragan Milićev

118

Logičko deljenje memorije

- ❖ U multiprocesnim sistemima je česta situacija da se kreira više procesa nad istim programom (isti *exe* fajl): npr. korisnik otvori više instanci (prozora) za istu aplikaciju (program)
- ❖ Ako bi se za svaki takav proces odvajao poseban prostor za smeštanje svih potrebnih logičkih segmenata, u memoriji bi bilo više istovetnih kopija istog programskog koda, koje bi bespotrebno zauzimale operativnu memoriju
- ❖ Kako se kod ne menja, a isti je u svim tim procesima, zašto se on ne bi *delio*: zašto svi ti procesi ne bi koristili istu, jednu jedinu kopiju programskog koda u fizičkoj memoriji
- ❖ Ovo je svakako lako izvodljivo, pa to sistemi uvek i rade: kada se pokreće nov proces nad istim programom za koji OS pronađe da već postoji proces, može jednostavno stranice logičkih segmenata sa kodom preusmeriti u iste okvire u kojima se taj kod već nalazi, prostim kopiranjem deskriptora iz PMT
- ❖ Ako je PMT u više nivoa, a neka tabela sadrži samo stranice logičkih segmenata sa kodom, onda ovi procesi mogu koristiti istu kopiju te tabele, pa OS štedi i na zauzeću sostvene memorije za PMT
- ❖ Uslov je, naravno, da se programski kod ne menja tokom izvršavanja, što se podrazumeva za današnje programe

Mart 2020.

Copyright 2020 by Dragan Milićev



121

Preklopi

- ❖ Obe tehnike, i dinamičko učitavanje i preklopi, datiraju iz doba kada su kapaciteti operativne memorije bili jako mali za današnje pojmove (reda nekoliko desetina kilobajta), npr. kod mikroročunara i malih kućnih ili personalnih monoprocesnih računara
- ❖ Njihova primena nije bila potrebna za male programe, dok za velike i zahtevne programe organizacija programa i podataka u module, kao i rukovanje modulima postaje sve složenije - organizacija modula, a posebno preklopa, i raspoređivanje podataka i potprograma u njih tako da se koriste zajedno oni koji su u istom preklupu, a nikada zajedno oni koji su u modulima koji se međusobno preklapaju, može postati izuzetno komplikovano
- ❖ Izvesnu pomoć može da pruži prevodilac koji podržava ove tehnike, uz kod za održavanje (učitavanje i zamenu) modula na način koji je prikazan, a koji se obezbedi kao biblioteka; međutim, prevodilac po pravilu zahteva uputstva programera za to kako da program organizuje u module / preklope
- ❖ Suštinski problem je u tome što je odgovornost za rukovanje vremenskim deljenjem memorije na samom programu, odnosno programeru
- ❖ Zato se ove tehnike danas jako retko koriste (iako mogu biti vrlo korisne u nekim posebnim situacijama)
- ❖ Međutim, isti suštinski motivi i ideje za dinamičko učitavanje delova adresnog prostora procesa samo ako i kada je to potrebno (dinamičko učitavanje), kao i za vremensko deljenje memorije (preklopi) su i dalje ostali isti, samo su dobili drugačije pojave oblike: odgovornost za to je prešla u domen OS-a, uz neophodnu podršku hardvera, dok je ceo taj mehanizam potpuno transparentan (nevidljiv) za proces koji jednostavno vidi svoj linearan i kompletan virtuelni adresni prostor

Mart 2020.

Copyright 2020 by Dragan Milićev

120

Logičko deljenje memorije

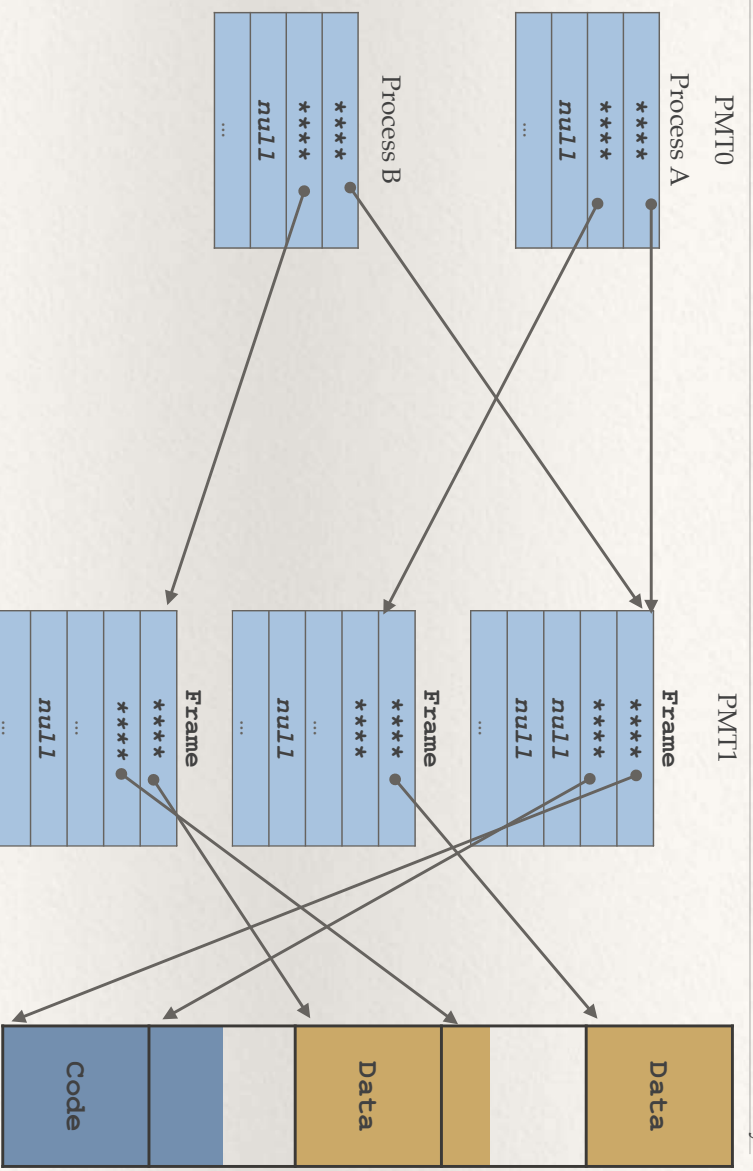
- ❖ Da li je slična stvar moguća i za podatke? Na primer, šta ako dva procesa izvršavaju isti program, sa inicijalnim istim vrednostima statički alociranih podataka
- ❖ Ako ti procesi ne menjaju te podatke, nema nikakve potrebe imati njihove odvojene kopije u memoriji, već se one mogu deliti na isti način kao i programski kod (usmeravanjem deskriptora u PMT oba procesa na iste okvire)
- ❖ Ali šta ako neki proces odluči da promeni takav podatak?
- ❖ Osnovna semantika procesa obezbeđuje da svaki proces ima svoj, izolovan virtuelni adresni prostor, sa sadržajem koji je samo njegov - drugim rečima, ako jedan proces promeni neki svoj podatak (odnosno lokacije u virtuelnom adresnom prostoru), drugi proces koji izvršava isti program ne vidi promenu svog primerka tog istog podatka (odnosno lokacije u virtuelnom adresnom prostoru)
- ❖ Ovo se realizuje tako što se iste virtuelne lokacije u različitim procesima preslikavaju u različite fizičke lokacije, koje onda imaju različit i nezavisan sadržaj
- ❖ Međutim, postoji tehnika koja može da obezbedi semantiku nezavisnih kopija neke informacije za različite "korisnike" te informacije, pri čemu oni dele istu fizičku kopiju te informacije, sve dok nijedan od njih ne želi da promeni tu informaciju - tada se pravi stvarna fizička kopija, kopije razdvajaju i nadalje svaki "korisnik" vidi i menja isključivo svoju kopiju
- ❖ Ova tehnika se naziva *kopiranje pri upisu* (*copy on write*) i veoma je opšta i korisna programska tehnika koja se često koristi u najrazličitijim kontekstima i na različitim nivoima apstrakcije

Mart 2020.

Copyright 2020 by Dragan Milićev

123

Logičko deljenje memorije

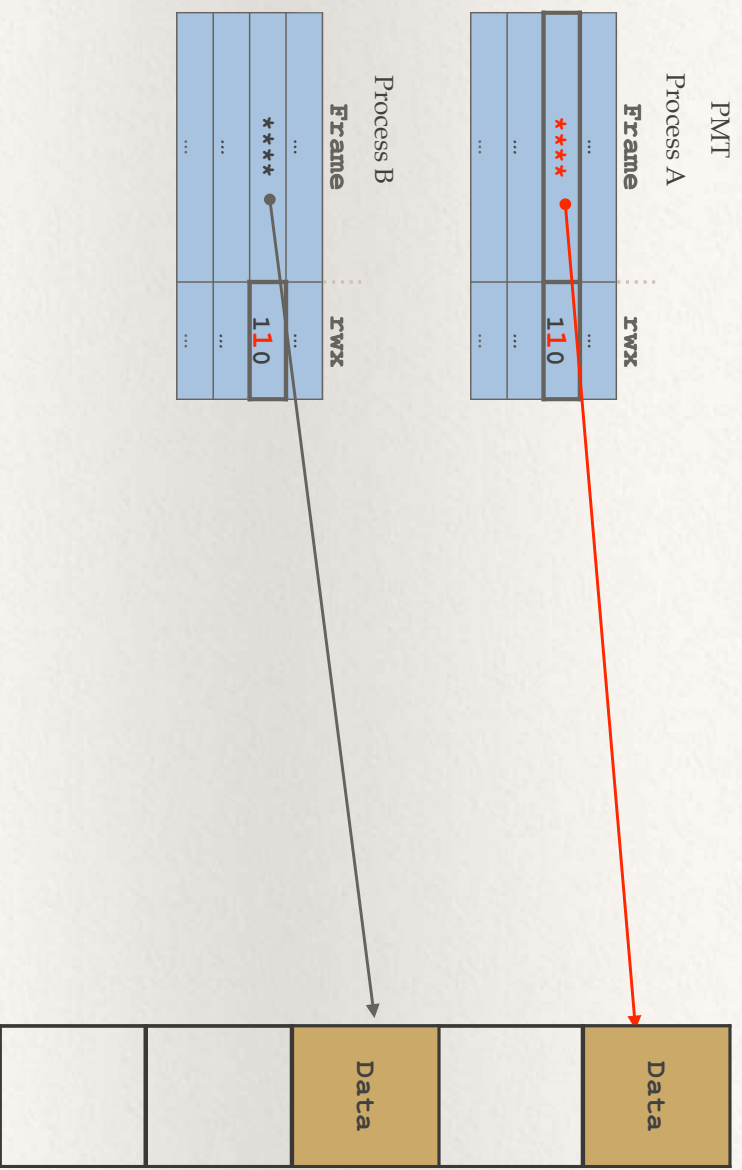


Mart 2020.

Copyright 2020 by Dragan Milićev

122

Logičko deljenje memorije



Mart 2020.

Copyright 2020 by Dragan Milićev

125

Logičko deljenje memorije

- ❖ Da bi se ova tehnika primenila na deljenje stranica sa podacima koji su dozvoljeni za upis, potrebno je presresti prvu operaciju upisa u deljenu stranicu i tom prilikom razdvojiti kopije: ovo presrećanje jedino može da uradi hardver (MMU), jer jedino on "zna" za svaku operaciju sa memorijom
- ❖ Implementacija ove tehnike za procese koji inicijalno imaju isti sadržaj nekog logičkog segmenta (statički podaci), ali treba da imaju njihovi nezavisne kopije, izgleda ovako:
 - ❖ u deskriptorima ovog logičkog segmenta oba procesa piše da je taj segment dozvoljen i za čitanje i upis
 - ❖ inicijalno se svaki ulaz u PMT za stranice tog segmenta oba procesa usmeravaju na isti okvir, tj. na istu inicijalnu kopiju sadržaja
 - ❖ ali se u deskriptorima tih stranica u oba procesa hardveru zabrani upis u te stranice
 - ❖ sve dok oba procesa samo čitaju ove stranice, oni dele istu kopiju određene stranice
 - ❖ kada bilo koji od ovih procesa izvrši prvu operaciju upisa u bilo koju lokaciju neke od ovih stranica, hardver generiše izuzetak, jer je upis zabranjen
 - ❖ OS obrađuje ovaj izuzetak, imajući informaciju o adresi koja ga je izazvala (može biti parametar izuzetka ili se može dekodovati iz instrukcije koja je izazvala izuzetak, a to je instrukcija na adresi koja je u sačuvanoj vrednosti registra PC)
 - ❖ OS pronalazi deskriptor segmenta kom pripada ta adresa, i u tom deskriptoru pronalazi informaciju da je logički segment zapravo logički dozvoljen za upis, ali je u pitanju "okidač" za kopiranje pri upisu (ako to nije tako, proces je zaista napravio pristup u pristupu memoriji i OS će ga ugasiti)
 - ❖ OS tada alokira nov okvir za stranicu, u nju prepisuje sadržaj adresirane stranice (pravi kopiju), usmerava deskriptor za tu stranicu u PMT nekog od procesa na tu novu kopiju (razdvaja kopije) i u deskriptorima te stranice oba procesa dozvoljava upis
 - ❖ OS tada vraća proces koji je izazvao izuzetak u izvršavanje, ponovo od iste instrukcije koja je izazvala izuzetak
- ❖ Ista ova tehnika kopiranja na upis može se primenljivati i na same PMT nekog nivoa

Mart 2020.

Copyright 2020 by Dragan Milićev

124



Deljene biblioteke

- ❖ U multiprocesnim sistemima je vrlo čest slučaj da više procesa koristi iste biblioteke:
 - ❖ svakako one koje implementiraju funkcije kojima se vrše sistemski pozivi
 - ❖ neke druge vrlo često korišćene biblioteke, npr. za rad sa stringovima, apstraktnim strukturama podataka, za složene matematičke operacije, grafiku, komunikacione protokole (one posebne, višeg nivoa apstrakcije, koje ne implementira kernel), itd.
- ❖ Kada bi se takve biblioteke statički linkovale u *exe* fajlove svih programa koji ih koriste, onda bi ponovo u memoriji postojale višestruke kopije istog sadržaja unutar prostora procesa koji izvršavaju različite programe
- ❖ Ideja je zato da se taj sadržaj deli između u procesa, na isti način kako se deli i ceo program za procese nad istim programom. Međutim, razlike su u sledećem:
 - ❖ ne deli se kod celog programa, nego samo deo - kod jedne biblioteke
 - ❖ taj kod treba povezati sa ostatkom programa koji koristi usluge te biblioteke (poziva potprograme iz nje) *dinamički*, za vreme izvršavanja procesa, a ne pre pokretanja procesa kao kod klasičnog, statičkog povezivanja
- ❖ Zato se ovakve biblioteke nazivaju (*deljene*) *biblioteke sa dinamičkim vezivanjem* (*/shared/ dynamic linking libraries*, DLL)

Mart 2020.

Copyright 2020 by Dragan Milićev

127

Logičko deljenje memorije

- ❖ Postoji ponekad potreba, a operativni sistemi i to podržavaju, da bilo koji procesi, čak i oni koji ne izvršavaju isti program, dele određeni segment memorije (svako svoj logički segment), kako bi preko te deljene memorije razmenjivali informacije: ono što jedan upiše u neku lokaciju tog deljenog segmenta, drugi će moći da pročita
- ❖ Isti sistemski poziv *mmap* može da služi i za ovu namenu, s tim da se u odgovarajućem parametru setuje bit maskom *MAP_SHARE* umesto *MAP_PRIVATE*
- ❖ Implementacija ovoga je jednostavna i poput već navedenih: deskriptori odgovarajućih stranica PMT ovih procesa ukazuju na isti fizički okvir, uz potrebne dozvole prava pristupa

Mart 2020.

Copyright 2020 by Dragan Milićev

126

Deljene biblioteke

- ❖ Detalji: kako se kod DLL-a mapira u različite logičke segmente u različitim procesima, taj kod ne može da koristi apsolutno adresiranje, već relativno; po pravilu, to se svodi na adresiranje skokova u potprogramima koji po pravilu sve svoje podatke adresiraju preko parametara (kojima se opet pristupa relativno, preko steka)
- ❖ Da bi proces identifikovao DLL, sistemski poziv sadrži i nekakav identifikator (naziv) tog DLL-a, kako bi ga OS pronašao
- ❖ DLL-ovi su softver koji trpi izmene, kao i svaki drugi, pa se postavlja pitanje kompatibilnosti programa koji koristi neki DLL sa novim ili stariim verzijama tog DLL-a
- ❖ Zato je često sastavni deo identifikacije DLL-a, pored naziva, i oznaka verzije, a u memoriji mogu postojati učitanе različite verzije istog DLL-a; svaki program koristiće onu verziju koju je tražio (ili noviju) i sa kojom je kompatibilan
- ❖ Dinamičke biblioteke se intenzivno koriste sa ciljem smanjenja zauzeća memorije, ali i zbog efikasnijeg rada sistema, jer se njihov sadržaj samo jednom učitava u memoriju (i iz nje po pravilu ne izbacuje dokle god je neki proces koristi)
- ❖ Pored toga, DLL-ovi se koriste i kada ne postoje ovakvi motivi, odnosno kada neki DLL koristi samo jedan program, jer se tako potencijalno velik program razbija na module:
 - ❖ ti moduli se onda mogu učítavati dinamički, ali sada kao DLL-ovi, samo ako i kada su potrebni, slično kao kod dinamičkog učítavanja, pa program smanjuje svoje zahteve za (sada virtuelnom) memorijom
 - ❖ softverski paket se lakše održava, jer se ispravke (tzv. "zакrpe", *patch*) mogu vršiti promenama samo u modulu u kome su učinjene, a taj modul se isporučuje korisnicima kao DLL, što je po pravilu manji paket nego što bi bio ceo glomazni *exe* koji bi bio izgrađen kao monolit

Mart 2020.

Copyright 2020 by Dragan Milićev

129

Deljene biblioteke

- ❖ Principijelno, DLL fajl ima isti format i sadržaj zapisa kao i obična (statička) biblioteka, samo što se ona ne povezuje statički; na primer, standardni format ELF služi i za zapisivanje DLL-ova
- ❖ Način implementacije može da bude sledeći:
 - ❖ program koji koristi neki DLL može pozivati potprogramme na uobičajen način, bez ikakve razlike, ali se statički poveže sa bibliotekom koja sadrži samo "patrljke" (*stub*) potprograma iz DLL-a
 - ❖ ovi patrljci sadrže kod sličan onom koji je pokazan za dinamičko učítavanje, ali taj kod ne alokira prostor i ne poziva sistemski poziv za učítavanje sadržaja iz bilo kog fajla, već poziva poseban sistemski poziv koji zahteva mapiranje određenog DLL-a
 - ❖ OS vodi evidenciju o DLL-ovima koji su već učítani u memoriju, na zahtev drugih procesa, kao i tabele simbola koje ti DLL-ovi izvoze izgrađene prilikom učítavanja tih DLL-ova
 - ❖ kada proces zatraži učítavanje i povezivanje sa DLL-om sistemskim pozivom, OS potraži taj DLL među već učitanim; ako DLL nije učítan, OS alokira prostor u fizičkoj memoriji za njegovo učítavanje, učítava njegov sadržaj iz fajla, i izgrađuje tabelu njegovih simbola, kao što to inače radi statički linker
 - ❖ ako i kada je DLL učítan, OS formira logički segment u virtuelnom adresnom prostoru pozivajućeg procesa, taj segment preslikava u prostor učítanog DLL-a, i vraća simbol (ili pokazivač na tabelu simbola) pozivajućem procesu
 - ❖ patrljak koji je pozvao ovaj sistemski poziv sada koristi vraćenu adresu simbola ili tabele simbola kako bi pozvao ciljni potprogram čija je implementacija u DLL-u

Mart 2020.

Copyright 2020 by Dragan Milićev

128

Zamena procesa

- ❖ To se, ipak, ne radi, jer bi bilo izrazilo neefikasno, pošto bi promena konteksta uključivala izuzetno zahtevne operacije snimanja velikog dela memorije na disk i isto takvo učitavanje sa diska, što bi trajalo jako dugo (reda sekundi) - potpuno neprihvatljivo za promenu konteksta koja se viši često
- ❖ Zato savremeni operativni sistemi rade zamenu celih procesa (*swapping*) samo u izuzetnim situacijama, kada je opterećenje sistema veliko, tj. kada je broj kreiranih procesa veliki i njihovi zahtevi za memorijom takvi da se ne mogu svi opslužiti, a da sistem funkcioniše efikasno:
 - ❖ ako ne mogu svi procesi da stanu u memoriju (uključujući i same strukture koje kernel organizuje za memorijski kontekst procesa u svom delu memorije)
 - ❖ kao tehnika za sprečavanje pojave zvane *batrganje* (*thrashing* - mlaćenje) - detalji kasnije
- ❖ Kako bi rasteretio "pritisak" na memoriju i uspeo da u nju smesti sve procese tako da se oni mogu efikasno izvršavati, OS odabere jedan ili više procesa koje će izbaciti iz memorije (*swap out*); te procese OS onda suspenduje na nešto duži rok, dok se situacija ne popravi, tj. neki procesi ugase, kada izbačene procese ponovo vraća u memoriju (*swap in*)
- ❖ OS može koristiti i druge kriterijume za izbacivanje procesa, npr. ako proces čeka na akciju korisnika u prozoru aplikacije (korisničkog interfejsa), a taj prozor nema fokus, proces te aplikacije je kandidat za izbacivanje; kada se vrati fokus na taj prozor, OS učitava taj proces. Ova tehnika je ipak efikasna samo u izuzetnim slučajevima preopterećenja, odnosno velikog broja aktiviranih procesa (primer je stara verzija 3.1 sistema Windows koji je ovako radio), jer je odziv sistema na ovakvu akciju korisnika prilično spor

Mart 2020.

Copyright 2020 by Dragan Milićev

131

Zamena procesa

- ❖ Da li se ceo proces može izbaciti iz memorije na neki rok, njegovo izvršavanje na taj rok suspendovati (zaustaviti, odložiti), a prostor u fizičkoj memoriji koji je on zauzimao osloboditi i dodeliti onda nekom drugom procesu?
 - ❖ Pre izbacivanja iz memorije, ceo memorijski kontekst procesa (deskriptore logičkih segmenata), kao i sadržaj tih segmenata memorije (barem onaj promenljivi, za podatke i stek) OS mora da snimi negde na disk
 - ❖ Kada odluči da nastavi izvršavanje ovog suspendovanog procesa, OS učitava sačuvane deskriptore izbačenog procesa sa diska, iznova kreira ceo memorijski kontekst procesa (PMT), alokira stranice i učitava sačuvan sadržaj u alocirane stranice i nastavlja izvršavanje procesa
- ❖ Ova tehnika naziva se *zamena procesa* (*swapping*): jedan proces se *izbacuje* (*swap out*) iz memorije, a drugi učitava, *ubacuje* (*swap in*) u memoriju
- ❖ Kada OS vrši promenu konteksta, odnosno kada tekući proces gubi procesor, on se neko vreme neće izvršavati dok ponovo ne dobije procesor; ako je već tako, onda mu ni sadržaj njegovog virtuelnog adresnog prostora nije potreban, jer mu proces svakako neće pristupati, pa zašto onda ne bi bio potpuno izbačen iz memorije?
- ❖ Na ovaj način bi OS mogao da vrši *vremensko deljenje* operativne memorije za cele procese, jer bi istu fizičku memoriju u jednom intervalu koristio jedan proces, a u drugom neki drugi proces. Štaviše, iako bi OS bio multiprocetni, u memoriji bi mogao da bude samo po jedan proces za svaki postojeći procesor i on bi mogao da koristi svu raspoloživu memoriju!?

Mart 2020.

Copyright 2020 by Dragan Milićev

130

Učitavanje stranica na zahtev

Implementacija ove tehnike je relativno jednostavna:

- ❖ inicijalno, pri kreiranju procesa, OS kreira memorijski kontekst procesa - deskriptore logičkih segmenata i PMT, ali u ulaze u PMT upiše vrednost *null*, osim za one stranice koje inicijalno alokira i učitava pri pokretanju procesa; potom pušta proces u izvršavanje
- ❖ kada neka instrukcija procesa prvi put pristupi nekoj stranici koja nije učitana, MMU će generisati straničnu grešku (*page fault*) - virtuelna adresa ne može da se preslika u fizičku, nezavisno od razloga, i tu se odgovornost hardvera završava; ovaj izuzetak obrađuje OS, kao i obično
- ❖ OS proverava da li virtuelna adresa koja je generisala ovaj izuzetak pripada regularno alociranom segmentu; ako ne pripada, proces je napravio prestup u pristupu memoriji i OS može da ga ugasi
- ❖ u suprotnom, radi se o tome da je proces ispravno adresirao virtuelnu adresu, ali stranica kojoj ta adresa pripada nije alocirana u memoriji
- ❖ OS rešava to tako što pronalazi slobodan okvir u operativnoj memoriji, zauzima taj okvir i u njega sa diska učitava sadržaj tražene stranice ili uradi drugu potrebnu radnju, a u deskriptor te stranice u PMT upisuje broj tog okvira
- ❖ sve dok ovaj postupak traje, proces je suspendovan; kada se ceo ovaj postupak završi, OS može da nastavi izvršavanje ovog procesa, ali sada od *iste instrukcije* koja je generisala straničnu grešku, ponavljanjem te instrukcije (registar PC mora da ima odgovarajuću vrednost adrese te instrukcije)
- ❖ kada nastavi izvršavanje ponavljanjem prekinute instrukcije, proces će moći da izvrši adresiranje, jer je tražena stranica sada u memoriji

Mart 2020.

Copyright 2020 by Dragan Milićev



133

Učitavanje stranica na zahtev

- ❖ Po sličnoj logici i sa istim motivima kao i za dinamičko učitavanje, zašto se ne bi stranice procesa učitavale dinamički, tek onda kada su potrebne, *na zahtev*? Ako neki deo virtuelnog adresnog prostora proces uopšte ne koristi tokom svog izvršavanja, te stranice nikada neće ni biti učitane, pa neće ni zauzimati memoriju
- ❖ Ovo je moguće i sasvim jednostavno za realizaciju uz već opisanu i postojeću podršku hardvera - tehnika *učitavanja stranica na zahtev* (*demand paging*)
- ❖ Prilikom pokretanja procesa, OS može da učitaa samo nekoliko stranica koje pokrivaju deo sa kodom u kom je početna instrukcija, možda sa stekom, ali u graničnom slučaju ne mora da učitaa ni jednu jedinu stranicu!
- ❖ Za razliku od dinamičkog učitavanja, ovaj mehanizam je sada potpuno nevidljiv (transparentan) za proces i semantiku njegovog izvršavanja, jer ceo posao obavlja OS uz podršku hardvera; naravno, kao i kod dinamičkog učitavanja, vreme inicijalnog učitavanja i pokretanja procesa je kraće, ali se vreme izvršavanja produžava zbog naknadnog učitavanja stranica

Mart 2020.

Copyright 2020 by Dragan Milićev

132

Učitavanje stranica na zahtev

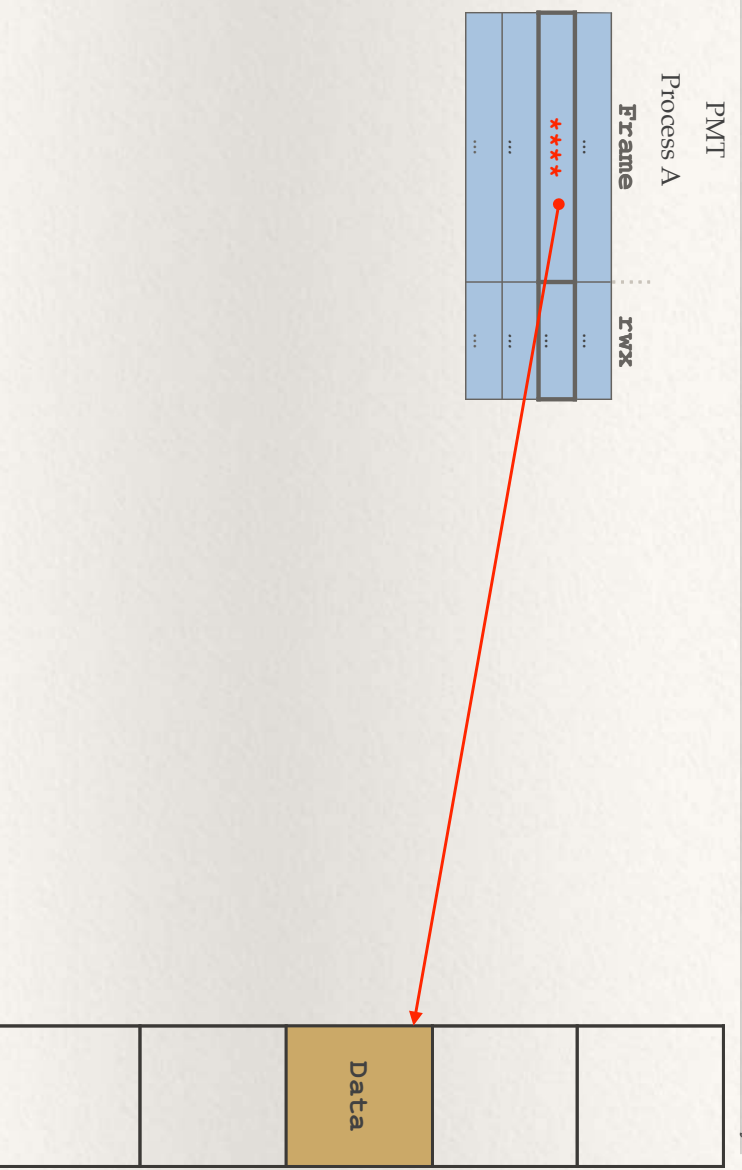
- ❖ Isto kao i kod tehnike kopiranja na zahtev (*copy on write*), proces koji je generisao izuzetak mora da nastavi izvršavanje ponavljanjem iste instrukcije koja je izazvala taj izuzetak, nakon što OS obradi izuzetak i reši nastalu situaciju na opisani način
- ❖ Da bi semantika izvršavanja instrukcija bila predvidiva i ista kao da ovakvih događaja i nema (transparentna za izvršavanje programa), hardver procesora mora da obezbedi da ponavljanje instrukcije nakon prekida ima isti efekat na stanje procesora i memorije kao i samo jedno njeno izvršavanje
- ❖ Hardver procesora to može da uradi na neki od sledećih načina:
 - ❖ ako je svaka operacija tokom izvršavanja instrukcije nakon koje se može dogoditi izuzetak *idempotentna*, tj. takva da je efekat ponovljene operacije isti kao i efekat jednom izvršene operacije, ništa posebno ne mora da se radi; na primer, operacija upisa iste vrednosti u istu lokaciju je idempotentna operacija, dok operacija inkrementiranja vrednosti nije
 - ❖ RISC procesori sa *load/store* arhitekturom po pravilu nemaju ovaj problem, jer su jedine instrukcije koje mogu generisati izuzetak, izvan samog dohvaćanja instrukcije, instrukcije *load* i *store* (sve ostale rade sa registrima procesora, pa ne pristupaju memoriji tokom izvršavanja); zato su one inherentno idempotentne
 - ❖ CISC procesori imaju veći problem jer mogu imati složene instrukcije koje izvršavaju složene, neidempotentne operacije sa operandima u memoriji; na primer, instrukcija koja inkrementira operand koji ima više reči u memoriji: pristup nižoj reči npr. ne izaziva straničnu grešku, jer je stranica u kojoj je ona u memoriji, ali pristup višoj reči izaziva, jer se ta reč nalazi u drugoj, neučitanoj stranici (operand je “prelomljen” granicom između stranica). Ovakvi procesori moraju da ostave stanje registara procesora i memorije onakvo kakvo je bilo *pre* izvršavanja prekinute instrukcije; na primer, tako što sačuvaju (u neke svoje privremene registre) stare vrednosti svih lokacija (registara ili memorije) koje je instrukcija izmenila, pa pri obradi izuzetka povrate te vrednosti (*rollback*)
- ❖ U svakom slučaju, ovo je odgovornost hardvera i sve je to transparentno za OS; zadatak operativnog sistema je samo da nastavi izvršavanje procesa od iste prekinute instrukcije, stavljanjem korektne vrednosti u PC; i sam procesor može raditi tako da vrednost PC vrati na adresu prekinute instrukcije, a ne one naredne, što olakšava posao operativnom sistemu

Mart 2020.

Copyright 2020 by Dragan Milićev

135

Učitavanje stranica na zahtev



Physical
Memory

Mart 2020.

Copyright 2020 by Dragan Milićev

134

Zamena stranica

- ❖ Kada je proces izazvao straničnu grešku i OS je obrađuje sa namerom da traženu stranicu učita, šta se dešava ako u memoriji nema nijednog slobodnog okvira u koji bi se ta stranica mogla učitati?
- ❖ Jedan pristup je taj da se proces izbaci (*swap out*) iz memorije i suspenduje do momenta kada se neki procesi ugase i oslobode memoriju, čime se proces suspenduje na duži rok
- ❖ Da li se može primeniti ista ideja kao kod preklopa, da se tražena stranica učita na mesto neke druge, kojoj se tako okvir koji zauzima "preotme", a ona "izbac" iz memorije? Naravno!
- ❖ Na taj način stranice *vremenski* dele fizičku memoriju, odnosno okvire
- ❖ Značajne posledice ovog pristupa su:
 - ❖ proces više ne mora imati učtane sve stranice koje će koristiti tokom svog izvršavanja, jer su neke izbačene, iako ih je ovaj proces koristio tokom svog izvršavanja
 - ❖ virtuelni adresni prostor, pa i onaj njegov deo koji je proces alocirao, može biti daleko veći od fizičke memorije; ovo je danas svakako slučaj za 64-bitne arhitekture (64-bitne virtuelne adrese)
 - ❖ proces na taj način poseduje i "vidi" i slobodno koristi linearnu *virtuelnu memoriju* (doduše, logički segmentiranu na segmente koje je alocirao) koja zapravo fizički uopšte ne mora da postoji u računaru

Mart 2020.

Copyright 2020 by Dragan Milićev

137

Učitavanje stranica na zahtev

- ❖ U zavisnosti od vrste logičkog segmenta kom adresirana stranica pripada, OS treba ili ne treba da vrši učitavanje stranice; informaciju o tipu svakog logičkog segmenta OS čuva u deskriptoru tog segmenta
- ❖ Kada neka instrukcija generiše straničnu grešku, osim alokacije okvira za smeštanje adresirane stranice, OS treba da uradi sledeće za svaki tip logičkog segmenta:

Tip segmenta	Akcija pri prvom pristupu (pored alokacije okvira)
Segment sa instrukcijama (tzv. <i>text segment</i>)	Učitavanje iz <i>exe</i> fajla
Segment sa statički alociranim podacima inicijalizovanim konstantnim izrazima	Učitavanje iz <i>exe</i> fajla
Segment sa statički alociranim podacima inicijalizovanim nulama (tzv. <i>bss segment</i>)	Inicijalizacija nulama
Segment sa statički alociranim podacima koje će inicijalizovati sam program	-
Dinamički alociran segment sa podacima koje će inicijalizovati sam program	-
Stek segment	-

Mart 2020.

Copyright 2020 by Dragan Milićev

136

Zamena stranica

- ❖ Procesori često imaju sledeću hardversku podršku: u deskriptoru stranice u PMT koriste jedan bit, tzv. *bit zaprljanosti* ili *bit modifikacije* (*dirty bit*, *modify bit*) koji MMU postavlja na 1 svaki put kada se izvrši operacija upisa u neku (bilo koju) reč te stranice - indikator da je sadržaj te stranice promenjen u odnosu na trenutak kada je ovaj bit resetovan
- ❖ Kada učitava stranicu u memoriju, OS resetuje ovaj bit u deskriptoru te stranice
- ❖ Kada izabere neku stranicu za izbacivanje, OS može da konsultuje ovaj bit i ako je on 0, sadržaj stranice nije menjan od kada je učitana, pa ne mora da se snima pri izbacivanju; tako obrada stranične greške može da bude znatno kraća

Mart 2020.

Copyright 2020 by Dragan Miličev

139

Zamena stranica

Obrada stranične greške (*page fault*) sada izgleda ovako:

- ❖ OS proverava da li virtuelna adresa koja je generisala ovaj izuzetak pripada alociranom segmentu; ako ne pripada, proces je napravio prestup u pristupu memoriji i OS može da ga ugasi
- ❖ u suprotnom, radi se o tome da je proces ispravno adresirao virtuelnu adresu, ali stranica kojoj ta adresa pripada nije učitana u memoriju
- ❖ OS pokušava da nađe slobodan okvir u operativnoj memoriji koji će dodeliti toj stranici
- ❖ ako slobodnog okvira nema, bira neku stranicu istog ili drugog procesa, tzv. *žrtvu* (*victim*), koja će biti izbačena iz memorije, odnosno kojoj će biti preotet okvir
- ❖ u deskriptoru odabrane stranice-žrtve označava da ona više nije u memoriji, tj. upisuje vrednost *null* kao broj okvira
- ❖ da bi kasnije mogao da povрати sadržaj te stranice kada ona bude adresirana, OS pokreće operaciju snimanja sadržaja te stranice na disk ako je potrebno
- ❖ kada se operacija snimanja stranice na disk završi, pokreće operaciju učitavanja tražene stranice sa diska
- ❖ kada se operacija učitavanja završi, u deskriptor te učitane stranice u PMT upisuje broj tog okvira
- ❖ sve dok ovaj postupak traje, proces je suspendovan; kada se ceo ovaj postupak završi, OS može da nastavi izvršavanje ovog procesa, ponovo od *iste instrukcije* koja je generisala straničnu grešku, ponavljanjem te instrukcije

Mart 2020.

Copyright 2020 by Dragan Miličev

138

Zamena stranica

- ❖ Kao i kod dinamičkog učitavanja stranica, neke stranice uopšte nije potrebno snimati pri izbacivanju, jer se njihov sadržaj ne menja - npr. stranice sa instrukcijama
- ❖ Operacije koje OS obavlja u različitim situacijama za stranice koje pripadaju različitim tipovima logičkih segmenata sada izgledaju ovako:

Tip segmenta	Pri prvom pristupu	Pri narednim pristupima	Pri izbacivanju
Segment sa instrukcijama	Učitavanje iz <i>exe</i> fajla	Učitavanje iz <i>exe</i> fajla	-
Segment sa statički alociranim podacima inicijalizovanim konstantnim izrazima	Učitavanje iz <i>exe</i> fajla	Učitavanje iz prostora za zamenu	Snimanje u prostor za zamenu
Segment sa statički alociranim podacima inicijalizovanim nulama (tzv. <i>bss segment</i>)	Inicijalizacija nulama	Učitavanje iz prostora za zamenu	Snimanje u prostor za zamenu
Segment sa statički alociranim podacima koje će inicijalizovati sam program	-	Učitavanje iz prostora za zamenu	Snimanje u prostor za zamenu
Dinamički alociran segment sa podacima koje će inicijalizovati sam program	-	Učitavanje iz prostora za zamenu	Snimanje u prostor za zamenu
Stek segment	-	Učitavanje iz prostora za zamenu	Snimanje u prostor za zamenu

Mart 2020.

Copyright 2020 by Dragan Milićev

141

Zamena stranica

- ❖ Prostor na disku u koji OS snima izbačene stranice i sa kog ih ponovo učitava naziva se *prostor za zamenu* (*swap space*). Taj prostor OS može organizovati:
 - ❖ unutar nekog fajla koji se konfiguracijom operativnog sistema odredi za tu namenu; OS snima i učitava stranice kao binarni sadržaj tog fajla; to znači da operacije snimanja i učitavanja stranica koriste usluge dela kernela koji implementira fajl sistem, što može biti manje efikasno, jer operacije sa fajlovima uključuju određene režije koje mogu biti značajne
 - ❖ na posebnoj particiji na disku koja služi samo za tu namenu i na kojoj nije instaliran fajl sistem, na tzv. *presnoj particiji* (*raw partition*), kao prostor za zamenu, pa nema potrebe za složenijim strukturama i operacijama koje zahteva fajl sistem, što može biti znatno efikasnije
- ❖ Neki operativni sistemi omogućavaju oba pristupa, u zavisnosti od konfiguracije koju postavi administrator
- ❖ Konkretno mesto u konfigurisanom prostoru za zamenu stranica određenog logičkog segmenta OS može da izračunava na osnovu informacija o mestu i načinu smeštanja tog segmenta koje se nalaze u deskriptoru tog segmenta

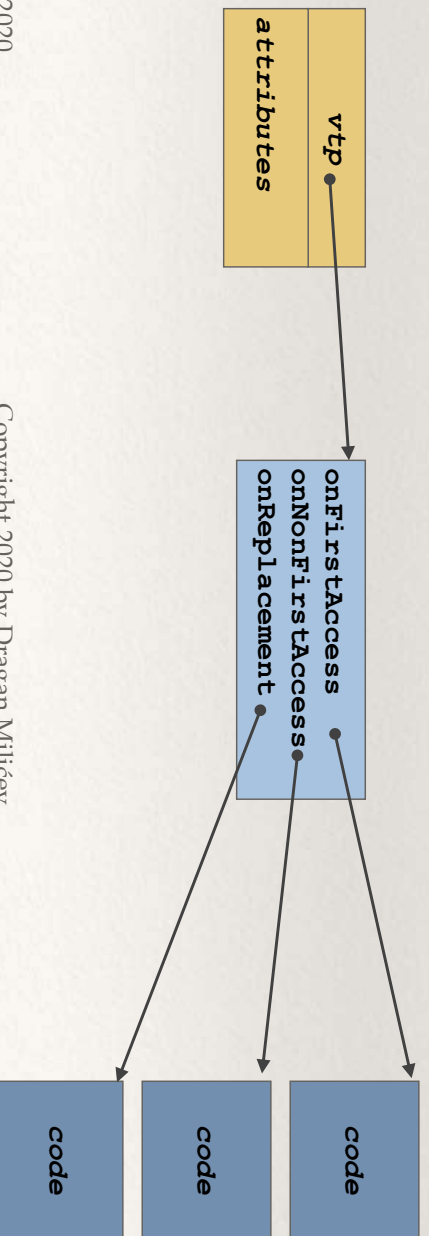
Mart 2020.

Copyright 2020 by Dragan Milićev

140

Zamena stranica

- ❖ Kako su operativni sistemi najčešće implementirani na jeziku C, ovaj polimorfizam implementira se u programu baš onako kako se implementiraju pozivi virtuelnih funkcija na jeziku C++:
- ❖ objekat (ovde deskriptor logičkog segmenta) predstavlja se strukturom u kojoj su podaci (atributi), ali i jedan pokazivač na tabelu pokazivača na virtuelne funkcije (*virtual table pointer*, VTP)
- ❖ za svaku konkretnu klasu postoji tabela sa pokazivačima na kod funkcija koje odgovaraju toj klasi (nasleđene ili redefinisane)
- ❖ poziv polimorfne funkcije obavlja se dinamičkim vezivanjem (*dynamic binding*): iz strukture se dohvata VTP, iz tabele na koju on ukazuje dohvata se sadržaj odgovarajućeg ulaza (za željenu funkciju) i taj sadržaj tretira kao pokazivač, odnosno kao adresu potprograma koji treba pozvati



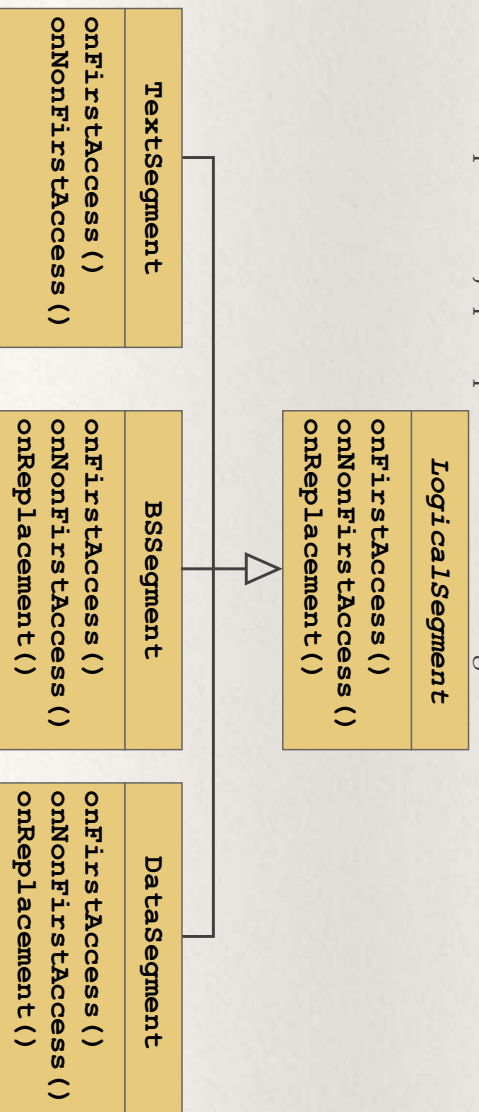
Mart 2020.

Copyright 2020 by Dragan Milićev

143

Zamena stranica

- ❖ Ovakva varijabilnost ponašanja može da se posmatra na sledeći način, u objektnom duhu:
- ❖ postoji apstraktna klasa *logičkog segmenta* sa polimorfnim operacijama koje kernel poziva u navedenim slučajevima
- ❖ konkretni tipovi segmentata predstavljeni su klasama izvedenim iz ove apstraktne klase
- ❖ te izvedene klase redefinišu ove operacije
- ❖ deskriptori segmentata su objekti ovih izvedenih klasa, čije operacije kernel poziva polimorfno tokom sprovođenja postupka obrade stranične greške



Mart 2020.

Copyright 2020 by Dragan Milićev

142

Zamena stranica

- ❖ Šta se dešava ako je u sistemu aktivno mnogo procesa, svaki zauzima deo operativne memorije, ali te memorije nema dovoljno, pa procesi imaju vrlo malo svojih stranica u memoriji?
- ❖ Procesi će vrlo često generisati stranične greške:
 - ❖ proces generiše straničnu grešku
 - ❖ OS preotme okvir od istog ili nekog drugog procesa
 - ❖ ubrzo taj drugi proces zatraži izbačenu stranicu i ponovo generiše straničnu grešku, itd.
- ❖ Sistem može da uđe u režim u kom se stranične greške dešavaju izuzetno često, procesi često i dugo bivaju suspendovani, pa procesor nema šta da radi, iskorišćenje mu je izuzetno nisko, dok ulazno-izlazni podsistem za operacije sa diskom postaje preopterećen, ne može da posluži sve zahteve, pa zahtevi dugo čekaju - disk podsistem postaje usko grlo celog računara
- ❖ Ovakva loša situacija naziva se *batrganje* (*thrashing* - mlaćenje) i dobar OS mora da se štiti od nje
- ❖ Da bi se zaštitio, OS mora ili da primenjuje tehnike sprečavanja ove pojave, ili da je detektuje da je nastala i da se iz nje izbavi - detalji u predmetu OS2
- ❖ Da bi sprečio ovu pojavu, OS mora da:
 - ❖ obezbedi dovoljan broj okvira svakom procesu, kako bi on mogao da napreduje
 - ❖ ukoliko to ne može, da izbacuje procese (*swamp out*) kako bi oslobodio delove memorije i kako bi procesi koji ostaju u memoriji mogli da dobiju dovoljno okvira i napreduju
- ❖ Praktična procena: da bi sistem funkcionisao efikasno, potrebno je da učestanost stranične greške bude reda 10^{-6}

Copyright 2020 by Dragan Milićev

145

Zamena stranica

- ❖ Dok OS ne završi obradu stranične greške i ne završi učitavanje tražene stranice, proces koji je generisao tu straničnu grešku je suspendovan i ne može da se izvršava, odnosno ne može da napreduje; kako bi postigao maksimalnu efikasnost, tj. kako bi procesi što manje bili zaustavljani u napredovanju i čekali, OS treba da teži da se:
 - ❖ stranične greške dešavaju što ređe
 - ❖ obrada stranične greške završi što brže
- ❖ Jedan od važnih elemenata koji utiču na efikasnost jeste pitanje odabira stranice-žrtve za izbacivanje:
 - ❖ logički, može se izabrati bilo koja stranica, jer to ne utiče na semantiku izvršavanja procesa: kada proces čija je stranica izbačena adresira tu stranicu, generisaće straničnu grešku, pa će ta stranica biti učitana
 - ❖ međutim, odabir stranice i te kako može da utiče na *performanse sistema*, odnosno na to koliko će se često dešavati stranične greške, odnosno koliko će često procesi biti zaustavljani zbog toga: ako vrlo brzo nakon izbacivanja izbačena stranica bude adresirana, vrlo brzo će biti generisana nova stranična greška i neki proces će ponovo biti zaustavljen
- ❖ Zato je *algoritam zamene stranica* (*page replacement algorithm*), tj. algoritam koji bira stranicu-žrtvu za izbacivanje, posebno značajan za ponašanje sistema
- ❖ Sistemi najčešće primenjuju neku varijantu aproksimacije *LRU algoritma* (*least recently used*): izbacuje se ona stranica koja je (približno) najdavnije korišćena (kojoj se najdavnije pristupalo) - detalji u predmetu OS2
- ❖ Za potrebe ovog algoritma, hardver procesora treba da podrži još jedan bit u deskriptoru stranice, tzv. *bit referenciranja* (*reference bit*) koji MMU postavlja prilikom svake operacije sa stranicom (i čitanja i upisa): OS razmatra ovaj bit tokom algoritma zamene kako bi video koje su stranice korišćene od trenutka kada je OS taj bit obrisao - detalji u predmetu OS2

Mart 2020.

Copyright 2020 by Dragan Milićev

144

dr Dragan Milićev

redovni profesor

Elektrotehnički fakultet u Beogradu

dmilicev@etf.rs, www.rcub.bg.ac.rs/~dmilicev

Operativni sistemi

Osnovni kurs

Slajdovi za predavanja

Deo III - Upravljanje procesima

Za izbor slajda drži miša
uz levu ivicu ekrana

Mart 2020.

Copyright 2020 by Dragan Milićev

1

Zamena stranica

- ❖ Prema tome, mehanizam virtualne memorije sa zamenom stranica predstavlja vrlo moćan mehanizam savremenih računara sa sledećim pozitivnim i negativnim efektima:
 - ❖ proces vidi potencijalno ogroman virtualni adresni prostor, kao linearan prostor, koji može biti znatno veći od fizičke memorije, jer ne mora ceo da stane u nju - proces poseduje *virtualnu memoriju*
 - ❖ iako, teorijski, proces može da alokira ceo svoj ogroman virtualni adresni prostor, to po pravilu OS neće moći da izvede, jer ne može da alokira potrebne strukture u raspoloživoj memoriji (problem velikih PMT za popunjene virtualne adresne prostore)
 - ❖ broj procesa koji se mogu kreirati u sistemu je logički neograničen
 - ❖ povećanje broja procesa povećava opterećenje memorije, povećava učestanost straničnih grešaka, pa procesi sporije napreduju - računar "sporije radi", ovaj efekat je manje uočljiv ako računar ima veću količinu operativne memorije, i obratno (ovo je objašnjenje za na prvi pogled sasvim nelogičnu vezu između brzine rada računara i količine operativne memorije koju on ima, a za koju mnogi laici znaju i koriste kao kriterijum odabira kada kupuju uređaje)
 - ❖ ovaj mehanizam može izazvati vrlo neprijatno ponašanje računara (*thrashing*) ako se ne koristi efikasno i ako sistem nije napravljen tako da se od te pojave štiti
 - ❖ zato su važni algoritmi zamene stranica, kao i način na koji OS stranicama procesa dodeljuje okvire - detalji u predmetu OS2

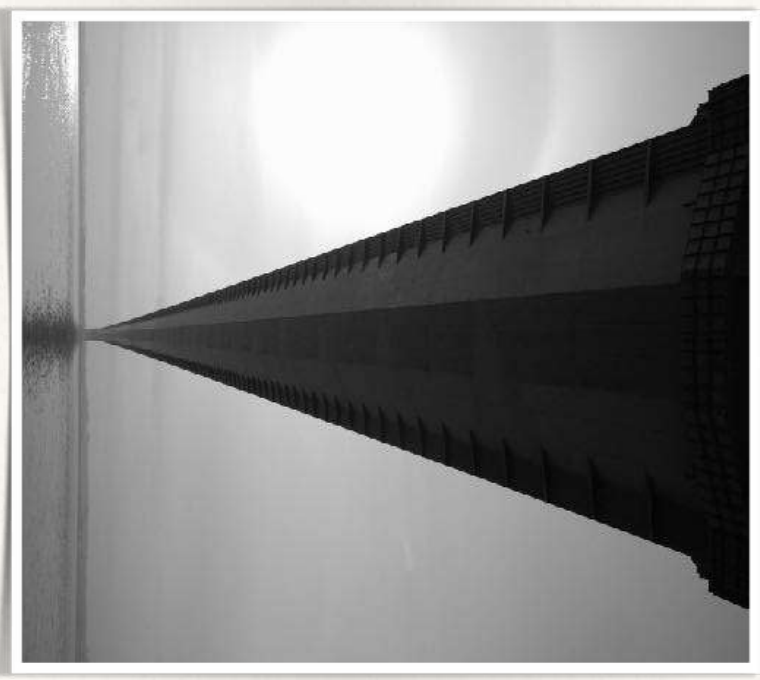
Mart 2020.

Copyright 2020 by Dragan Milićev

146

Glava 7: Procesi i niti

- ❖ Pojam procesa
- ❖ Kreiranje procesa
- ❖ Gašenje procesa
- ❖ Pojam niti
- ❖ Operacije sa nitima



Mart 2020.

Copyright 2020 by Dragan Milićev

3

Deo III: Upravljanje procesima

Mart 2020.

Copyright 2020 by Dragan Milićev

2

Pojam procesa

- ❖ Dva osnovna elementa semantike izvršavanja programa, koja potiče još iz doba prvih i najjednostavnijih, monoprocesnih računara, a zapravo predstavljaju pojavni oblik Tjuringove mašine i Fon Nojmanove arhitekture, jesu:
 - ❖ *tok kontrole* (*control flow* ili *flow of control*): redosled sekvencijalnog izvršavanja instrukcija, jedne po jedne, u kom iza prethodne sledi sledeća koja je odmah iza nje u memoriji, osim ako instrukcija ne uradi drugačije (instrukcija uslovnog ili bezuslovnog skoka), uključujući i skokove u potprograme, sa tragom povratnih adresa na steku, i sa povratkom iz potprograma indirektnim skokom preko adrese skinute sa steka
 - ❖ *stanje* (*state*) registara procesora i lokacija memorije, adresiranih adresama koje generiše instrukcija raspoloživim načinima adresiranja, pri čemu stanje koje za sobom ostavi prethodno izvršena instrukcija u toku kontrole, sledeća instrukcija u tom toku zaštiće u registrima ili lokacijama memorije
- ❖ Kod prvih, jednostavnih, monoprocesnih računara, program u izvršavanju imao je na raspolaganju ceo procesor (procesor je izvršavao instrukcije samo tog procesa) i celu operativnu memoriju (sve adrese koje proces generiše bile su fizičke adrese operativne memorije), tako da su ova dva elementa semantike izvršavanja procesa bila ispoljena i u realnosti, fizičkim izvršavanjem na procesoru i u operativnoj memoriji

Mart 2020.

Copyright 2020 by Dragan Milićev

5

Pojam procesa

- ❖ *Proces* (*process*) je jedno izvršavanje nekog programa sa jednim (*virtuelnim*) *adresnim prostorom* (*address space*):
 - ❖ na multiprocesnom računaru, može se pokrenuti (kreirati) više procesa, nad istim ili različitim programima
 - ❖ ti procesi izvršavaju se *uporedo* (konkurentno, *concurrently*), na jednom procesoru (multiprogramiranjem) ili na više procesora (multiprocesiranjem)
 - ❖ svaki proces ima svoj (*virtuelni*) adresni prostor sa skupom podataka sa kojima radi i koji su samo njegovi, iako se uporedo izvršavaju i drugi procesi, možda i nad istim programom
- ❖ Procesi pokrenutih na sistemu mogu biti:
 - ❖ *interaktivni*: imaju interakciju sa korisnikom kroz korisnički interfejs (CLI ili GUI), recimo tako što imaju aktivan prozor u GUI-u (tzv. "aplikacije")
 - ❖ *pozadinski* (*background, batch*): nemaju interakciju sa korisnikom, već izvršavaju neke radnje uporedo sa drugim, "izvršavaju se u pozadini", većinu ovakvih procesa pokreće sam sistem pri podizanju, ali mogu da ih pokrenu i korisnici ili njihovi procesi



Prikaz svih aktivnih procesa u sistemu u nekom trenutku

Mart 2020.

Copyright 2020 by Dragan Milićev

4

Kreiranje procesa

- ❖ Proces nad nekim programom može kreirati korisnik, posredstvom korisničkog interfejsa:
 - ❖ komandom u komandnoj liniji (CLI); najčešće je samo ime komande (prvi podniz znakova u komandnoj liniji) naziv programa nad kojim se kreira proces:
`my_program arg1 arg2 arg3`
 - ❖ akcijom (GUI), npr. pritiskom miša na ikonicu programa
- ❖ Prilikom kreiranja procesa, mogu mu se dostaviti i opcioni *argumenti komandne linije* (*command line arguments*):
 - ❖ CLI: interpreter zapravo *sve* podstringove (nizove susednih znakova, razdvojene prazninama) prosleđuje procesu kao argumente
 - ❖ GUI: OS omogućava da se programu, prilikom instalacije ili kasnijim podešavanjem, podeše argumenti (nije često u uobičajenoj upotrebi)

Mart 2020.

Copyright 2020 by Dragan Milićev

7

Pojam procesa

- ❖ Koncept procesa zapravo predstavlja prvi oblik i nivo *virtualizacije* izvršavanja programa, jer je semantika izvršavanja procesa na multiprocesnom sistemu ista kao što je izvorno i bila, s tim da:
 - ❖ proces više nije sam na procesoru, već isti procesor izvršava i instrukcije drugih procesa, ali proces to “ne vidi”: tok kontrole jednog procesa je isti, i iza jedne instrukcije procesa logički sledi ona koja bi bila naredna da je proces sam, iako, realno, između te dve instrukcije može da protekne duži period u kom procesor izvršava instrukcije mnogih drugih procesa
 - ❖ se stanje koje održava proces očuvava:
 - ❖ ono što jedna instrukcija procesa ostavi u registrima procesora, sledeća instrukcija istog procesa će u njima zateći, iako je između te dve instrukcije, realno, fizički, procesor izvršavao mnoge druge instrukcije koje su mogle da promene te iste registre; *promena konteksta* (*context switch*) je postupak koji obavlja OS i koji ovo obezbeđuje na sledeći način: pre nego što procesor pređe na izvršavanje instrukcija drugog procesa, OS *sačuvava* stanje registara procesa čije izvršavanje se prekida, a potom *restaurira* stanje registara procesa na čije izvršavanje prelazi
 - ❖ vrednost koju jedna instrukcija procesa upiše u neku memorijsku lokaciju, sledeća će u toj lokaciji zateći, bez obzira na to što između njih mnogi drugi procesi upisuju u iste (virtuelne) adrese; koncept virtuelne memorije ovo obezbeđuje tako što se virtuelne adrese različitih procesa preslikavaju u različite fizičke adrese memorije, sa različitim sadržajem, a prilikom promene konteksta OS menja i memorijski kontekst (*PMTT / SMTP*)

Mart 2020.

Copyright 2020 by Dragan Milićev

6

Kreiranje processa

- ❖ Jedan proces može kreirati nov proces sistemskim pozivom. U žargonu, *proces roditelj (parent process)* kreira *proces dete (child process)*
- ❖ Na primer, kada se korisnički interfejs, CLI ili GUI, izvršava kao proces (školjka), ovaj proces ima potrebu da kreira proces dete, kada prepozna odgovarajuću komandu odnosno akciju
- ❖ I bilo koji drugi proces može kreirati proizvoljno mnogo novih processa, svoje dece, sistemskim pozivima
- ❖ Sistemski poziv za kreiranje processa može biti takav da se proces kreira nad zadatim programom, uz opcioni prenos argumentata komandne linije; na primer, na Win32 API to principijelno izgleda ovako (mnogi detalji su izostavljeni, jer su složeni, a ovde nebitni):
`CreateProcessA(strProgramName, strCommandArgs, ...);`
- ❖ U takvim sistemskim pozivima, naravno, nov proces dete izvršava zadati program, sa svojim, novim adresnim prostorom, inicijalizovanim prema sadržaju *exe* fajla

Mart 2020.

Copyright 2020 by Dragan Milićev

9

Kreiranje processa

- ❖ Ove argumente program na jeziku C/C++ dobija kao argumente funkcije *main*:
`int main (int argc, char* argv[]);`
argc: ukupan broj argumentata, uvek veći od 0
argv: niz stringova veličine *argc*, sa stringovima koji predstavljaju argumente (podstringove) iz komandne linije, pri čemu je *argv[0]* uvek sam naziv programa
- ❖ Ili, na jeziku C# i sistemu Windows, pozivom:
`public static string[] Environment.GetCommandLineArgs ();`
- ❖ Drugi skup parametara programa su tzv. *variable okruženja (environment variables)*:
 - ❖ skup identifikatora (prostitih nizova znakova, tipično velikim slovima) sa pridruženim vrednostima tipa nizova znakova (parovi *name = value*)
 - ❖ imaju različito značenje; neke varijable postavlja sam OS, npr. koreni direktorijum, tip standardnog ulaza itd, ali se pri kreiranju programa ili kasnije tokom njegovog izvršavanja mogu postaviti vrednosti proizvoljnih novih ili postojećih varijabli
 - ❖ služe za podešavanje parametara izvršavanja programa
- ❖ proces može očitati vrednosti ovih varijabli sistemskim pozivom, npr.:
`char* getenv (const char* name);` Vraća vrednost (niz znakova) varijable sa zadatim imenom

Mart 2020.

Copyright 2020 by Dragan Milićev

8

Kreiranje processa

- ❖ Ovaj sistemski poziv vraća negativnu vrednost u slučaju greške (kod greške); u tom slučaju, proces dete nije ni kreiran
 - ❖ Ukoliko ovaj sistemski poziv uspe, postoje dva toka kontrole koja nastavljaju svoja izvršavanja povratkom iz funkcije *fork*: roditeljski, koji vrši povratak iz “normalnog” poziva te funkcije i dete, koji se vraća “niotkuda”, kako bi se u kodu koji se izvršava iza poziva *fork* mogla napraviti razlika između ta dva konteksta, odnosno toka kontrole, *fork* vraća:
 - ❖ 0 u kontekstu (toku kontrole) procesa deteta
 - ❖ vrednost veću od 0 u kontekstu roditelja; ova vrednost predstavlja *identifikator procesa deteta*, tzv. *process id* (skraćeno *pid*)
- ```
pid_t pid = fork();
if (pid < 0) {
 ..
} else
if (pid == 0) {
 ..
} else {
 ...
}
```
- Greška, proces dete nije ni kreiran
- U kontekstu procesa deteta, izvršavaće se ovaj deo koda
- U kontekstu procesa roditelja, izvršavaće se ovaj deo koda
- ❖ Celobrojna vrednost identifikatora procesa je jedinstvena identifikacija koja se kasnije upotrebljava kao argument sistemskih poziva za operacije nad procesom koji se tim identifikatorom identifikuje

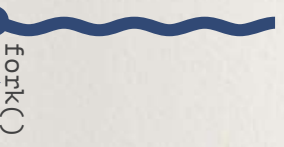
Mart 2020.

Copyright 2020 by Dragan Milićev

11

# Kreiranje processa

- ❖ Međutim, na sistemima nalik sistemu Unix, postoji drugačiji tradicionalan način da proces roditelj kreira nov proces dete - sistemski poziv *fork* (*fork* - “račva”):  
`pid_t fork ();` — *pid\_t (pid type)* je zapravo sinonim za neki celobrojni tip, npr. *int*, koji može da smesti *pid*
- ❖ Ovaj sistemski poziv, ukoliko uspe, kreira nov proces dete, kao identičnu kopiju, “klon” procesa roditelja, sa sledećim značenjem:
  - ❖ proces dete dobija identičan tok kontrole, koji izvršava isti programski kod kao i roditelj, od iste tekuće pozicije u izvršavanju, tj. od mesta poziva i povratka istog sistemskog poziva *fork*, sa istim tragom poziva i povratnih adresa na steku, tj. sa istom “predistorijom”, kao da se i proces dete izvršavao pre toga na identičan način kao i roditelj, iako faktički nije ni postojao, sa istim tekućim stanjem registara procesora
  - ❖ proces dete kao da dobija inicijalno identičnu kopiju sadržaja virtuelnog adresnog prostora, sa svim sadržajem (programski kod, statički i dinamički podaci, stek); njihovi adresni prostori su inicijalno identični, ali predstavljaju *nezavisne logičke kopije*: svaka izmena koju u svom adresnom prostoru jedan od ovih procesa napravi, drugi proces nadalje (nakon povratka svog toka kontrole iz *fork*) neće videti; fizički, naravno, ovo se implementira tehnikom kopiranja na upis, pa sistem ne pravi stvarne, fizičke kopije delova adresnog prostora (segmenta, stranica) u memoriji dok to ne bude potrebno



Mart 2020.

Copyright 2020 by Dragan Milićev

10

# Kreiranje processa

- ❖ Pored ovoga - pitanja inicijalizacije adresnog prostora, sistemi i njihove usluge kreiranja processa mogu da variraju u još nekim aspektima postojanja različitih sistemskih poziva i njihove semantike
- ❖ Jedan je pitanje nastavka izvršavanja processa roditelja i processa deteta pri sistemskom pozivu kreiranja processa:
  - ❖ u jednoj varijanti, kakve su opisane, process roditelj nastavlja odmah svoje izvršavanje, uporedo i nezavisno od processa deteta
  - ❖ process roditelj se implicitno *suspenduje* (privremeno se odlaže njegovo izvršavanje) dok se kreirani process dete ne ugasi (završi); na primer, CLI, kao process, pri izvršavanju komande koja pokreće process, suspenduje svoje izvršavanje dok se taj process ne završi, i tek onda nastavlja dalje učitavanjem nove komandne linije
- ❖ Operativni sistemi najčešće podrazumevaju prvu navedenu varijantu semantike sistemskog poziva kreiranja processa, s tim što obezbeđuju posebne sistemske pozive kojim process roditelj može eksplicitno da se suspenduje do gašenja nekog od svoje dece ili svih njih, sistemskim pozivom
- ❖ Na primer, u sistemima nalik sistemu Unix, nekoliko varijanti sistemskog poziva *wait* suspenduje pozivajućí process dok se neki process dete ne završi, uz dodatne opcije kontrole:

```
wait(NULL);
waitpid(pid, NULL);
```

Suspenduje process dok se neki od processa dece ne završi

Suspenduje process dok se ne završi process dete sa datim *pid*

Mart 2020.

Copyright 2020 by Dragan Milićev

13

# Kreiranje processa

- ❖ Na primer:

```
const int N = ...;
```

```
int main () {
 pid_t pid[N];
 for (int i=0; i<N; i++)
 pid[i] = fork();
}
```

Koliko će na kraju ukupno processa biti kreirano?

Kada se nad ovim kodom pokrene jedno izvršavanje (jedan process), on će u svom adresnom prostoru imati niz *pid* i svoju promenljivu *i* (na svom steku). Ako pretpostavimo da će svi sistemski pozivi uspeti, ovaj process će u prvoj iteraciji (*i=0*) kreirati nov process dete. I process roditelj i process dete nastavljaju dalje, svaki svojim tokom, i svaki ima svoju, nezavisnu vrednost za *i* i svoj niz *pid*, s tim što će u kontekstu roditelja *pid[0]* imati vrednost identifikatora processa deteta, a u kontekstu deteta imati vrednost 0. Dalje oba ova processa nastavljaju svoja izvršavanja, uporedo i potpuno nezavisno, prelaze svako u svoju sledeću iteraciju, za *i=1*, i dalje svaki od njih kreira još po jedan process itd.

Mart 2020.

Copyright 2020 by Dragan Milićev

12



# Kreiranje processa

- ❖ Pored sistemskog poziva *fork*, sistemi nalik sistemu Unix poseduju sistemski poziv *exec*, tj. čitavu familiju sličnih funkcija sa istim efektom, uz varijacije parametara, koje rade sledeće: ne kreira se nikakav nov proces, kao entitet u operativnom sistemu, već se ceo postojeći memorijski kontekst tekućeg processa (onog koji je pozvao *exec*) potpuno odbacuje, i iznova inicijalizuje iz zapisa u *exe* fajlu koji je zadat parametrom ovog poziva i započne tok kontrole ispočetka, izvršavanjem programa u tom *exe* fajlu:

```
int execl (const char* path, const char* arg, ...);
int execlp (const char* file, const char* arg, ...);
int execlx (const char* path, const char* arg, ..., char* const envp[]);
int execv (const char* path, const char* argv[]);
int execvp (const char* file, const char* argv[]);
int execvpe (const char* file, const char* argv[], char *const envp[]);
```

- ❖ Ovim funkcijama mogu se zadati i vrednosti argumenata komandne linije, a u nekim varijantama i vrednosti promenljivih okruženja

Mart 2020.

Copyright 2020 by Dragan Milićev

15

# Kreiranje processa

- ❖ Još jedan aspekt u kom semantička operacija kreiranja processa u operativnim sistemima može da varira jeste pitanje koje od sistemskih resursa proces dete nasleđuje od roditelja, a koje ne, i kojim resursima sme da pristupa, a kojim ne; pod sistemskim resursima podrazumevaju se fizički i logički objekti kojima rukuje OS a koje je proces roditelj dobio na korišćenje, kao što su otvoreni fajlovi, uređaji, uključujući i standardni ulaz i izlaz, i slično
- ❖ U jednoj varijanti, proces dete može da koristi ili da zahteva da koristi (eksplicitnim sistemskim pozivima) samo resurse koje je proces roditelj koristio. U drugoj varijanti, može da zatraži da koristi bilo koji resurs
- ❖ Osim toga, pitanje je i to koje resurse proces dete *nasleđuje* od roditelja. Na primer, u jednoj čestoj varijanti, proces dete od processa roditelja implicitno nasleđuje standardni ulazni i standardni izlazni uređaj, kao i *korisnika* u čije ime i sa čijim pravima pristupa se izvršava u višekorisničkim sistemima
- ❖ Primer upotrebe ovakvog pristupa jeste ponovo CLI kao proces (školjka): kada kreira proces dete, taj proces dete podrazumevano nasleđuje isti standardni ulazni uređaj (npr. tastaturu), sa koje učitava znakove, i isti standardni izlazni uređaj (kozolu, prozor) na koji ispisuje svoj izlaz, kao i sva prava pristupa; na ovaj način se i stvara utisak da CLI izvršava komandu koja može da ispisuje neke rezultate, ili čak dalje da interaguje sa korisnikom preko konzole, ako se radi o komandi pokretanja programa
- ❖ CLI, kao proces roditelj, može i da preusmeri proces dete na drugi ulazni ili izlazni uređaj (koncept redirekcije), kao i da promeni prava pristupa, naglašavajući takvu varijantu semantike sistemskog poziva kreiranja processa deteta
- ❖ U ovim aspektima sistemi mnogo variraju i semantika u različitim sistemima i njihovim uslugama ima puno detalja za koje treba uvek konsultovati specijalizovanu literaturu i referentnu dokumentaciju za konkretan sistem

Mart 2020.

Copyright 2020 by Dragan Milićev

14



# Gašenje procesa

- ❖ Proces može ugasiti sebe eksplicitnim zahtevom, odnosno tražiti završetak (*termination*) svog izvršavanja sistemskim pozivom *exit*:  
`void exit (int status);`
- ❖ Ovaj sistemski poziv prima jedan parametar koji ima značenje "informacije o statusu" koji OS prenosi roditeljskom procesu procesa koji se gasi, prema konvenciji, vrednost 0 označava "regularan završetak", sve drugo nekaav drugi
- ❖ Interpretacija ove vrednosti je u svakom slučaju na roditeljskom procesu, a on taj status može dobiti preko sistemskog poziva *wait*, tako što kao parametar *status* ovog poziva prenese pokazivač na celobrojnu promenljivu u koju će ovaj sistemski poziv upisati status procesa deteta po njegovom gašenju:  
`pid_t wait (int *status);`  
`pid_t waitpid (pid_t pid, int *status, int options);`

U slučaju uspeha, vraćaju *pid* procesa deteta koji se završio

❖ Na primer:

```
int status;
wait(&status);
if (status==0)
... // Child completed regularly
else
... // Child completed with an error
```

Mart 2020.

Copyright 2020 by Dragan Milićev

17

## Kreiranje procesa

- ❖ Na primer, sledeći program koristi sistemske pozive *fork* i *exec* da bi pokrenuo proces nad programom zadatim drugim argumentom svoje komandne linije:

```
int main (int argc, char* argv[]) {
 if (argc<2) {
 printf("Syntax: %s programName\n", argv[0]);
 return -1;
 }
 pid_t pid = fork();
 if (pid<0) {
 printf("Cannot create process for %s\n", argv[1]);
 return -1;
 } else
 if (pid==0) {
 int s = execlp(argv[1], NULL);
 if (s<0) {
 printf("Cannot execute file %s\n", argv[1]);
 return -1;
 }
 }
 wait(NULL);
 return 0;
}
```

Kontekst procesa deteta

Ako poziv *exec* uspe, iz njega nema povratka u isti tok kontrole, pa se kod iza uspešnog završetka poziva *exec* nikada ne izvršava!

Kontekst procesa roditelja

Mart 2020.

Copyright 2020 by Dragan Milićev

16

# Gašenje procesa

- ❖ Operativni sistemi po pravilu omogućavaju i to da jedan proces ugasi neki drugi proces sistemskim pozivom (u žargonu, da ga “ubije”); ovaj sistemski poziv se tradicionalno naziva *kill*
- ❖ Zbog toga što bi neograničeno pravo da neki proces gasi druge procese bio štetan, sistemi obično uvode ograničenja u pogledu prava na ovu operaciju; ta ograničenja se razlikuju od sistema do sistema, pa čak i od verzije do verzije, ali su neka česta ograničenja sledećih tipova:
  - ❖ proces sme da ugasi samo procese koji se izvršavaju u ime istog korisnika kao i on
  - ❖ proces sme da ugasi samo procese koje je sam kreirao
- ❖ Ovaj sistemski poziv *kill* u sistemima nalik sistemu Unix, pa i u mnogim drugim ima zapravo nešto složeniji efekat: on ne predstavlja eksplicitnu operaciju gašenja procesa (iako je inicijalno za to namenjen i tako i dobio naziv), već zahtev za slanjem *signala* navedenom određišom procesu
- ❖ *Signal* je prosta informacija o identifikaciji nekakve proste poruke, tipično jednostavna celobrojna vrednost
- ❖ OS prenosi tu informaciju određišom procesu, a reakcija na taj signal je poziv potprograma (rutine) koja se izvršava u kontekstu tog određišog procesa i u njegovom adresnom prostoru
- ❖ Za sve signale, OS za svaki proces definiše podrazumevanu reakciju, odnosno implicitne, podrazumevane rutine, ali za neke signale proces može tu reakciju da preusmeri na svoje potprograme
- ❖ Podrazumevano ponašanje na signale je uglavnom to da se proces gasi, odnosno da vrši sistemski poziv *exit*

Mart 2020.

Copyright 2020 by Dragan Milićev

19

# Gašenje procesa

- ❖ U suštinu, eksplicitni poziv *exit* je jedini način da se proces regularno završi tako što sam traži svoje gašenje: procesor uvek izvršava nekakve instrukcije, dohvata sledeću sa adrese na koju ukazuje *PC* i izvršava je, i tako u nedogled; nekako se operativnom sistemu mora reći da je “proces gotov sa izvršavanjem”, a kontrolu mora preuzeti OS da bi uradio sve potrebne radnje po gašenju procesa, da taj proces kao entitet ne bi više postojao u sistemu (i da ne bi dobio procesor), a da OS procesor preda nekom drugom procesu; to se može uraditi, i mora tako reći operativnom sistemu jedino sistemskim pozivom
- ❖ Međutim, poznato je to da se C / C++ program implicitno završava (preciznije, proces koji izvršava taj program gasi) kada tok kontrole napusti funkciju *main*, bilo eksplicitno (npr. naredbom *return*) ili implicitno, kada tok kontrole “propadne” na kraj bloka ove funkcije ili “iskoči” zbog bačenog a neuhvaćenog izuzetka. Slično je i na drugim jezicima
- ❖ Kako se to onda implementira, kada tu onda nigde nema sistemskog poziva *exit*?
- ❖ Naravno, tako što kod funkcije *main* nije jedini kod “glavnog programa” koji proces izvršava: pre njega se izvršava inicijalizacioni kod, kao što je rečeno, koji onda poziva funkciju *main* kao svaku drugu funkciju, a nakon izlaska iz funkcije *main* (na bilo koji način) takođe se nastavlja taj kod koji dealocira sve statičke objekte (poziv destruktora na jeziku C++) i obavlja druge implicitne radnje koje se podrazumevaju semantikom jezika; upravo taj završni kod na kraju sadrži i sistemski poziv *exit*
- ❖ Drugi način da proces ugasi sam sebe, odnosno da se proces ugasi kao posledica njegovog delovanja, jeste situacija u kojoj taj proces generiše neki hardverski izuzetak koji OS ne može da prevaziđe (neregularna instrukcija, neregularan način adresiranja, pristup u pristupu memoriji itd.)
- ❖ Ovo je, kao i sistemski poziv *exit*, sinhroni prekid procesa - posledica akcije samog procesa, s tim da je *exit* eksplicitni, “voljni” prekid, dok je izuzetak “nevoljni”, implicitni način da proces ugasi sebe (tj. da izazove svoje gašenje od strane operativnog sistema)

Mart 2020.

Copyright 2020 by Dragan Milićev

18

# Gašenje procesa

- ❖ OS pruža mogućnost gašenja odabranog procesa i kroz svoj UI:
  - ❖ CLI: odgovarajućom komandom, tipično *kill*, koja zapravo samo “omotava” sistemski poziv *kill*, tj. šalje odgovarajući signal procesu sa zadatim identifikatorom:

```
kill -9 1234
kill -KILL 1234
kill -15 1234
kill -TERM 1234
```

Prvi argument, iza znaka - je vrednost signala koji se šalje procesu čiji je *pid* dat kao drugi argument
  - ❖ GUI: svaki OS obezbeđuje neki način da u nekom prikazu pokaže sve aktivne procese ili aplikacije i mogućnost da se neki od njih odabere i zahteva njegovo blagonaklono ili nasilno gašenje; npr. opcija *Task Manager* koja se dobija pritiskom kombinacije tastera *Ctrl+Shift+Esc* ili *Ctrl+Alt+Del* u sistemima Windows, ili opcija *Force Quit* u sistemu Mac OS X i slično

Mart 2020.

Copyright 2020 by Dragan Milićev

21

# Gašenje procesa

- ❖ Zbog svega toga, sistemski poziv *kill* izgleda ovako na sistemima nalik sistemu Unix:

```
int kill (pid_t pid, int signal);
```

Ovaj sistemski poziv šalje dati signal procesu sa datim identifikatorom *pid*
- ❖ Neki predefinisani signali su definisani kao sledeće simboličke konstante:
  - ❖ *SIGKILL*: tipično ima vrednost 9, nasilno gasi određeni proces; proces ne može da redefiniše reakciju na ovaj signal, OS obavezno nasilno gasi taj proces
  - ❖ *SIGTERM*: tipično ima vrednost 15 i služi da “blagonaklono” (*gracefully*) ugasi proces, odnosno da mu uputi zahtev da se ugasi sam; proces može da preusmeri reakciju na ovaj signal na svoju proceduru kojom će uraditi sve potrebne radnje pre svog gašenja (npr. upozoriti korisnika, snimiti dokument i slično); podrazumevana reakcija je opet nasilno gašenje, kao za *SIGKILL*
  - ❖ postoje mnogi drugi definisani signali, npr. *SIGSTOP*, *SIGINT* itd, od kojih se neki šalju kada se u CLI pritisnu neke karakteristične kombinacije tastera, npr. *CTL+C*, *Ctrl+Z* i slično

Mart 2020.

Copyright 2020 by Dragan Milićev

20



# Pojam niti

---

- ❖ Šta je uopšte motiv za korišćenje niti?
- ❖ Uporedo obavljati neke aktivnosti ili obradu, ili reakcije na događaje iz okruženja koji se dešavaju asinhrono - u proizvoljnim, nepredvidivim trenucima i poretku
- ❖ Ako bismo obradu za uporedne aktivnosti ili reakcije na asinhronone događaje programirali klasičnim, sekvencijalnim tehnikama programiranja, posledice bi bile sledeće:
  - ❖ programer bi morao da vodi računa o tome da svakoj aktivnosti ili događaju posvećuje dovoljno pažnje i vrši njihovu obradu u potrebnoj meri; to bi značilo da u program ugrađuje logiku koja se malo bavi jednom aktivnošću, ali ne previše, kako ne bi zapostavio neku drugu, onda nekom drugom, a onda proveriti da li se dogodio neki događaj na koji treba reagovati itd.
  - ❖ ovakav način razmišljanja i programiranja značajno otežava konstrukciju programa, a te programe čini nepreglednijim, teškim za razumevanje i održavanje, pa time i podložnim greškama
  - ❖ programer mora da vodi računa o tome da ako neka aktivnost mora da se suspenduje, odnosno da čeka na nešto, prede na izvršavanje drugih aktivnosti, a onda se vrati ovoj kada uslov nastavka njenog izvršavanja bude ispunjen
  - ❖ ukoliko postoji više procesora koji mogu paralelno, istovremeno izvršavati sekvence instrukcija, nema načina da se ovakve aktivnosti rasporede po tim procesorima i obavljaju paralelno

Mart 2020.

Copyright 2020 by Dragan Milićev

23

# Pojam niti

---

- ❖ Kao što je rečeno, pojam procesa čine dva ključna semantička elementa:
  - ❖ *tok kontrole*: sekvencijalno izvršavanje jedne po jedne instrukcije, u skladu sa njihovom semantikom, uključujući i instrukcije skoka, kao i pozive potprograma i povratak iz njih preko adresa sačuvanih na steku
  - ❖ *adresni prostor*: sadržaj memorijskih lokacija koje instrukcije procesa mogu adresirati, tako da efekat upisa jedne instrukcije u neku lokaciju “vide” naredne instrukcije tog procesa
- ❖ Međutim, ova dva elementa su u principu nezavisni: može se formirati više uporednih tokova kontrole koji *dele isti virtuelni adresni prostor*
- ❖ Upravo to predstavlja *niti* (*thread*): tok kontrole koji teče uporedo sa drugim tokovima kontrole, ali koji deli virtuelni adresni prostor sa nekim drugim tokom ili tokovima kontrole (nitima)
- ❖ Naravno, svaka nit ima svoj kontekst izvršavanja, odnosno sve ono što je potrebno za sopstveni tok kontrole:
  - ❖ svoje stanje registara procesora, uključujući na prvom mestu *PC*, koji ukazuje na tekuću poziciju u sekvenci izvršavanja instrukcija u toku kontrole, kao i *SP*, koji ukazuje na poziciju steka na kom su trag izvršavanja i aktuelni (nezavršeni) aktivacioni blokovi, ali i sve ostale registre dostupne u korisničkom režimu rada
  - ❖ svoj stek
- ❖ Sa druge strane, nekoliko niti može da deli:
  - ❖ zajednički ostatak adresnog prostora (osim steka): statičke i dinamičke podatke, kao i programski kod
  - ❖ resurse operativnog sistema, kao što su otvoreni fajlovi, standardni ulazni i standardni izlazni uređaji i drugo

Mart 2020.

Copyright 2020 by Dragan Milićev

22

# Pojam niti

- ❖ Jedan jednostavan primer: potrebno je sabrati dve velike matrice, što se može uraditi sekvencijalno, ali i *uporedno* (konkurentno, *concurrently*), tako što zbir elemenata svake vrste obavlja po jedna nit; ako računar ima više procesora, OS ove niti može rasporediti na te procesore, pa će se one izvršavati paralelno i obrada će biti daleko efikasnija
- ❖ Suština je u tome da je pitanje raspoređivanja niti na procesore transparentno (nevidljivo) za program i programera, on o tome ne brine: definisanjem niti, programer specifikuje mogućnost uporednog izvršavanja, a OS to implementira
- ❖ Razlika između procesa i niti ponekad se naglašava i sledećim terminima:
  - ❖ proces, ili “teški” proces (*heavyweight process*)
  - ❖ nit, ili “laki” proces (*lightweight process*)
- ❖ Operativni sistemi vrlo često procesom zovu jedan kreiran adresni prostor, tj. jedan memorijski kontekst, a svaki tok kontrole koji koristi taj adresni prostor zovu nit; pri kreiranju procesa, zapravo se kreira jedna inicijalna nit, jedan početni tok kontrole koji dalje može kreirati neke nove
- ❖ Nit se obično kreira kao novi tok kontrole koji započinje izvršavanje pozivom nekog potprograma, opciono sa prenesenim argumentima, i dalje sledi tok koji diktriraju instrukcije tog potprograma (i svih ugnežđenih potprograma)

Mart 2020.

Copyright 2020 by Dragan Milićev

25

# Pojam niti

- ❖ Koncept uporednih tokova kontrole, tj. procesa ili niti, upravo predstavlja rešenje za ovakav problem:
  - ❖ svakoj aktivnosti ili obradi događaja posvećuje se poseban tok kontrole koji predstavlja sekvencijalno izvršavanje, a koji se bavi samo tom aktivnošću ili obradom događaja; konstrukcija ovakvog programa je jednostavnija, jer je programer fokusiran na sekvencijalnu obradu samo jedne stvari
  - ❖ ako je potrebno da se jedna aktivnost ili obrada suspenduje, programer to jednostavno kaže (npr. sistemskim pozivom), ne vodi računa o drugim aktivnostima: OS je taj koji će voditi računa o tome da drugi tokovi kontrole dobiju procesor i nastave izvršavanje, a i da suspendovani tok kontrole nastavi kada se za to steknu uslovi
  - ❖ OS vodi računa o tome da uporedne tokove kontrole rasporedi na više procesora, ako postoje, kako bi se izvršavale paralelno, što je efikasnije
- ❖ A zašto se za ovakve potrebe uvek ne koriste procesi?
- ❖ Upravo zato što je potrebno da ovakvi tokovi kontrole obrađuju neke deljene strukture podataka, razmenjuju informacije preko tih podataka i izvršavaju iste potprograme tj. delove koda

Mart 2020.

Copyright 2020 by Dragan Milićev

24



# Pojam niti

Još nekoliko primera upotrebe niti:

- ❖ Korisnik radi u nekom programu za obradu teksta (tekst procesoru). Nakon nekih izmena u dokumentu, korisnik želi da taj dokument sačuva u fajlu i bira komandu za to (*save*). Moguće su dve vrste reakcije programa, u zavisnosti od toga kako je realizovan:
  - ❖ Program obavlja sve potrebne operacije za ovu radnju u istom toku kontrole koji je i detektovao izdavanje komande za snimanje, sekvencijalno: obilazi internu strukturu podataka kojom je dokument predstavljen u memoriji, serijalizuje je (pretvara u niz bajtova), traži uslugu operativnog sistema za snimanje tog niza bajtova u fajl, suspenduje se dok se ta operacija snimanja ne završi (što može da potraje nekoliko sekundi), i tek nakon toga dobija kontrolu nazad i spreman je da detektuje narednu komandu korisnika i reaguje na nju
  - Posledica: tokom obavljanja ove radnje program (proces) je neosetljiv na spoljašnje akcije, ne reaguje na akcije korisnika koji mora da čeka da se ova radnja završi. Da li to mora tako? Zašto bi korisnik čekaao da računar obavlja neke radnje od kojih korisnik ne zavisi?
- ❖ Drugo, bolje rešenje: kada detektuje komandu korisnika, glavni tok kontrole procesa (glavna nit), pokrene opisanu radnju u novoj niti koja onda tu radnju obavlja uporedo, “u pozadini” (*in the background*), jer ne interaguje sa korisnikom; za to vreme glavna nit, ona interaktivna, “u prednjem planu” (*in the foreground*) može da prihvati novu komandu i da reaguje na nju
- Posledica: program ima daleko bolji odziv (*responsiveness*), reaktivniji je (*responsive*)

Mart 2020.

Copyright 2020 by Dragan Milićev

27

# Pojam niti

Na primer, programski jezik Java ima ugrađenu podršku za niti na sledeći način:

- ❖ Skup niti istog tipa (nad istim programskim kodom) koje se mogu kreirati u toku izvršavanja deklarise se kao klasa izvedena iz bibliotečne klase *Thread*; kod (telo) niti definiše se kao polimorfna operacija *run* ove klase

- ❖ U toku izvršavanja programa mogu se kreirati objekti ove klase na uobičajen način; međutim, izvršavanje niti se mora eksplicitno pokrenuti pozivom operacije *start* objekta koji predstavlja tu nit

- ❖ Primer: potrebno je sabrati *n* trodimenzionalnih vektora; sabiranje po svakoj dimenziji može da obavlja posebna nit, pa se postupak može paralelizovati na više procesora

```
class VectorDimAdder extends Thread {
 public VectorDimAdder (double[][][] vect,
 double[] res, int dim) {
 v = vect; r = res; d = dim;
 }
}
```

```
 public void run () {
 r[d] = 0.0;
 for (int i=0; i<v.length; i++)
 r[d] += v[i][d];
 }
}
```

```
 private double[][][] v;
 private double[] r;
 private int d;
```

```
}
```

Ovo su još uvek samo obični objekti ove klase, još uvek nisu pokrenuti novi tokovi kontrole

```
...
double[][][] vect = ...;
double[] res;
```

```
Thread t0 = new VectorDimAdder(vect, res, 0);
Thread t1 = new VectorDimAdder(vect, res, 1);
Thread t2 = new VectorDimAdder(vect, res, 2);
t0.start();
t1.start();
t2.start();
```

Tek sada, nakon ova tri poziva, nastaju tri nova toka kontrole (niti) nad pozivima operacije *run* svakog od ovih objekata

Mart 2020.

Copyright 2020 by Dragan Milićev



# Operacije sa nitima

- ❖ Niti mogu biti podržane konceptima samog programskog jezika, kao što je to slučaj sa jezikom Java ili Ada. Na jeziku Ada, nit se naziva "zadatak" (*task*). Na primer:

```
type Vector3D is array (1..3) of Real;
procedure addVectors (v:array 1..N of Vector3D, result:out Vector3D)
is
 task type VectorDimAdder (dim:Integer) body is
 begin
 result(dim) := 0.0;
 for j in Integer range 1..N loop
 result(dim) := result(dim) + v(j,dim);
 end loop;
 end task;
adders : array (1..3) of access VectorDimAdder;
```

*access* označava referencu na dati tip

```
begin
 for i in Integer range 1..3 loop
 adders(i) := new VectorDimAdder(i);
 end loop;
end procedure;
```

Zadaci se mogu kreirati kao i bilo koji drugi objekti

Procedura *addVectors* čeka na završetak svih zadataka *adders(i)* pre nego što se sama završi

Mart 2020.

Copyright 2020 by Dragan Milićev

29

## Pojam niti

Još nekoliko primera upotrebe niti:

- ❖ Pregledač veb stranica (*web browser*) dovlači jednu HTML stranicu, a ta stranica sadrži kabaste, velike multimedijalne sadržaje (slike). Čim dovuče sam HTML sadržaj stranice, program može da strukturira i prikaže tekstualni sadržaj stranice. Samo dovlačenje referenciranih slika obavlja u posebnim, pozadinskim nitima, što može da bude sporo, posebno ako je veza male propusnosti ili otkazuje; svaka ta nit bavi se dovlačenjem po jednog multimedijalnog fajla i prikazom dovučenog zadržaja. Za to vreme je ona glavna, interaktivna nit u mogućnosti da reaguje na akcije korisnika, pa korisnik može da pomera stranicu (*scroll*), unosi u nju znakove ako treba i na drugi način interaguje sa njom, ili čak aktivira hiperlinkove za nove stranice itd.

- ❖ Serverski sistem koji opslužuje zahteve koji stižu uporedo sa udaljenih klijentskih procesa: sa svakim od tih klijenata treba sprovesti odgovarajući protokol, koji zavisi od stanja, tj. predistorije poruka sa tog klijenta, opsluživati njegove zahteve u zavisnosti od stanja itd. Programiranje ovakve obrade uporednih zahteva sa mnogo klijenata tehnikama sekvencijalnog programiranja je veoma teško. Zato se svakom klijentu, tj. zahtevima sa njega može posvetiti posebna nit, i ona sekvencijalno obrađuje zahteve samo tog klijenta. Kada jedna nit mora da čeka, npr. sledeću poruku sa datog klijenta, OS nju ne raspoređuje na procesor, već procesor izvršava instrukcije drugih niti koje su spremne za izvršavanje. Osim toga, uporedne niti se mogu izvršavati i paralelno, na više procesora koje server svakako poseduje

Mart 2020.

Copyright 2020 by Dragan Milićev

28

# Operacije sa nitima

- ❖ Operativni sistemi takođe podržavaju niti i operacije sa njima odgovarajućim sistemskim pozivima
- ❖ U biblioteci POSIX niti se kreiraju takođe nad funkcijom, kojoj se može dostaviti samo jedan argument tipa *void\**. Ukoliko je funkciji potrebno više argumenata, potrebno ih je složiti u neku strukturu na koju će ukazivati ovaj argument
- ❖ Nit se kreira pozivom sledeće bibliotečne funkcije:  

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void*(*thread_body)(void*), void *arg);
```
- ❖ Prvi parametar ukazuje na strukturu tipa *pthread\_t* koja je deskriptor kreirane niti i koja će kasnije biti korišćena za sve druge operacije u kojima treba identifikovati tu nit. Drugi parametar je pokazivač na strukturu kojom se mogu podešavati atributi niti (npr. vreme aktivacije); ukoliko je ovaj argument *null*, nit se kreira sa podrazumevanim atributima. Treći parametar je pokazivač na funkciju nad kojom se kreira nit, a kojoj se dostavlja, kao jedini argument, parametar *arg*
- ❖ Čekanje na završetak niti može se zahtevati pozivom funkcije *pthread\_join*. Prvi parametar ove funkcije identifikuje nit čiji se završetak čeka. Ovoj funkciji može se dostaviti i drugi argument koji je pokazivač na tip *void\**, u ovaj pokazivač tipa *void\** funkcija koju je izvršavala nit može da prosledi svoj povratni status niti koja izvršava *join* (status koji nit vrati pozivom *pthread\_exit*)

Mart 2020.

Copyright 2020 by Dragan Milićev



31

## Operacije sa nitima

- ❖ I jezik C++, počev od verzije C++11, podržava niti putem standardne biblioteke
- ❖ Klasa *std::thread* definisana u zaglavlju *<thread>* predstavlja niti. Kreiranjem objekta ove klase aktivira se nova nit koja se izvršava nad funkcijom datom parametrom konstruktora ove klase. Nit se pokreće implicitno, odmah po kreiranju objekta klase *thread*, i dalje izvršava nezavisno od niti u kojoj je kreirana i uporedo sa njom
- ❖ Konstruktor ove klase kao svoj prvi parametar očekuje pokazivač na funkciju nad kojom će se kreirati nit (može se dostaviti i objekat klase koja ima preklapjen operator poziva funkcije). Opcioni preostali parametri (može ih biti proizvoljno mnogo) se prosleđuju kao argumenti poziva te funkcije nad kojom se kreira nit
- ❖ Čekanje na završetak pokrenute niti može se zahtevati pozivom operacije *join* objekta te niti, nije dobro pozivati tu funkciju iz više različitih niti. Za isti primer od ranije:

```
#include <thread>
using std::thread;

typedef double Vector3D[3];

void VectorDimAdder (const Vector3D vect[], Vector3D res, int size, int dim) {
 res[dim] = 0.0;
 for (int i=0; i<size; i++) res[dim] += vect[i][dim];
}

void matrixAdder (const Vector3D vect[], Vector3D res, int size) {
 thread* adder0 = new thread(VectorDimAdder, res, size, 0);
 thread* adder1 = new thread(VectorDimAdder, res, size, 1);
 thread* adder2 = new thread(VectorDimAdder, res, size, 2);

 adder0->join();
 adder1->join();
 adder2->join();

 delete adder0; delete adder1; delete adder2;
}
```

Tri nova toka kontrole pokreću se odmah po kreiranju ovih objekata

Nit koja poziva *join* biće suspendovana dok se ne završi nit koju predstavlja objekat čija se funkcija *join* poziva

Mart 2020.

Copyright 2020 by Dragan Milićev

30



# Operacije sa nitima

- ❖ U ovom kursu korišćemo za ilustraciju i jedno malo jezgro, implementirano samo za potrebe nastave - tzv. *školsko jezgro*
- ❖ Školsko jezgro je implementirano na jeziku C++, i zamišljeno je tako da se može instalirati i koristiti neposredno na hardveru, bez podrške operativnog sistema "domaćina"
- ❖ Korišnički kod povezuje se (linkuje) sa kodom jezgra u jedinstvenu celinu, a onda izvršava na ciljnom hardveru
- ❖ Školsko jezgro podržava niti bibliotečnom klasom *Thread*, pri čemu se niti mogu praviti na sledeće načine:
  - ❖ klasično, proceduralno, kao tok kontrole nad funkcijom nečlanicom
  - ❖ slično kao na jeziku Java, kao tok kontrole nad polimorfnom operacijom *run* koju izvedena klasa redefiniše
- ❖ Niti se mora eksplicitno pokrenuti pozivom operacije *start* objekta klase *Thread* koji prestavlja tu nit. Poziv destruktora tog objekta suspenduje pozivajuću nit dok se nit koja se uništava ne završi

```
class Thread {
public:
 Thread (void (*body)(void*));

 void start ();

protected:
 Thread ();
 virtual void run () {}
};
```

Konstruktor za kreiranje niti nad funkcijom nečlanicom

Konstruktor za kreiranje niti nad redefinisanim operacijom *run*

Mart 2020.

Copyright 2020 by Dragan Milićev



33

## Operacije sa nitima

```
#include <pthread>

typedef double Vector3D[3];

struct VectorDimAdderParams {
 const Vector3D* vect; Vector3D* res; int size, dim;
};

void vectorDimAdder (void* params) {
 VectorDimAdderParams* p = (VectorDimAdderParams*)params;
 (*p->res)[p->dim] = 0.0;
 for (int i=0; i<p->size; i++)
 (*p->res)[p->dim] += p->vect[i][p->dim];
}
```

Kada funkcija nad kojom se kreira nit zahteva više parametara, oni se moraju složiti u strukturu

Tada je neophodna i ovakva eksplicitna konverzija

```
void matrixAdder (const Vector3D vect[], Vector3D res, int size) {
 VectorDimAdderParams args0 = {vect,&res,size,0}; pthread_t adder0;
 pthread_create(&adder0, nullptr, &vectorDimAdder,&args0);

 VectorDimAdderParams args1 = {vect,&res,size,1}; pthread_t adder1;
 pthread_create(&adder1, nullptr, &vectorDimAdder,&args1);

 VectorDimAdderParams args2 = {vect,&res,size,2}; pthread_t adder2;
 pthread_create(&adder2, nullptr, &vectorDimAdder,&args2);

 pthread_join(&adder0, nullptr);
 pthread_join(&adder1, nullptr);
 pthread_join(&adder2, nullptr);
}
```

Mart 2020.

Copyright 2020 by Dragan Milićev

32



# Operacije sa nitima

OO varijanta sa kreiranjem niti nad polimotfnom operacijom *run*

```
#include "kernel.h"
typedef double Vector3D[3];

class VectorDimAdder : public Thread {
public:
 VectorDimAdder (const Vector3D* vect, Vector3D* res, int size, int dim)
 : v(vect), r(res), s(size), d(dim) {}

 virtual void run ();

private:
 const Vector3D* v; Vector3D* r; int s, d;
};

void VectorDimAdder::run () {
 (*this->r)[this->d] = 0.0;
 for (int i=0; i<this->s; i++)
 (*this->r)[this->d] += this->v[i][this->d];
}

void matrixAdder (const Vector3D vect[], Vector3D res, int size) {
 Thread* t0 = new VectorDimAdder(vect,&res,size,0);
 t0->start();

 Thread* t1 = new VectorDimAdder(vect,&res,size,1);
 t1->start();

 Thread* t2 = new VectorDimAdder(vect,&res,size,2);
 t2->start();

 delete t0;
 delete t1;
 delete t2;
}
```

Mart 2020.

Copyright 2020 by Dragan Milićev

35

# Operacije sa nitima

Varijanta sa kreiranjem niti nad funkcijom nečlanicom

```
#include "kernel.h"
typedef double Vector3D[3];

struct VectorDimAdderParams {
 const Vector3D* vect; Vector3D* res; int size, dim;
};

void vectorDimAdder (void* params) {
 VectorDimAdderParams* p = (VectorDimAdderParams*)params;
 (*p->res)[p->dim] = 0.0;
 for (int i=0; i<p->size; i++)
 (*p->res)[p->dim] += p->vect[i][p->dim];
}

void matrixAdder (const Vector3D vect[], Vector3D res, int size) {
 VectorDimAdderParams args0 = {vect,&res,size,0};
 Thread* t0 = new Thread(&vectorDimAdder,&args0);
 t0->start();

 VectorDimAdderParams args1 = {vect,&res,size,1};
 Thread* t1 = new Thread(&vectorDimAdder,&args1);
 t1->start();

 VectorDimAdderParams args2 = {vect,&res,size,2};
 Thread* t2 = new Thread(&vectorDimAdder,&args2);
 t2->start();

 delete t0;
 delete t1;
 delete t2;
}
```

Destruktor podrazumeva čekanje na to da se ciljna nit završi

Mart 2020.

Copyright 2020 by Dragan Milićev

34

# Operacije sa nitima

- ❖ Školsko jezgro obezbeđuje još jedan jednostavan sistemski poziv kojim pozivajuća nit “traži” od jezgra (ili mu daje priliku) da izvrši jednostavnu promenu konteksta:  
`void dispatch ();`
- ❖ Ovaj sistemski poziv realizovan je samo radi ilustracije, posebno implementacije ove najjednostavnije situacije promene konteksta u sistemskom kodu (detalji kasnije)
- ❖ On se može pozvati na bilo kom mestu u telu niti. Na primer:  

```
void VectorDimAdder::run () {
 (*this->r)[this->d] = 0.0;
 dispatch();
 for (int i=0; i<this->s; i++) {
 (*this->r)[this->d] += this->v[i][this->d];
 dispatch();
 }
}

void matrixAdder (const Vector3D vect[], Vector3D res, int size) {
 Thread* t0 = new VectorDimAdder(vect, &res, size, 0);
 t0->start(); dispatch();

 Thread* t1 = new VectorDimAdder(vect, &res, size, 1);
 t1->start(); dispatch();

 Thread* t2 = new VectorDimAdder(vect, &res, size, 2);
 t2->start(); dispatch();

 delete t0; delete t1; delete t2;
}
```

Mart 2020.

Copyright 2020 by Dragan Milićev

37

# Operacije sa nitima

- ❖ Proceduralni, klasičan način kreiranja niti nad globalnom funkcijom po pravilu omogućava prenos argumenta (obično nekog krajnje opšteg tipa, poput tipa *void\** ili *int*). Zašto?
- ❖ Zato što bi bez toga bilo nemoguće kreirati više niti nad istom funkcijom, parametrizovanih tako da rade različite stvari, recimo nad različitim podacima ili delovima struktura podataka, kao u korišćenim primerima
- ❖ Ako je funkciji potrebno više parametara, potrebno ih je složiti u strukturu i pokazivač na tu strukturu preneti funkciji, koja te parametre onda “raspakuje” iz date strukture
- ❖ U OO varijanti, nit se kreira nad polimorfnom operacijom *run* koja *nema* parametre, a opet je korisna, jer se izvršavanje niti ipak može parametrizovati. Na primer, u prikazanim ilustracijama funkcija *run* je pristupala podacima članovima objekta u čijem kontekstu se izvršava, pa su ti podaci članovi bili parametri izvršavanja niti - svaka nit je radila sa svojom dimenzijom vektora
- ❖ Kako to da se sada ista stvar, parametrizacija izvršavanja niti, dobila “ni iz čega”, tj. unutar funkcije *run* bez parametara, a u proceduralnoj varijanti bio je neophodan parametar funkcije nad kojom se nit kreira?
- ❖ Odgovor je u tome što nestatička funkcija članica klase, kakva je i funkcija *run*, zapravo *ima* jedan implicitan, skriven parametar *this*: svaki direktan pristup nestatičkom podatku članu objekta prevodi u indirektan pristup preko pokazivača *this*
- ❖ To znači da zapravo struktura podataka članova, a tako se objekat i implementira u memoriji, predstavlja strukturu argumenata, dok je *this* taj jedini parametar, i to baš tipa pokazivača

- ❖ Prema tome, mehanizam u pozadini kojim se sve ovo implementira je zapravo identičan u obe varijante, jer u obe varijante nit započinje izvršavanje funkcije sa jednim parametrom - pokazivačem na strukturu, samo što u OO varijanti neophodne provere tipova i konverzije radi prevodilac, pa je sve lepše “upakovano” i skriveno od programera

Mart 2020.

Copyright 2020 by Dragan Milićev

36

# Operacije sa nitima

- ❖ Prema tome:
    - ❖ promena konteksta se može, ali i ne mora dogoditi u pozivu funkcije *dispatch*
    - ❖ promena konteksta se može dogoditi i u pozivu *dispatch*, ali i na bilo kom drugom mestu, odnosno u bilo kom drugom trenutku izvršavanja
- Iz ovoga sledi da *dispatch* zapravo nema nikakvog smisla, jer nema nikakvo posebno značenje!
- ❖ Zato operativni sistemi po pravilu i ne obezbeđuju ovakve jednostavne pozive: nema nikakvog smisla, a ni potrebe, da proces sam traži promenu konteksta. Ako i kad je to potrebno, sistem će to uraditi implicitno, a proces u sistemskom pozivu traži neku drugu, konkretnu uslugu. Samo promena konteksta i nije nikakva usluga, jer nema mnogo praktičnog smisla: zašto bi proces uopšte tražio da ne nastavlja dalje svoje izvršavanje ako za to nema razloga, tj. iako može da nastavi?
  - ❖ Ova usluga je u školskom jezgriu napravljena iz dva razloga:
    - ❖ u jednostavnijoj izvedbi, jezgro vrši samo sinhronu promene konteksta, a ne i asinhronu, pa bi izvršavanje korisničkih niti bilo “neinteresantno” bez ikakvih sistemskih poziva, jer bi se promene konteksta dešavale isključivo onda kada neka nit pozove neku uslugu jezgra, ili tek kada se završi, ako nit to ne uradi
    - ❖ radi demonstracije implementacije ovog najjednostavnijeg slučaja sistemskog poziva, kada jezgro ne treba da uradi ništa drugo već samo promenu konteksta

Mart 2020.

Copyright 2020 by Dragan Milićev

39

## Operacije sa nitima

- ❖ Međutim, kakav je zapravo smisao ovog sistemskog poziva?
- ❖ Sa jedne strane, kada kernel dobije kontrolu u ovom sistemskom pozivu, on može, ali uopšte *ne mora* da izvrši promenu konteksta, njegovo je “diskreciono pravo” da o tome odluči: pitanje je da li uopšte ima drugih niti koje su spremne za izvršavanje (možda sve ostale nešto čekaju), pitanje je kakav je algoritam raspoređivanja (možda taj algoritam opet izabere istu nit) i da li kernel uopšte hoće da izvrši promenu konteksta; prema tome, nakon ovog sistemskog poziva, može se dogoditi i to da ista pozivajuća nit nastavi svoje izvršavanje na istom procesoru
- ❖ Sa druge strane, ukoliko je kernel napravljen tako da vrši preotimanje, tj. da se promena konteksta može napraviti i *asinhrono*, u bilo kom, nepredvidivom trenutku, nezavisno od toga da li je tekući tok kontrole pozvao sistemski poziv ili ne, tj. kao posledica spoljašnjeg događaja, onda se promena konteksta može dogoditi ne samo u pozivu *dispatch*, nego i u bilo kom drugom delu izvršavanja niti
- ❖ U opštem slučaju, ako se želi postići prenosiv i nezavisan kod, ne sme se ništa pretpostaviti o implementaciji kernela; tačnije, mora se pretpostaviti opštiji slučaj, taj da kernel vrši i asinhronu promene konteksta, a ne samo sinhronu (kada proces pozove sistemski poziv ili izazove izuzetak)

Mart 2020.

Copyright 2020 by Dragan Milićev

38



# Mehanizam prekida

- ❖ Preotimanje (*preemption*): situacija u kojoj procesor izvršava instrukcije jednog procesa, dogodi se nešto zbog čega neki drugi proces može da nastavi izvršavanje i on treba da preuzme procesor što pre, jer je njegova reakcija važnija od onoga što radi tekući proces, ne čekajući da se tekući proces sam odrekne procesora sistemskim pozivom
- ❖ Jedna ovakva situacija je i istek vremenskog odsečka (kvanta) koji je dodeljen tekućem procesu za izvršavanje na procesoru u jednom naletu kod sistema sa vremenskom raspodelom
- ❖ Kako podržati preotimanje?
- ❖ Da bi se izvršila promena konteksta, neophodno je da procesor pređe na instrukcije koje pripadaju kodu kernela i koje izvršavaju promenu konteksta - promena konteksta nije ništa drugo nego izvršavanje instrukcija kernela koje treba da prebace procesor sa izvršavanja jednog dela koda na izvršavanje drugog dela koda u operativnoj memoriji, uz promenu memorijskog konteksta (promenu vrednosti *SMTTP/PMTP*)
- ❖ Bez dodatnog mehanizma u odnosu na ono što je do sada razmatrano, dok procesor izvršava instrukcije nekog procesa, OS ne može dobiti kontrolu, pa tako ni izvršiti promenu konteksta i procesor prebaciti na izvršavanje instrukcija drugog procesa dok se ne dogodi nešto od sledećeg:
  - ❖ tekući proces pozove neki sistemski poziv
  - ❖ instrukcija tekućeg procesa izazove neki hardverski izuzetak
- ❖ Međutim, ovo se može dogoditi neodređeno odloženo, ili možda nikad: šta ako proces, zbog greške ili izvršavanjem malicioznog koda, ne uradi ovo nikad (npr. izvršava beskonačnu petlju) ili uradi mnogo odloženo?

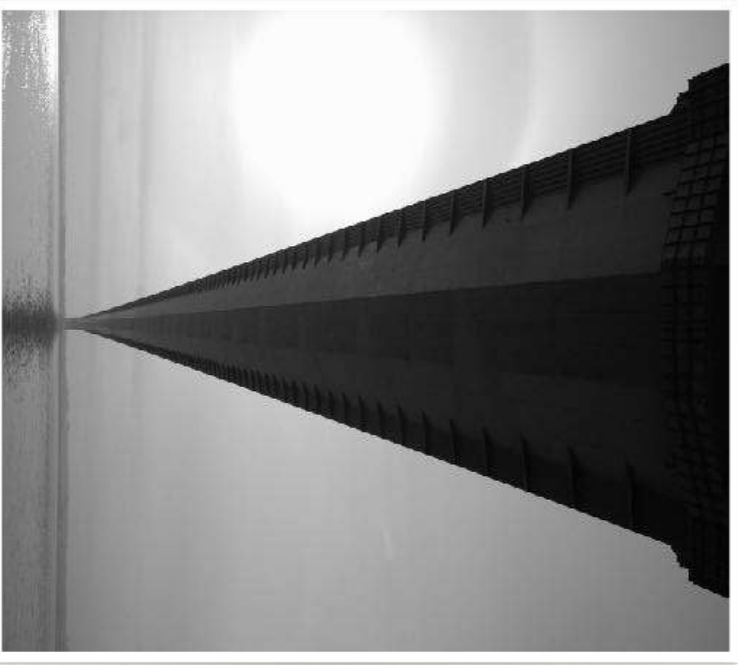
Mart 2020.

Copyright 2020 by Dragan Milićev

41

## Glava 8: Implementacija procesa

- ❖ Mehanizam prekida
- ❖ Kontekst i stanja procesa
- ❖ Promena konteksta
- ❖ Implementacija niti
- ❖ Raspoređivanje

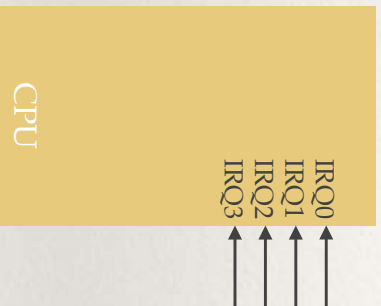


Mart 2020.

Copyright 2020 by Dragan Milićev

40

# Mehanizam prekida



```
load r0, [r1]
add r0, r2
load r2, [r1+0x20]
mul r0, r2
store r0, [r1]
...
...
...
...
...
push r0,
load r0, running
...
pop r0
iret
```

Mart 2020.

Copyright 2020 by Dragan Milićev

43

## Mehanizam prekida

- ❖ Čak i ako proces prepusti kontrolu kernelu, sistemskim pozivom, kako da kernel ispita da li se dogodilo nešto što treba da se dogodi (događaj, istek vremena)?
- ❖ Jedan način je da procesor izvršava instrukcije kojima ispituje da li se to dogodilo, čitanjem vrednosti nekih registara u U/I adresnom prostoru; procesor eventualno čita i ispituje izvršavajući instrukcije u petlji - *uposljeno čeka (busy wait)* ili *proziva (polling)* - neefikasno, jer procesor izvršava "jalove" i beskorisne instrukcije ispitivanja i čekanja da se nešto dogodi
- ❖ Rešenje: procesor može da radi neki koristan posao, da izvršava instrukcije procesa ili čak i kernela, a događaj se signalizira posebnim hardverskim signalom koji predstavlja spoljašnji *zahtev za prekid (interrupt request)* - zahtev dolazi signalom koji spolja ulazi u procesor, a potiče od hardverskih uređaja, a ne iz samog procesora. Zahtev za prekid mogu tako generisati:
  - ❖ ulazno-izlazni uređaji koji signaliziraju različite događaje: završena jedna U/I operacija, greška u prenosu i slično
  - ❖ vremenski brojači ili *tajmeri (timer)*: hardverski uređaji koji mogu da mere vremenske intervale i generišu prekide periodično ili po isteku zadatog vremenskog intervala
- ❖ Kada stigne zahtev za prekid, procesor završava tekuću instrukciju, i u principu radi isto što i kod obrade izuzetka: čuva kontekst (neke od programskih dostupnih registara) na steku i prelazi na izvršavanje posebnog programa za obradu prekida - *prekidne rutine (interrupt routine)*
- ❖ Prekidna rutina je deo koda kernela, za obradu prekida od odgovarajućeg izvora, na isti način kao i za izuzetke i sistemske pozive
- ❖ Važno: procesor uvek završava izvršavanje tekuće instrukcije pre nego što pređe na obradu spoljašnjeg zahteva za prekid - izvršavanje instrukcije je atomično, a tu atomičnost obezbeđuje hardver procesora! (U suprotnom, programiranje bi bilo jako teško jer efekti delimično izvršene instrukcije ne bi bili predvidivi)
- ❖ Kada završi prekidnu rutinu, procesor se vraća na mesto gde je prekinuto izvršavanje, kao iz najobičnijeg potprograma

Mart 2020.

Copyright 2020 by Dragan Milićev

42



# Mehanizam prekida

- ❖ Kako se određuje broj ulaza u vektor tabeli pridružen spojašnjem prekidu, a na osnovu koga se iz vektor tabele čita adresa prekidne rutine? Mogući su različiti pristupi:
  - ❖ statički, tako što je hardverom predodređeno i nepromenljivo preslikavanje ulaznih signala zahteva za prekid u brojeve ulaza u vektor tabeli
  - ❖ dinamički, na sledeći način: prilikom prihvatanja prekida sa određene linije zahteva za prekid (*IRQ*), procesor postavlja odgovarajući signal uparen sa ovim (potvrda prihvatanja, *interrupt acknowledgment*); ovaj signal vezan je za trostatičke baferne na izlazu nekog registra kontrolera prekida ili periferijskog uređaja, koji na magistralu podataka tako postavlja vrednost iz tog registra; procesor tu vrednost očita sa magistrale i koristi kao broj ulaza
  - ❖ Dinamički pristup je danas češći i omogućava da se dinamički, softverski, periferijskim uređajima koji generišu prekide dinamički raspodeljuju ulazi u vektor tabeli
  - ❖ Ovo radi OS prilikom inicijalizacije sistema, ali i kasnije, kada se sistemu "registruje" neku uređaj, kako će biti objašnjeno kasnije: OS vodi evidenciju o zauzetim i slobodnim ulazima za prekide i dodeljuje neki slobodan kada se nov uređaj instalira, upisom u odgovarajući registar, odnosno konfiguracijom kontrolera prekida

Mart 2020.

Copyright 2020 by Dragan Milićev

45

# Mehanizam prekida

- ❖ Procesor ovo realizuje tako što:
  - ❖ postojanje spojašnjih signala zahteva za prekid pamti u internim flip-flopovima
  - ❖ na kraju obrade svake instrukcije, proverava postojanje ovih signala i:
    - ❖ ako ne postoje, ili ako nisu ispunjeni uslovi prihvatanja prekida, prelazi na dohvatanje sledeće instrukcije na koju ukazuje *PC*
    - ❖ ako postoje i ako su ispunjeni uslovi prihvatanja prekida, vrši obradu prekida
  - ❖ Obrada prekida se, u principu, obavlja na isti način kao i za interne izuzetke ili sistemske pozive:
    - ❖ procesor sačuvava određene registre, po pravilu na steku
    - ❖ procesor odredi adresu prekidne rutine, preko vektor tabele, za broj ulaza pridružen prekidu
    - ❖ procesor smešta u *PC* određenu adresu prekidne rutine i prelazi na sledeću instrukciju na toj adresi
  - ❖ Povratak iz prekidne rutine obavlja se posebnom instrukcijom koja ima iste efekte kao i instrukcija povratka iz sistemskog poziva - na isti način se tretiraju:
    - ❖ sa steka se restauriraju oni registri koji su implicitno sačuvani pri prihvatanju prekida, a time i *PC* u koji se smešta adresa instrukcije na koju se vrši povratak

Mart 2020.

Copyright 2020 by Dragan Milićev

44



# Mehanizam prekida

Prema tome, načini na koje procesor može preći na izvršavanje kernel koda, odnosno prekinuti izvršavanje tekućeg procesa i preći na kernel kod opisanim mehanizmom koji je jednobrazan za sve ove situacije:

- ❖ Sinrono, kao posledica onoga što je tekuća instrukcija uradila:
  - ❖ izuzetak (tj. interni hardverski prekid), greška u izvršavanju instrukcije, zbog koje ta instrukcija ne može da se izvrši do kraja i prekida se pre svog završetka:
    - ❖ nelegalan kod operacije, nedozvoljen način adresiranja, nedozvoljena instrukcija u nepriviligovanom režimu itd.
    - ❖ aritmetička greška: prekoračenje, deljenje nulom i slično
    - ❖ pristup u adresiranju memorije: povreda prava pristupa ili stanična greška (*page fault*)
    - ❖ i slično
  - ❖ sistemski poziv izazvan instrukcijom softverskog prekida
- ❖ Asinhrono, nezavisno od toga što radi tekuća instrukcija, u potpuno nepredvidivim trenucima, kao posledica spoljašnjeg hardverskog prekida koji dolaze od:
  - ❖ ulazno-izlaznih uređaja (tipično su maskirajući): signaliziraju završetak operacije, grešku u operaciji i slično
  - ❖ vremenskih brojača (*timer*), periodično ili u određeno vreme (tipično su maskirajući)
  - ❖ uređaja za nadzor ispravnosti rada hardvera (tipično su nemaskirajući): pad napona napajanja ili ispražnjena baterija, greška u kontroli parnosti sadržaja memorijske ćelije itd.

Mart 2020.

Copyright 2020 by Dragan Milićev

47

## Mehanizam prekida

- ❖ Procesori po pravilu omogućuju da se programskim putem, odgovarajućim instrukcijama, zabrane spoljašnji prekidi – tzv. *maskiranje prekida* (*interrupt masking*)
- ❖ Ovo se implementira tako što u procesoru postoji poseban programski dostupan jednobitni registar koji svojim stanjem dozvoljava ili ne dozvoljava prihvatanje spoljašnjih prekida
- ❖ Ovim registrom obično manipulišu posebne instrukcije, npr. tipa:
  - int  
intd
  - Onemogućavanje (*inte - interrupt enable*) ili demaskiranje (*immask*) prekida
  - Onemogućavanje (*intd - interrupt disable*) ili maskiranje (*imask*) prekida
- ❖ Neki procesori omogućuju i selektivno maskiranje prekida sa pojedinačnih linija: poseban programski dostupan registar maske (*interrupt mask register, IMR*) svakim svojim razredom omogućava ili ne prekid sa određene linije zahteva za prekid, a upis u ovaj registar utiče na te bite
- ❖ Prekid se prihvata samo ako nije selektivno ili globalno maskiran; ako jeste, prekid se prosto ignoriše, kao da ga nije ni bilo i procesor nastavlja izvršavanje na podrazumevani način
- ❖ Važno: naravno, ove instrukcije koje utiču na registre za maskiranje prekida, odnosno na to da li su spoljašnji prekidi dozvoljeni ili ne, dostupne su samo u privilegovanom, sistemskom režimu rada procesora
- ❖ Detalj koji je važan za implementaciju kernela: da bi se prekidnoj rutini pružila prilika da neke operacije uradi atomično, "na miru i bez cimanja", tj. bez daljih prekida, prilikom prihvatanja prekida procesor po pravilu implicitno maskira spoljašnje prekide; prekidi se maskiraju da se ne bi ugnežđivali – prekidna rutina je podrazumevano (ali ne uvek) atomična – atomičnost obezbeđuje hardver; ako to u prekidnoj rutini nije neophodno, ona uvek može demaskirati prekide odgovarajućom instrukcijom

Mart 2020.

Copyright 2020 by Dragan Milićev

46

# Kontekst i stanja procesa

- ❖ Ove informacije OS organizuje u odgovarajuće strukture podataka kojima predstavlja procese u sistemu
- ❖ Način organizacije ovih struktura podataka može da bude najrazličitiji: strukture podataka treba organizovati tako da odgovaraju operacijama sa njima
- ❖ Tradicionalno, struktura podataka kojom se predstavlja proces u sistemu naziva se *kontrolni blok procesa* (*process control block*, *PCB*)
- ❖ Ovaj tradicionalni naziv vodi poreklo iz vremena kada su sistemi bili skromnih mogućnosti, a memorija malog kapaciteta, pa su strukture podataka bile jednostavne, linearno složene u memoriju redom, a o svakom bajtu i bitu se vodilo računa, pa su se te strukture predstavljale kao “tabele” ili “blokovi”, u kojima se definisalo značenje svake reči, bajta ili bita

|                           |
|---------------------------|
| <i>PID</i>                |
| <i>Processor Context</i>  |
| <i>Scheduling Context</i> |
| <i>Memory Context</i>     |
| <i>Resources</i>          |
| <i>Accounting</i>         |
| <i>Misc</i>               |

Mart 2020.

Copyright 2020 by Dragan Milićev

# Kontekst i stanja procesa

- ❖ Da bi podržao izvršavanje procesa na jednom ili više procesora i sa odgovarajućim virtuelnim adresnim prostorima, OS mora da rukuje procesom kao entitetom, logičkim objektom, odnosno da u svom memorijskom prostoru organizuje strukture podataka koje predstavljaju procese i čuvaju informacije o svakom procesu
- ❖ Informacije koje OS vodi o svakom procesu nazivaju se i *kontekstom procesa* (*process context*) i sadrže svojstva (atribute) svakog procesa:
  - ❖ identifikator procesa (*process ID*, *pid*): jedinstveni identifikator procesa u sistemu (najčešće jednostavan ceo broj)
  - ❖ *kontekst procesora* (*processor context*) ili *kontekst izvršavanja* (*execution context*): vrednosti svih registara programski dostupnih u neprivilegovanom režimu, sačuvane u trenutku kada je taj proces izgubio procesor, a potrebno ih je restaurirati kada taj proces bude ponovo dobio procesor
  - ❖ informacije potrebne za raspoređivanje procesa na procesoru (*scheduling context*): prioritet, velična vremenskog odsečka koji mu se dodeljuje za izvršavanje i drugo
  - ❖ *memorijski kontekst* (*memory context*) za virtuelni adresni prostor procesa: deskriptori logičkih segmenata, bazna adresa i granica, ili tabele preslikavanja i vrednost *PMTT/SMTP*
  - ❖ deskriptori resursa koje je proces alocirao (tražio i dobio na korišćenje), kao što su otvoreni fajlovi, ulazno-izlazni uređaji i drugi fizički i logički resursi operativnog sistema
  - ❖ “knjigovodstvo” (*accounting*): evidencija korišćenja resursa računara i operativnog sistema (iskorišćeno procesorsko vreme, memorija itd.) ako je potrebno izveštavati o tome i naplaćivati usluge
  - ❖ sve druge potrebne informacije, kao što su informacije o tekućem direktorijumu, korisniku u čije ime se proces izvršava i druge

Mart 2020.

Copyright 2020 by Dragan Milićev



# Kontekst i stanja procesa

- ❖ Konceptualno, tokom svog životnog veka, tj. za vreme od trenutka kada je traženo njegovo kreiranje, pa dok se ne ugasi, svaki proces prolazi kroz određena *stanja* (*state*):
  - ❖ stanje *inicijalizacije* (*initializing*): od trenutka kada je neki proces zahtevao kreiranje novog procesa, OS mora da obavi svu potrebnu inicijalizaciju struktura podataka kojima predstavlja taj proces, da kreira njegov memorijski kontekst, inicijalni kontekst izvršavanja, pre nego što taj proces postane *izvršiti*
  - ❖ *terminalno* stanje (*terminating*): od trenutka kada je traženo gašenje procesa (na bilo koji od navedenih načina), dok on sasvim ne nestane iz operativnog sistema kao entitet (objekat), OS mora da oslobodi resurse koje je proces koristio, dealocira memoriju, kao i sve druge strukture koje je OS koristio za predstavljanje procesa
- Između ova dva stanja, početnog i krajnjeg, proces tokom svog aktivnog života menja sledeća principijelna stanja:
  - ❖ *izvršava se* (*running*): u ovom stanju je onaj jedan proces (na jednoprocorskom sistemu) ili više njih (na multiprocorskom sistemu, za svaki procesor po jedan takav proces) koji se trenutno izvršava na procesoru, odnosno koji je “tekući”, čije instrukcije izvršava procesor (bolje reći: u čijem kontekstu procesor izvršava instrukcije)

Mart 2020.

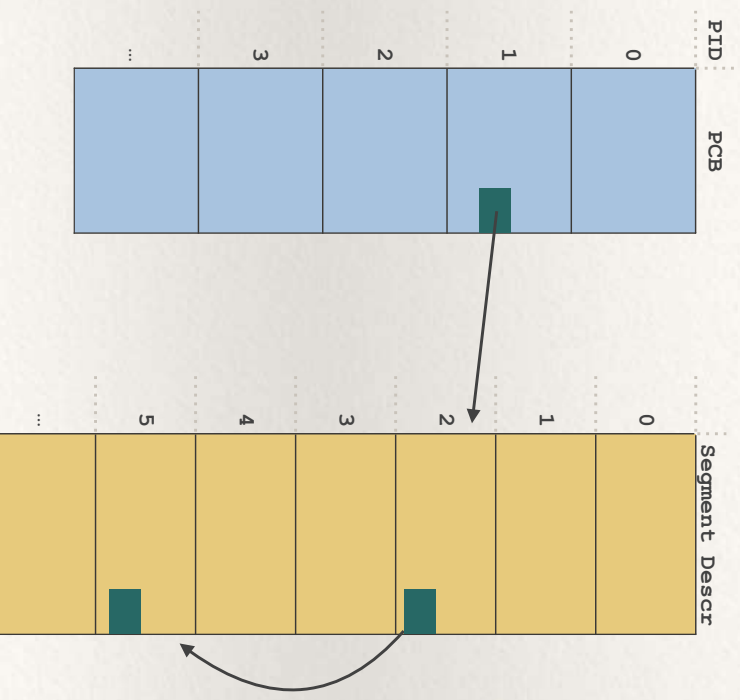
Copyright 2020 by Dragan Milićev



51

## Kontekst i stanja procesa

- ❖ Danas su te strukture složenije, često nelinearne (ne samo proste strukture ili nizovi) i dinamičke, iako se i dalje nazivaju ovako tradicionalno, pa se u terminologiji operativnih sistema često koriste nazivi “tabela” ili “blok”
- ❖ Na primer, PCB može biti prosta struktura, eventualno sa pokazivačima na druge strukture, a te strukture mogu biti složene u nizove, radi efikasnijeg korišćenja memorije za potrebe operativnog sistema
- ❖ Više detalja o organizaciji struktura podataka i alokaciji prostora za potrebe samog kernela u predmetu OS2



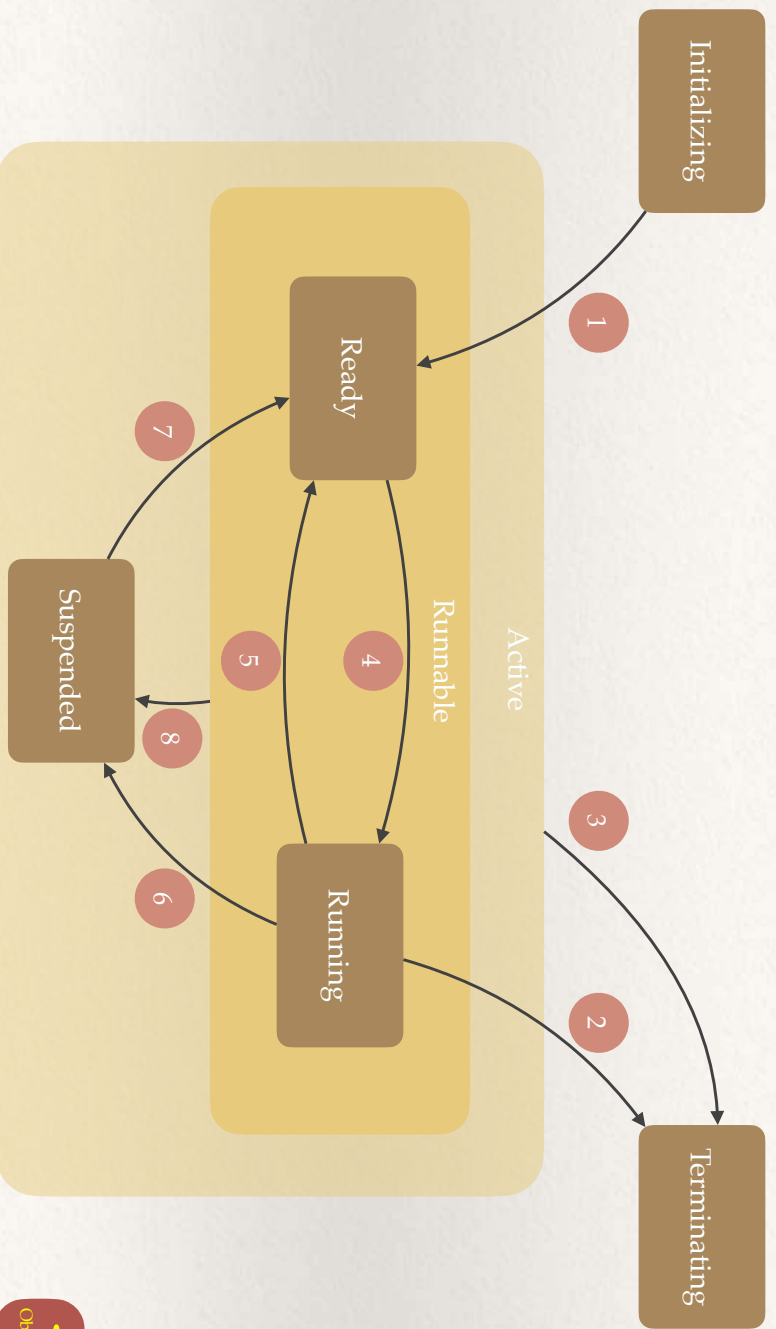
Mart 2020.

Copyright 2020 by Dragan Milićev

50



# Kontekst i stanja procesa



Mart 2020.

Copyright 2020 by Dragan Milićev



53

## Kontekst i stanja procesa

### ❖ Stanja procesa (nastavak):

- ❖ *spreman za izvršavanje (ready)*: u ovom stanju su oni procesi koji su spremni za izvršavanje, zato što imaju ispunjene sve uslove nastavka svog izvršavanja, samo se trenutno ne izvršavaju, jer nemaju procesor; iz ovog skupa spremnih procesa OS može izabrati jedan (bilo koji) kome će predati procesor prilikom promene konteksta

Ova dva stanja, spremnosti i izvršavanja, ponekad se nazivaju jedinstvenim nazivom, stanjem *izvršivosti (runnable)*

- ❖ *suspendovan ili blokiran, čeka (suspended, blocked, waiting)*: proces čeka na ispunjenje uslova nastavka svog izvršavanja, iz različitih razloga; na primer, proces može da čeka na:
  - ❖ završetak zahtevane ulazno-izlazne operacije ili nekog drugog sistemskog poziva koji je izvršio
  - ❖ završetak obrade stranične greške ili da ponovo bude ubačen u memoriju (*swapping*)
  - ❖ ispunjenje nekog logičkog uslova
  - ❖ neki događaj
- ❖ da istekne neko zadato vreme, ili da dođe neko određeno vreme, na primer vreme aktivacije periodičnog procesa



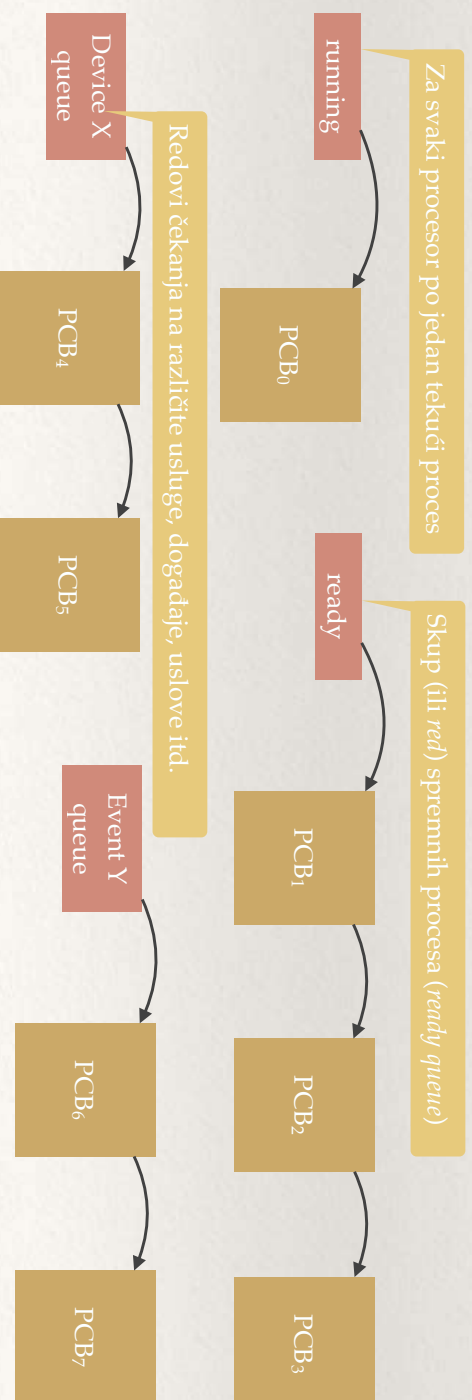
Mart 2020.

Copyright 2020 by Dragan Milićev

52

# Kontekst i stanja procesa

- ❖ Ovo je konceptualan prikaz, a informaciju o stanju OS može implementirati na najrazličitije načine, na primer:
  - ❖ eksplicitno, atributom u PCB-u koji opisuje tekuće stanje,
  - ❖ implicitno, organizovanjem PCB-ova u strukture koje grupišu procese u određenom stanju; jedan kranje jednostavan, nikako jedini niti najbolji način - ulančane liste
- ❖ Implementacija promena stanja procesa svodi se tako na promenu vrednosti atributa ili premeštanje strukture PCB iz jedne u drugi red/ skup - jednostavne i uobičajene operacije sa strukturama podataka



Mart 2020.

Copyright 2020 by Dragan Milićev

55

## Kontekst i stanja procesa

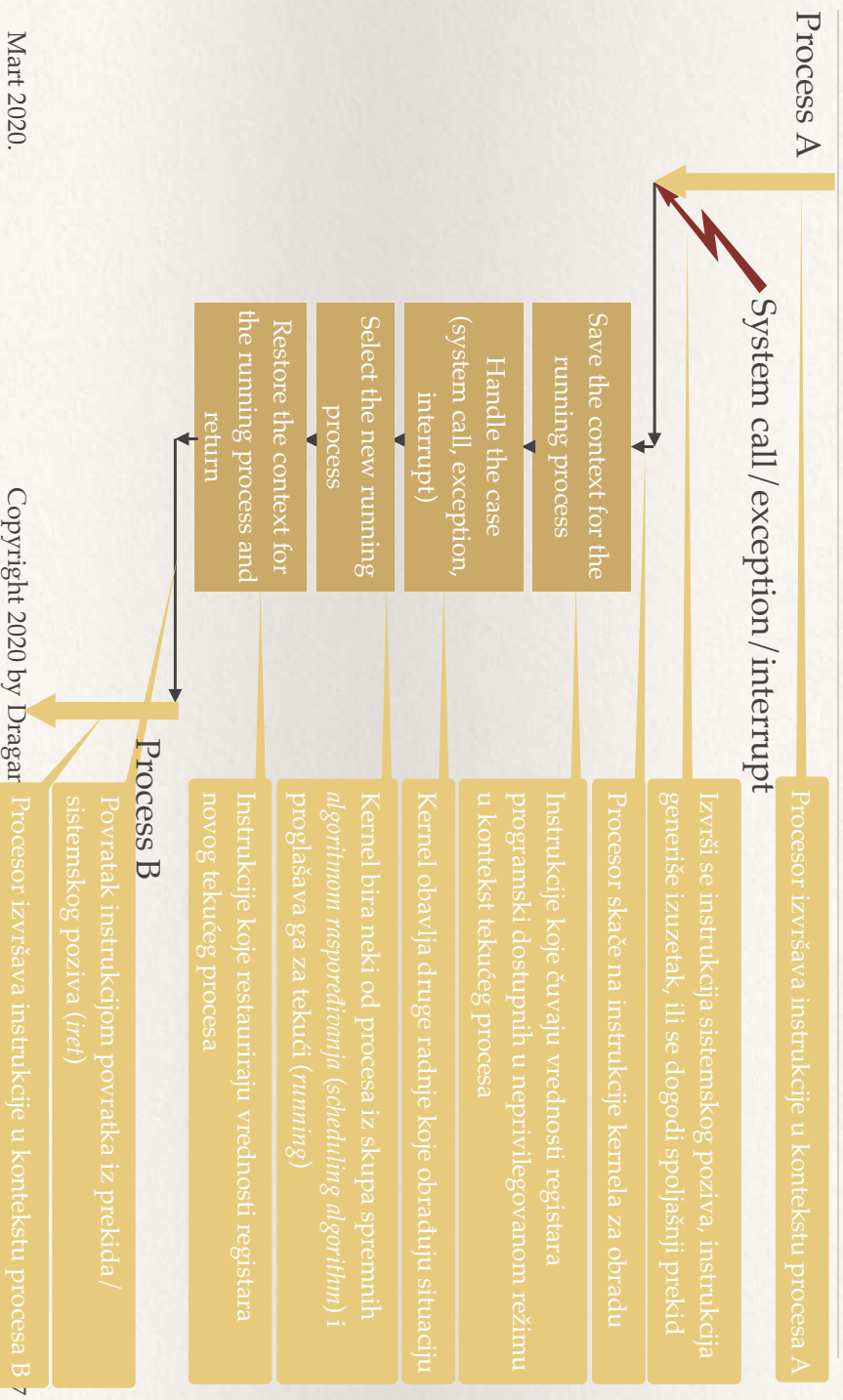
- ❖ Prelazi (numeracija je samo u svrhu referenciranja, ne implicira nikakav poredak):
- 1. OS je završio inicijalizaciju procesa i napravio njegov memorijski i procesorski kontekst, proces može da počne izvršavanje
- 2. Ovaj proces je, kao tekući, izazvao izuzetak od kog nema oporavka ili pozvao sistemski poziv *exit*
- 3. Neki drugi proces pozvao sistemski poziv *kill* za gašenje ovog procesa
- 4. OS izabrao ovaj proces za izvršavanje algoritmom raspoređivanja (*scheduling*)
- 5. a) Sinhrono: ovaj proces, kao tekući, pozvao *neblokirajući* sistemski poziv - proces ne treba da se suspenduje, ali OS dobija priliku da izvrši promenu konteksta i izabere drugi proces za izvršavanje, dok ovaj ostaje i dalje spreman za izvršavanje, može da nastavi bilo kada
- b) Asinhrono: desio se spoljašnji prekid zbog nekog događaja ili isteklog vremena, OS dobija priliku da izabere neki drugi proces (možda je neki ovim i postao spreman) - preotimanje (*preemption*)
- 6. Ovaj proces je, kao tekući, pozvao neki *blokirajući* sistemski poziv (npr. *wait* ili *join*) ili izazvao izuzetak koji ga suspenduje (npr. straničnu grešku koja zahteva dovlačenje stranice sa diska)
- 7. Ispunjen uslov na koji je suspendovani proces čekao ili je prestao razlog za suspenziju: završena operacija ili sistemski poziv, završena obrada stranične greške, ispunjen logički uslov, OS odlučio da ponovo učita proces u memoriju (*swap in*), dogodio se događaj, isteklo vreme čekanja ili došlo vreme aktivacije
- 8. OS odlučuje da ovaj proces, kao tekući ili spreman, izbac iz memorije (*swap out*) i suspenduje na duže vreme

Mart 2020.

Copyright 2020 by Dragan Milićev

54

# Promena konteksta



## Promena konteksta

- ❖ Načini na koje dolazi do *promene konteksta* (*context switch*):
  - ❖ *Sinhrono*, kao posledica izvršavanja same tekuće instrukcije procesa:
    - ❖ “voljno”, eksplicitno: proces je pozvao sistemski poziv; ako je poziv blokirajući, OS svakako treba da odabere neki drugi proces za izvršavanje; ako je poziv neblokirajući, isti proces može da nastavi izvršavanje, ali i ne mora, OS može da odabere i neki drugi
    - ❖ “nevoljno”, implicitno: instrukcija je izazvala neki hardverski izuzetak koji OS mora da obradi na odgovarajući način
  - ❖ *Asinhrono*, potpuno nezavisno od tekuće instrukcije i onoga što ona radi, u proizvoljnim, nepredvidivim trenucima vremena, kao posledica *spoljašnjeg hardverskog prekida* (*interrupt*)
  - ❖ Sve ove situacije procesor obrađuje istim mehanizmom, skokom na rutinu na koju ukazuje odgovarajući vektor u vektor tabeli, pri čemu je broj ulaza u vektor tabeli određen vrstom izuzetka, izvorom spoljašnjeg prekida, ili parametrom u instrukciji softverskog prekida (sistemskog poziva)
- ❖ Kao posledica toga, procesor:
  - ❖ prelazi u privilegovani režim rada i najčešće prebacuje izvršavanje na drugi, sistemski stek na koji ukazuje sistemski SP
  - ❖ čuva određene programski dostupne registre prepisujući ih negde, npr. na stek
    - ❖ u PC upisuje adresu koju je dohvatio iz vektor tabele za dati ulaz; to je adresa sledeće instrukcije koju će izvršiti
- ❖ Ova instrukcija na koju procesor skače je prva instrukcija prekida rutine i svakako pripada kodu kernela koji obrađuje dati sistemski poziv, izuzetak ili prekid
- ❖ Nakon obrade te situacije, OS mora da *povrati kontekst* procesa koji je odabran za izvršavanje (novi tekući proces) i vratiti se na njegovo izvršavanje, tj. u njegov kontekst, tako da on nastavlja od mesta na kom je prekinut - *promena konteksta* (*context switch*)



# Promena konteksta

- ❖ Primer 1, najjednostavniji slučaj: procesor ima dva skupa registara *PC*, *SP* i *PSW*, jedan skup koristi u nepriviligovanom, drugi u priviligovanom režimu rada; ima i jedan registar *RX* koji je dostupan *samo* u priviligovanom režimu, tako da ga proces sigurno ne koristi:

```
; Save the current context
load rx,running
store r0,[rx+offsR0] ; save regs
store r1,[rx+offsR1]
...
store r15,[rx+offsR15]
store pc,[rx+offsPC] ; save pc
store psw,[rx+offsPSW] ; save psw
store sp,[rx+offsSP] ; save sp
; Handle the case:
...
; Select the next running process
call scheduler
; Restore the new context
load rx,running
load sp,[rx+offsSP] ; restore sp
load psw,[rx+offsPSW] ; restore psw
load pc,[rx+offsPC] ; restore pc
load r0,[rx+offsR0] ; restore regs
load r1,[rx+offsR1]
...
load r15,[rx+offsR15]
; Return
iret
Mart 2020.
```

Copyright 2020 by Dragan Milićev

59

## Promena konteksta

- ❖ Šta to znači “sačuvati” i “restaurirati” procesorski kontekst?
  - ❖ čuvanje procesorskog konteksta: prepisati sadržaj registara koji su programski dostupni u nepriviligovanom režimu (instrukcijama procesa) u strukturu koja čini procesorski kontekst tekućeg procesa (*running*)
    - ❖ restauracija procesorskog konteksta: odatle pročitati sačuvane vrednosti i upisati ih u registre
  - ❖ Gde čuvati procesorski kontekst? Svakako negde u operativnoj memoriji, i to tako da to bude pridruženo tekućem procesu
  - ❖ Moguće su i varijante:
    - ❖ u strukturi unutar PCB koja je za to namenjena
    - ❖ na steku procesa, pošto je stek svakako deo konteksta svakog procesa (pa i niti), a samo vrednost *SP* sačuvati u PCB
- ❖ Ovo se radi odgovarajućim instrukcijama prenosa podataka iz registara u memoriju ili obratno - to su instrukcije kernela koje se izvršavaju prilikom promene konteksta
- ❖ Kako konkretno to uraditi? Implementacija može da varira, a svakako zavisi od toga kakvu podršku pruža procesor, odnosno kakva mu je arhitektura

# Promena konteksta

- ♦ Primer 3: procesor više nema rezervisan registar *RX*, mora se koristiti neki od registara opšte namene, dostupnih i u neprivilegovanom režimu (if, dostupnih procesu):

```
; Save the current context
push r0 ; save temporarily r0
load r0,running
store r1,[r0+offsR1] ; save regs
pop r1 ; save r0
store r1,[r0+offsR0]
store r2,[r0+offsR2] ; save r2 etc.
...
store r15,[r0+offsR15]
pop r1 ; save pc
store r1,[r0+offsPC]
pop r1 ; save psw
store r1,[r0+offsPSW]
pop r1 ; save original sp
store r1,[r0+offsSP]

; Handle the case:
...
; Select the next running process
call scheduler

; Restore the new context
load r0,running
load r1,[r0+offsSP] ; restore original sp
push r1
load r1,[r0+offsPSW] ; restore original psw
push r1
load r1,[r0+offsPC] ; restore pc
push r1
load r1,[r0+offsR1] ; restore regs
load r2,[r0+offsR2]

...
load r15,[r0+offsR15]
load r0,[r0+offsR0]
; Return
iret
```

Mart 2020.

Copyright 2020 by Dragan Milićev

61

# Promena konteksta

- ♦ Primer 2, sledeći slučaj: procesor više nema odvojene skupove registara, a pošto se *PC*, *SP* i možda *PSW* odmah implicitno menjaju pri samoj obradi prekida, procesor implicitno čuva njihove vrednosti pri obradi prekida, i to na steku na koji ukazuje poseban sistemski registar, sistemski *SP*:

```
; Save the current context
load rx,running
store r0,[rx+offsR0] ; save regs
store r1,[rx+offsR1]
...
store r15,[rx+offsR15]
pop r0 ; save pc
store r0,[rx+offsPC]
pop r0 ; save psw
store r0,[rx+offsPSW]
pop r0 ; save original sp
store r0,[rx+offsSP]

; Handle the case:
...
; Select the next running process
call scheduler

; Restore the new context
load rx,running
load r0,[rx+offsSP] ; restore original sp
push r0
load r0,[rx+offsPSW] ; restore original psw
push r0
load r0,[rx+offsPC] ; restore pc
push r0
load r0,[rx+offsR0] ; restore regs
load r1,[rx+offsR1]

...
load r15,[rx+offsR15]
; Return
iret
```

Mart 2020.

Copyright 2020 by Dragan Milićev

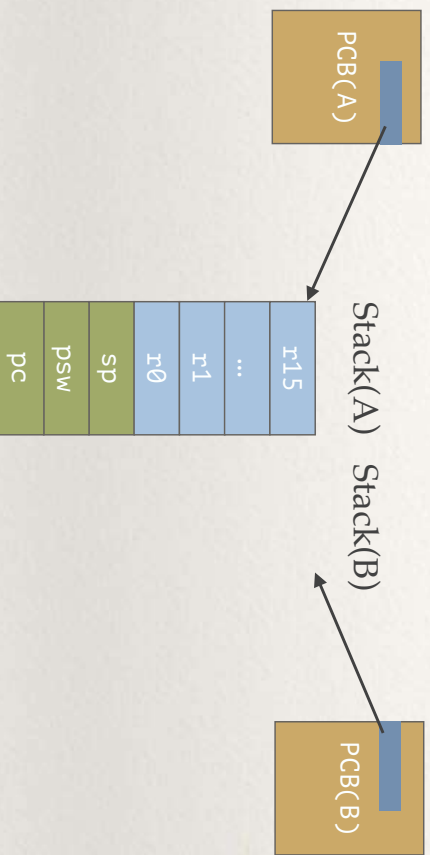
60

# Promena konteksta

- ❖ Primer 4: čuvanje konteksta na steku tekuće niti, na vrh sistemskog steka ukazuje poseban registar procesora, *sistemski SP (SSP)*, koji se koristi samo u privilegovanom režimu i čija se vrednost čuva u PCB procesa:

```
; Save the current context
push r0 ; save regs
push r1
...
push r15
load r0,running
store ssp,[r0+offsSSP] ; save ssp
; Handle the case:
...
; Select the next running process
call scheduler
```

```
; Restore the new context
load r0,running
load ssp,[r0+offsSSP] ; restore ssp
pop r15 ; restore regs
pop r14
...
pop r0
; Return
iret
```



Isti ovaj postupak može da se izvede na potpuno isti način i ako postoji samo jedan stek za proces ili nit, ne mora da postoji sistemski stek i SSP, već se koristi samo SP

Mart 2020.

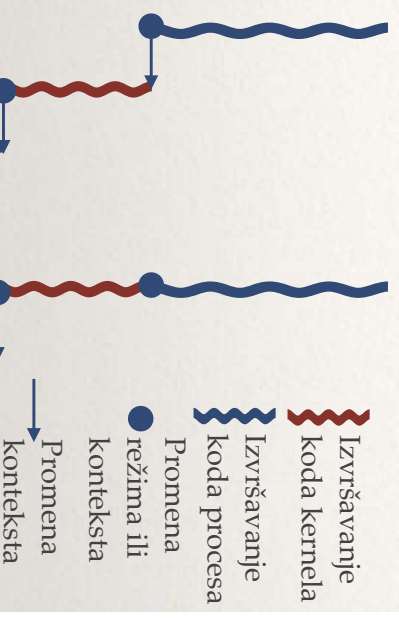
Copyright 2020 by Dragan Milićev

63

# Promena konteksta

- ❖ Do sada je pretpostavljano da sav kernel, odnosno sve instrukcije kernela koje se izvršavaju u ovakvim situacijama, predstavljaju isti tok kontrole i koriste tako isti stek, tzv. kernel stek na koji procesor implicitno prelazi po obradi sistemskog poziva / izuzetka / prekida
- ❖ Jedan od pristupa koji primenjuju sistemi jeste taj da svaki proces ima svoj stek koji se koristi u privilegovanom režimu rada procesora, odnosno pri izvršavanju kernel koda
- ❖ OS za svaki proces, tačnije za svaki tok kontrole (proces ili nit) alokira poseban stek koji se koristi u korisničkom režimu i stek koji se koristi u privilegovanom režimu; kernel kod koji se izvršava na sistemski poziv / izuzetak / prekid se izvršava u istom kontekstu, istog toka kontrole prekinutog procesa, samo u privilegovanom režimu rada procesora, ali na steku tekućeg toka kontrole
- ❖ Na ovaj način zapravo se sve izvršavanje, bez obzira na to da li procesor izvršava instrukcije procesa ili kernela, posmatra kao *višenižno (multithreaded)*, pa je i sam kernel višenižni, konkurentan

Privilegovani režim, stek drugog tekućeg procesa



Mart 2020.

Copyright 2020 by Dragan Milićev

62



# Promena konteksta

- ❖ Razne druge mogućnosti koje može podržati procesor:
  - ❖ posebna instrukcija koja upisuje vrednosti svih registara programski dostupnih u neprivilegovanom režimu na određeno mesto u memoriji, na koje ukazuje neki specijalizovan registar, dostupan samo u privilegovanom režimu; i analogna instrukcija za restauraciju tih registara - umesto izvršavanja više instrukcija, procesor može samo jednom instrukcijom da prepíše registre
  - ❖ procesor ima više registarskih fajlova, od kojih je samo jedan “aktivan”, tj. “tekući”, a posebne instrukcije učitavaju ili upisuju vrednosti registara iz memorije i u nju; kernel može samo da prebaci izvršavanje na drugi registarski fajl, nekom instrukcijom, a da pokrene promenu sadržaja drugog registarskog fajla instrukcijama koje procesor radi “u pozadini”, paralelno sa drugim instrukcijama
  - ❖ i mnoge, mnoge druge varijante

Mart 2020.

Copyright 2020 by Dragan Milićev

65

# Promena konteksta

- ❖ Ovo omogućava da proizvoljne radnje kernel odradi u kontekstu istog, tekućeg procesa ili niti (onog koji se napušta), na njegovom steku, a onda izvrši promenu konteksta na samom kraju obrade situacije, ili u bilo kom drugom trenutku kada se tok kontrole prebacuje na drugi kontekst i stek:

```
inline void yield () {
 asm (
 ; Save the current context
 push r0 ; save regs
 push r1
 ...
 push r15
 load r0,oldRunning
 store ssp,[r0+offSSP] ; save ssp
 ; Restore the new context
 load r0,running
 load ssp,[r0+offSSP] ; restore ssp
 pop r15 ; restore regs
 pop r14
 ...
 pop r0
);
}

interrupt void sys_call () {
 ... // Handle the system call
 oldRunning = running;
 running = Scheduler::get();
 yield();
}
```

Klasa *Scheduler* implementira red spremnih procesa i algoritam raspoređivanja (*scheduling algorithm*), a statička operacija *get* iz reda spremnih uzima proces odabran za izvršavanje

Mart 2020.

Copyright 2020 by Dragan Milićev

64

# Promena konteksta

---

- ❖ Dosadašnji primeri koristili su assembler ciljnog procesora da bi implementirali promenu konteksta
- ❖ Pod određenim uslovima, promena konteksta može se izvršiti i bez ijedne asemblerske intrukcije, time nezavisno od procesora, korišćenjem samo standardne biblioteke jezika C čije su deklaracije u zaglavlju *setjmp.h*:
  - ❖ tip *jmp\_buf* predstavlja strukturu koja sadrži polja za čuvanje vrednosti svih programski dostupnih registara koje dati prevodilac koristi na datom procesoru
  - ❖ *int setjmp (jmp\_buf context)*: funkcija koja čuva kontekst procesora u strukturu datu parametrom (jednostavno prepisuje vrednosti registara u polja strukture) i vraća 0
  - ❖ *void longjmp (jmp\_buf context, int value)*: restaurira kontekst dat kao argument, a koji je sačuvan pomoću *setjmp*; pošto se time skače u sam interni kod funkcije *setjmp*, iz funkcije *longjmp* nema povratka na mesto poziva, već se tok kontrole prebacuje na *setjmp*, iz koga se vraća na isto mesto poziva te funkcije u prethodnom slučaju; da bi se na tom mestu mogao razlikovati slučaj povratka iz "normalnog" poziva, i povratka "niotkuda" skokom sa *longjmp*, u ovom drugom slučaju *setjmp* vraća vrednost parametra *value* funkcije *longjmp* koja zato mora biti različita od 0

Mart 2020.

Copyright 2020 by Dragan Milićev

67

# Promena konteksta

---

- ❖ Prema tome, delimično i u zavisnosti od podrške procesora, konstrukcija kernela može da bude različita u zavisnosti od sledećih projektnih odluka (*design decision*):
  - ❖ da li se sav kod kernela izvršava na samo jednom steku (kernel nije višenitni)
  - ❖ da li se kod kernela, izvršen kao posledica sistemskog poziva / izuzetka / prekida izvršava na (kernel) steku pridruženom tekucem toku kontrole; svaki tok kontrole (proces ili nit) ima svoj stek koji se koristi za izvršavanje kernel koda
  - ❖ da li je sam kernel višenitni; u ovom slučaju, sama obrada zahteva koji je proces postavio može da se obavlja čak i u kontekstu posebnih kernel niti, a ne u istom toku kontrole u kom je izvršen sistemski poziv
- ❖ Odluka o ovome nezavisna je od odluke o tome kada kernel vrši promenu *memorijskog konteksta*, kao što je već opisano: odluku o tome kada i na koji memorijski kontekst kernel treba da prebaci izvršavanje procesora (tj. kada da upiše drugu vrednost u *SMTT/PMTP*), kernel donosi nezavisno od ovoga, već samo u zavisnosti od toga kojim delovima memorije treba da pristupa koji deo koda kernela; u svakom slučaju, kada se vraća kontrola procesu, on mora da ima svoj memorijski kontekst
- ❖ Osim toga, podrazumeva se da se, bez obrira na opisane varijante, instrukcije koje pripadaju kodu procesa izvršavaju u nepriviligovanom, a instrukcije kernela u privilegovanom režimu procesora

Mart 2020.

Copyright 2020 by Dragan Milićev

66

# Promena konteksta

---

- ❖ Ukoliko se ova biblioteka koristi za asinhronu promenu konteksta (na spoljašnje prekide), treba biti obazriv
- ❖ Naime, funkcije *setjmp* i *longjmp* su pravljane tradicionalno za sinhronu pozive, iz izraza u programu na jeziku C; zato je moguće da implementacija ovih funkcija i strukture obezbeđuje čuvanje i restauraciju samo nekih programski dostupnih registara, a ne i onih koje dati prevodilac koristi za čuvanje međuvrednosti tokom izračunavanja izraza, a za koje zna da njihove vrednosti nisu potrebne nakon toga, kada se uopšte može pozvati neka funkcija, pa i *setjmp*
- ❖ Ovo može da predstavlja problem jer se asinhroni prekid može dogoditi u bilo kom trenutku, pa i usred izračunavanja podizraza, kada je vrednost ovakvog privremenog registra bitna; ako ona ne bude sačuvana i kasnije restaurirana, povratak na izračunavanje prekinutog podizraza daće pogrešnu vrednost, jer u datom registru može da se zatekne neka druga vrednost koju je ostavio drugi proces / nit

Mart 2020.

Copyright 2020 by Dragan Milićev

69

# Promena konteksta

---

- ❖ Implementacija sistemskog poziva (statičke funkcije članice) *dispatch* školskog jezgra zato može da izgleda ovako:

```
void Thread::dispatch () {
 lock ();
 if (setjmp(runningThread->context)==0) {
 Scheduler::put(runningThread);
 runningThread = Scheduler::get();
 longjmp(runningThread->context, 1);
 } else {
 unlock ();
 return;
 }
}
```

Mart 2020.

Copyright 2020 by Dragan Milićev

68



# Implementacija niti

- ❖ Kada programski jezik podržava niti, onda se koncept niti može implementirati i unutar *izvišnog okruženja* (*runtime environment*) tog jezika, bez neposredne podrške operativnog sistema
- ❖ Izvišno okruženje može biti:
  - ❖ biblioteka koja obezbeđuje podršku izvršavanju programa na tom jeziku, u skladu sa konceptima tog jezika
  - ❖ virtuelna mašina (interpreter) - softver koji interpretira program na tom jeziku
- Izvišno okruženje svakako koristi usluge operativnog sistema pozivajući sistemске pozive
- ❖ Izvišno okruženje može obezbediti konkurentno izvršavanje niti u programu, a to znači kontekste niti, njihove stekove i promenu konteksta. Pritom OS ništa od toga ne vidi, on vidi samo jedan proces i jedan tok kontrole
- ❖ Upravo u tome i leži osnovni problem ovog pristupa: ako neka od niti programa treba da se suspenduje, da nešto čeka, recimo ulazno-izlaznu operaciju koju mora da obezbedi OS, operativni sistem mora da suspenduje jedini tok kontrole koji zapravo i vidi, a to znači da ceo taj program i sve njegove niti postaju suspendovane, jer taj proces više ne dobija procesor
- ❖ Zato je efikasnije kada OS podržava koncept niti, a izvišno okruženje onda može da preslikava niti iz programa u niti operativnog sistema, i to na različite načine:
  - ❖ jedan u jedan: svaka nit u programu implementira se jednom niti operativnog sistema
  - ❖ više u jedan: više niti u programu implementira se jednom niti operativnog sistema, a promenu konteksta između njih obavlja izvišno okruženje
  - ❖ više u više

Mart 2020.

Copyright 2020 by Dragan Milićev

71

## Promena konteksta

- ❖ Promena konteksta je čist režijski trošak, jer procesor ne izvršava korisne instrukcije procesa, ali koliko zapravo traje?
- ❖ To zavisi od sledećih faktora:
  - ❖ arhitekture procesora: koje registre ima dostupne u neprivilegovanom režimu rada, odnosno šta treba sačuvati i restaurirati, koju podršku ima za promenu konteksta (koje registre čuva implicitno, sam, a koje instrukcije ili druge mehanizme ima za čuvanje odnosno restauraciju ostalih registara)
  - ❖ konstrukcije samog operativnog sistema, načina organizacije struktura podataka za predstavljanje konteksta procesa i operacija koje treba da uradi
- ❖ U praksi, trajanje promene konteksta može biti od reda 1 mikrosekunde ili manje, pa do 1 milisekunde

Mart 2020.

Copyright 2020 by Dragan Milićev

70

# Raspoređivanje

---

- ❖ Kada vrši promenu konteksta, OS treba da izabere neki proces (ili nit, zapravo tok kontrole) iz skupa spremnih procesa kom će dodeliti procesor. Koji?
- ❖ Izbor procesa za izvršavanje iz skupa spremnih nikako ne utiče na samu semantiku (logiku) izvršavanja tih procesa: pošto su svi oni spremni, imaju zadovoljene uslove nastavka izvršavanja, bilo koji od njih može dobiti procesor
- ❖ Zato se ovaj izbor može apstrahovati, a njegova implementacija lokalizovati i enkapsulirati u poseban deo kernela koji je za to odgovoran, a ostatak sistema od toga ne zavisi:  
`running = Scheduler::get();`
- ❖ Međutim, izbor procesa za izvršavanje itekako može da utiče na *vremensko ponašanje* sistema, odnosno vreme odziva i performanse celog sistema
- ❖ Zato je *algoritam raspoređivanja* procesa na procesoru (*process scheduling, processor scheduling*) izuzetno bitan element svakog kernela. Razvojem operativnih sistema tokom istorije ovi algoritmi su stalno unapređivani, u skladu sa iskustvima i novim idejama, a i dalje se unapređuju

Mart 2020.

Copyright 2020 by Dragan Milićev

73

# Implementacija niti

---

- ❖ Implementacija promene konteksta niti unutar OS nema nikakve suštinske razlike od implementacije promene konteksta procesa: jedino pitanje jeste to da li se prilikom promene procesorskog, menja i memorijski kontekst toka kontrole
- ❖ Međutim, kako se to radi upisom odgovarajuće vrednosti u *PMTP / SMTP*, vrednost tog registra može biti deo konteksta svakog toka kontrole (procesa ili niti) i uvek upisivati, pri svakoj promeni konteksta, tako da je potpuno svedjedno da li se radi o drugom procesu ili niti
- ❖ Zato neki operativni sistemi u najvećoj meri ne prave razliku između procesa i niti, odnosno generalizuju ta dva koncepta; na primer, Linux ih naziva jednim nazivom *zadatak (task)*
- ❖ Naravno, samo kreiranje procesa razlikuje se od kreiranja niti, jer se tom prilikom kreira nov memorijski kontekst i alociraju drugi resursi operativnog sistema koji su zajednički za sve niti (npr. standardni ulazno-izlazni uređaji)
- ❖ Zbog toga se trajanje kreiranja procesa može mnogo razlikovati od trajanja kreiranja niti, i do nekoliko desetina puta može biti duže
- ❖ Naravno, kada ne treba praviti fizičke kopije stranica memorije, kao što je slučaj kod sistemskog poziva *fork*, koristi se tehnika kopiranja na upis (*copy on write*) koja značajno smanjuje vreme kreiranja početnog memorijskog konteksta

Mart 2020.

Copyright 2020 by Dragan Milićev

72



# Raspoređivanje

- ❖ Očigledan je problem ovog algoritma: procesi koji se veoma kratko izvršavaju mogu dugo da čekaju ukoliko su ispred njih u redu procesi koji se vrlo dugo izvršavaju
- ❖ Ukoliko bi se procesi sa kratkim izvršavanjem pustili “preko reda”, dugotrajni procesi ne bi trpeli veliku štetu, a kratkotrajni procesi bi značajno skratili svoje vreme čekanja (primer reda čekanja na kasi u samoposluzi - ljudi sa samo jednim ili dva artikla ispred kojih su ljudi sa punim kolicima)
- ❖ FIFO algoritam ima još jedan problem, tzv. *konvoji efekat* (*convoy effect*):
  - ❖ kratkotrajni proces je bio u redu spremnih iza nekog dugotrajnog procesa
  - ❖ ovaj prvi završi svoj nalet izvršavanja i ode iz reda spremnih, recimo zato što je zahtevao neku ulazno-izlaznu operaciju sa diskom
  - ❖ kratkotrajni proces dobija procesor, ali brzo završava svoj nalet izvršavanja, jer i on traži operaciju sa diskom
  - ❖ u međuvremenu je onaj prvi proces započeo operaciju sa diskom, pa ovaj drugi mora da čeka da on završi, kako bi obavio svoju operaciju sa diskom
  - ❖ pošto prvi proces prvi i završava operaciju sa diskom, opet dolazi u red spremnih i opet ispred ovog drugog procesa, pa se situacija ponavlja
  - ❖ zbog toga neki “spori” procesi (poput sporih kamiona i šlepera na uskom putu) “vuku iza sebe kolone brzih” procesa, koji nikako ne mogu da ih preteknu, pa se negativan efekat samo povećava

Mart 2020.

Copyright 2020 by Dragan Milićev

75

# Raspoređivanje

- ❖ Najjednostavniji algoritam jeste opsluživanje po *redosledu dolaska* (*first come - first served*, FCFS, ili *first in - first out*, FIFO): procesor dobija onaj proces koji je najdavnije došao u red spremnih
- ❖ Implementacija mu je jednostavna, pomoću FIFO reda
- ❖ Međutim, ovaj algoritam ima ozbiljne nedostatke. Prvi je taj što vreme čekanja procesa na procesor, a time i vreme ukupnog izvršavanja ili odziva na akciju, može da bude nerazumno veliko
- ❖ Na primer, neka su u redu spremnih, tim redom, sledeći procesi sa datim vremenima trajanja sledećeg naleta izvršavanja na procesoru (*CPU burst*): ako se oni izvršavaju po redosledu dolaska (FIFO), vreme čekanja svakog procesa na procesor dato je u tabeli
- ❖ Ukupno vreme čekanja za ove procese je 105, odnosno u proseku 26,25

| Proces               | Trajanje izvršavanja | Vreme početka izvršavanja | Vreme završetka izvršavanja |
|----------------------|----------------------|---------------------------|-----------------------------|
| <i>P1</i>            | 18                   | 0                         | 18                          |
| <i>P2</i>            | 24                   | 18                        | 42                          |
| <i>P3</i>            | 3                    | 42                        | 45                          |
| <i>P4</i>            | 6                    | 45                        | 51                          |
| Ukupno vreme čekanja |                      | 105                       |                             |

Mart 2020.

Copyright 2020 by Dragan Milićev

74



# Raspoređivanje

- ❖ U današnjim sistemima upotrebljavaju se mnogi sofisticiraniji algoritmi, od kojih jednu značajnu grupu čine algoritmi zasnovani na *prioritetima* (*priority*):
  - ❖ svakom procesu dodeli se prioritet
  - ❖ za izvršavanje se bira onaj proces iz skupa spremnih koji ima najviši prioritet
  - ❖ OS može dinamički i menjati prioritet procesa, u zavisnosti od ponašanja procesa, npr. od toga da li je interaktivan (pa zahteva brži odziv) i drugo
- ❖ U sistemima sa raspodelom vremena koristi se tzv. *round robin* algoritam: kružno opsluživanje spremnih procesa, isto kao FIFO, ali sa ograničenim vremenskim odsečkom, kako je to ranije već opisano
- ❖ Više detalja u predmetu OS2

Mart 2020.

Copyright 2020 by Dragan Milićev

# Raspoređivanje

- ❖ Zbog ovoga se FIFO algoritam više nikada ne upotrebljava (osim u najranijim multiprocesnim sistemima), već su osmišljeni brojni drugi algoritmi
- ❖ Jedan, teorijski optimalan, ali praktično neprimenjiv u egzaktnoj formi, jeste algoritam koji se intuitivno nameće iz prethodnog razmatranja: algoritam *najkraći posao prvi* (*shortest job first*, SJF); za prethodni primer ista četiri procesa, izvršavanje bi išlo ovako, a ukupno (39) i prosečno (9,75) vreme čekanja znatno kraće

| Proces               | Trajanje izvršavanja | Vreme početka izvršavanja | Vreme završetka izvršavanja |
|----------------------|----------------------|---------------------------|-----------------------------|
| $P_3$                | 3                    | 0                         | 3                           |
| $P_4$                | 6                    | 3                         | 9                           |
| $P_1$                | 18                   | 9                         | 27                          |
| $P_2$                | 24                   | 27                        | 51                          |
| Ukupno vreme čekanja |                      | 39                        |                             |

Mart 2020.

Copyright 2020 by Dragan Milićev

# Kooperativni procesi

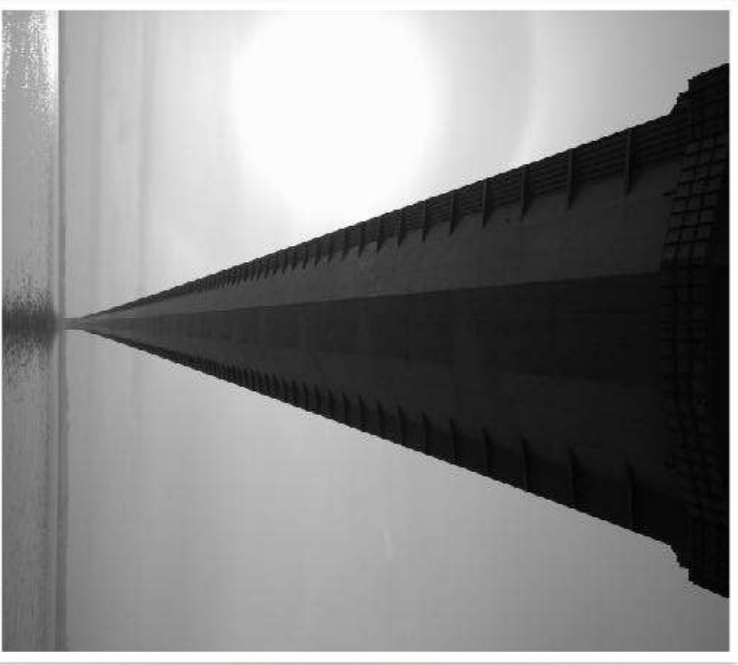
- ❖ Koncept procesa je u osnovi zamišljen kao način da se obezbedi nezavisno, izolovano izvršavanje programa na računaru, na način na koji on ima isto ponašanje kao i da je sam, iako se izvršava uporedo sa drugim procesima
- ❖ U principu, veliki broj programa je i napravljen tako da se izvršavaju kao *logički nezavisni procesi*, odnosno tako da ni na koji način ne interaguju sa drugim procesima i sa njima ne razmenjuju nikakve informacije; svako na svom računaru izvršava puno takvih logički nezavisnih procesa
- ❖ Međutim, u mnogim prilikama postoji potreba da procesi *interaguju*, na primer tako što će međusobno razmenjivati informacije; ovakvi procesi nazivaju se *kooperativnim procesima* (*cooperating processes*)
- ❖ Primer: mehanizam “cevovoda” (*pipe*) kojim se izlaz jednog procesa preusmerava na ulaz drugog procesa:  
`cat listing.txt | less`
- ❖ Drugi primer su niti koje dele adresni prostor: takvi uporedni tokovi kontrole se i prave kao niti upravo zato da bi pristupale zajedničkim, deljenim strukturama podataka u istom adresnom prostoru
- ❖ Često se procesi namerno prave tako da razmenjuju informacije sa drugim procesima, na primer preko fajla ili deljene memorije (podsetiti se mehanizma logičkog deljenja virtuelne memorije):
  - ❖ Zašto se prave kao nesavisni programi? Zbog modularnosti: njihovo funkcionisanje je u najvećem delu nezavisno i lakše ih je održavati (ili su nezavisno i napravljeni) kada su odvojeni
  - ❖ Zašto se izvršavaju kao uporedni procesi? Zbog efikasnosti (multiprogramiranje, multiprocesiranje)

Sistemski program *cat* na svoj izlaz izbacuje znakove pročitane iz fajla koji je zadat kao argument. Sistemski

Mart 20 protiran *less* omogućava prikaz znakova sa svog ulaza uz kontrolu prikaza po stranicama od strane korisnika

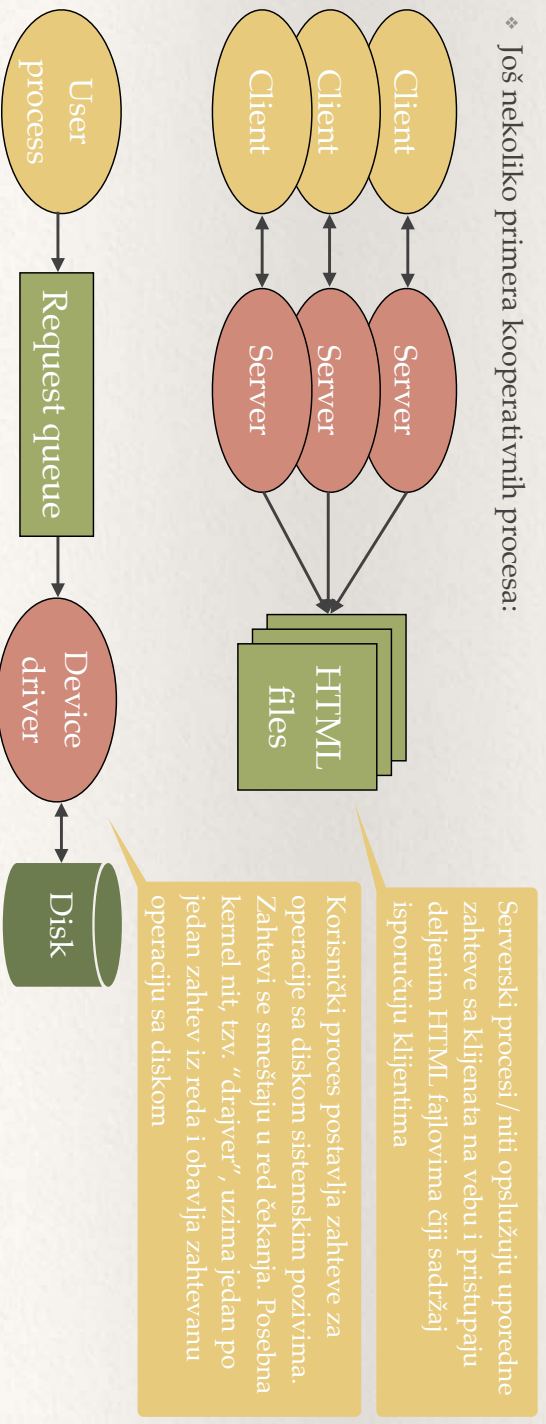
## Glava 9: Sinhronizacija procesa

- ❖ Kooperativni procesi
- ❖ Sinhronizacija procesa
- ❖ Uposleno čekanje
- ❖ Podrška hardvera
- ❖ Semafori
- ❖ Druge tehnike sinhronizacije
- ❖ Modeli međuprocen  
ne komunikacije



# Kooperativni procesi

- ❖ Problemi koji nastaju zbog konkurentnosti (konflikti) posledica su interakcije između uporednih tokova kontrole. Kako i procesi (ne samo niti) mogu pristupati deljenoj memoriji ili drugim deljenim podacima (npr. fajlovima), problemi konkurentnosti o kojima će ovde biti reči zapravo su potpuno zajednički i za procese i niti
- ❖ Zato će se, nadalje, osim ako se u datom kontekstu ne smatra drugačije, termin *proces* odnositi na tok kontrole, bilo (teški) proces ili nit, svejedno
- ❖ Još nekoliko primera kooperativnih procesa:



Mart 2020.

Copyright 2020 by Dragan Milićev

81

# Kooperativni procesi

- ❖ Međutim, čak i kada su procesi pravljeni tako da budu logički nezavisni od drugih, kada se izvršavaju na istom računaru, odnosno na istom operativnom sistemu (kernelu), oni svakako implicitno interaguju, iako toga "nisu ni svesni", jer procesi:
  - ❖ pozivaju sistemske pozive istog kernela, ulaze u iste delove koda koji pristupa istim strukturama podataka kernela u memoriji,
  - ❖ zahtevaju operacije sa istim fizičkim uređajima,
  - ❖ konkuriraju za isti procesor i istu operativnu memoriju,
  - ❖ pristupaju istom fajl sistemu
- ❖ Prema tome, na jednom računaru potpuno nezavisnih procesa zapravo i nema!
- ❖ Potpuno nezavisni procesi su tako samo oni koji su logički nezavisni (izvršavaju programe koji nemaju interakciju sa drugima) i izvršavaju se na različitim računarima
- ❖ Zašto interakcija procesa čini stvari komplikovanijim? Jer su mogući *konflikti* (*conflicts*), kao i u realnom životu: kada bi svaki automobil imao samo svoj put, konflikata (uđesa) praktično ne bi ni bilo
- ❖ A zašto se procesi izvršavaju na istom računaru? Naravno, zbog efikasnosti i praktičnosti, tj. upravo zbog deljenja resursa

Mart 2020.

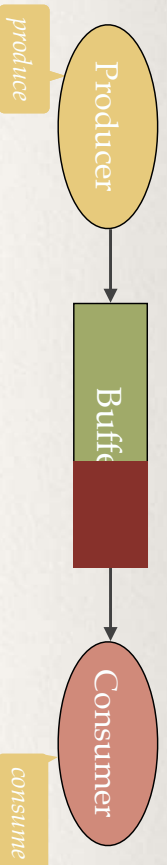
Copyright 2020 by Dragan Milićev

80



# Kooperativni procesi

- ❖ Rešenje za ovaj problem je uvođenje *bafera* (*buffer*): bafer je nekakav memorijski element, deljena struktura podataka, fajl ili slično, u koji proizvođač može da upisuje svoje proizvode, a potrošač da iz njega čita; sada:
  - ❖ proizvođač ne mora da čeka da potrošač potroši prethodni proizvod, već ga prosto smešta u bafer (sve dok u baferu ima mesta)
  - ❖ potrošač ne mora da čeka da proizvede naredni proizvod, već ga prosto uzima iz bafera (sve dok u baferu ima proizvoda)
- ❖ Na ovaj način su proizvođač i potrošač raspregnuti (dekuplovani), a postojanje bafera omogućava povremene razlike u brzinama proizvodnje i potrošnje (*buffer* je reč koja označava odstojnik, amortizer između vagona koji služi da amortizuje udarce koji nastaju zbog povremenih malih razlika u brzinama kretanja susednih vagona)



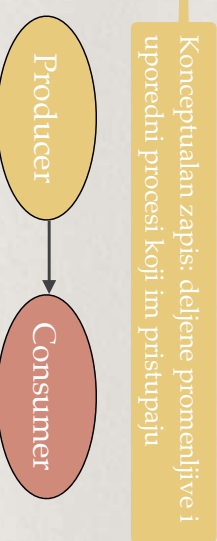
Mart 2020.

Copyright 2020 by Dragan Milicev

83

## Kooperativni procesi

- ❖ Jedan veoma čest obrazac, model saradnje između uporednih procesa jeste tzv. model *proizvođač-potrošač* (*producer-consumer*):
  - ❖ jedan proces ili više njih *proizvode* nekakve informacije, podatke, pakete ili poruke koje treba da prolede
  - ❖ jedan proces ili više njih *konzumiraju* (troše) te informacije, podatke, pakete ili poruke koje je proizveo proizvođač
- ❖ Jedan pristup za koordinaciju jeste taj da se ovi procesi neposredno spregnu:
  - shared var ready : boolean:=false;**  
**shared var item : Data;**  
**process producer;**  
**loop**  
**while ready do null;**  
**produce(item);**  
**ready := true;**  
**end loop;**  
**process consumer;**  
**loop**  
**while not ready do null;**  
**consume(item);**  
**ready := false;**  
**end loop;**



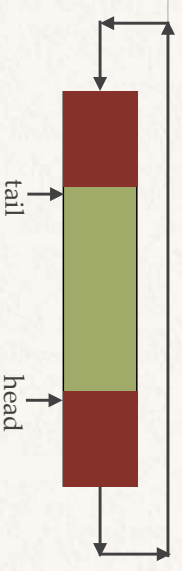
- ❖ Problem: proizvođač i potrošač su čvrsto spregnuti i nametnuta im je naizmeničnost u operacijama: proizvođač ne može da proizvede nov proizvod ako prethodni nije potrošen i obratno, potrošač ne može da konzumira naredni proizvod ako ga potrošač nije proizveo; zato njihovo napredovanje zavisi od onog drugog, što nije dobro ako oni u pojedinim intervalima imaju različite brzine proizvodnje i potrošnje

Mart 2020.

Copyright 2020 by Dragan Milicev

82

# Kooperativni procesi



- ❖ Ograničeni bafer - skica implementacije - nastavak:

```
BoundedBuffer::BoundedBuffer () :
 head(0), tail(0), count(0) {}
```

```
void BoundedBuffer::append (Data* d) {
```

```
 while (count==N);
 buffer[tail] = d;
 tail = (tail+1)%N;
 count++;
}
```

```
Data* BoundedBuffer::take () {
```

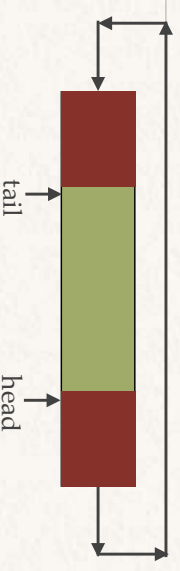
```
 while (count==0);
 Data* d = buffer[head];
 head = (head+1)%N;
 count--;
 return d;
}
```

Mart 2020.

Copyright 2020 by Dragan Milićev

85

# Kooperativni procesi



- ❖ Bafer je neakav memorijski element, struktura podataka koja može biti logički neograničenog kapaciteta (dinamički se proširuje), ali često i ograničenog kapaciteta; čak i kad je logički neograničen, uvek postoji neko fizičko ograničenje
- ❖ Zato se često govori o modelu saradnje proizvođača i potrošača preko *ograničenog bufera* (*bounded buffer*)
- ❖ Primer: skica jednostavne implementacije ograničenog bafera pomoću niza elemenata ograničenog kapaciteta, kao kružnog bafera

```
const int N = ...;
class Data;

class BoundedBuffer {
public:
 BoundedBuffer ();

 void append (Data*);
 Data* take ();

private:
 Data* buffer[N];
 int head, tail, count;
};
```

Mart 2020.

Copyright 2020 by Dragan Milićev

84

# Sinhronizacija procesa

- ❖ Zbog toga je osnovni postulat konkurentnog programiranja taj da se ništa ne pretpostavlja o ovim elementima platforme izvršavanja, odnosno da se pretpostavlja najopštiji slučaj, a on podrazumeva da se sekvence instrukcija uporednih procesa mogu preplitati na bilo koji način
- ❖ Pojam *uporednosti* ili *konkurentnosti* (*concurrency*) upravo apstrahuje ovakvo, najopštije izvršavanje, kao i *potencijalni paralelizam* tog izvršavanja
- ❖ *Paralelno* izvršavanje ili *paralelizam* (*parallelism*) podrazumeva fizički istovremeno izvršavanje, što je moguće samo na više procesora. *Konkurentnost* podrazumeva mogućnost takvog paralelnog izvršavanja, ukoliko za to postoje mogućnosti, a u svakom slučaju podrazumeva i multiprogramiranje sa nepredvidivim načinom preplitanja izvršavanja instrukcija uporednih procesa
- ❖ Logička ispravnost rezultata izvršavanja potpuno nezavisnih procesa ne zavisi od načina preplitanja njihovih instrukcija. Međutim, za kooperativne procese to ne važi: rezultat može zavisi od redosleda tog preplitanja, pa zato može biti nepredvidiv
- ❖ Osim u izuzetno retkim slučajevima kada se želi baš takvo ponašanje softvera, nepredvidivo ponašanje je nepoželjno; upravo zbog takvog ponašanja konstrukcija, razumevanje, održavanje i testiranje konkurentnih programa postaje znatno teže nego kod sekvencijalnih programa, a eventualne greške su teško ponovljive i zato se teško otkrivaju (jer se mogu manifestovati samo u određenim, retkim situacijama nezgodnog preplitanja)

Mart 2020.

Copyright 2020 by Dragan Milićev

87

# Sinhronizacija procesa

- ❖ Podrazumevano se procesi izvršavaju *uporedo* (*concurrently*), što znači da se sekvence njihovih instrukcija izvršavaju proizvoljno prepleteno ili čak fizički paralelno
- ❖ Konkretan fizički, vremenski raspored i redosled izvršavanja tih instrukcija, osim od samih njihovih tokova kontrole koji diktiraju njihov redosled, zavisi od sledećih elemenata *platforme* na kojoj se izvršavaju:
  - ❖ da li se procesi izvršavaju na jednom procesoru, multiprogramiranjem (tj. vremenskim multipleksiranjem, *multiprogramming*) ili na više procesora, multiprocesiranjem (fizički paralelno, istovremeno, *multiprocessing*)
  - ❖ ako se neki procesi izvršavaju na jednom procesoru, multiprogramiranjem, pitanje je u kojim se tačno trenucima dešava promena konteksta; ako se ona dešava samo sinhrono, onda se ti trenuci mogu predvideti; ako se ona izvršava i asinhrono (na spoljašnje prekide), što je opštiji i realan slučaj, onda se ti trenuci ne mogu predvideti
  - ❖ kada se i dogodi promena konteksta, pitanje je koji proces će biti izabran za izvršavanje, što zavisi od algoritma raspoređivanja; čak i ako je taj algoritam poznat, teško je u svakom trenutku odrediti kako izgleda skup spremnih procesa
- ❖ Neki od ovih elemenata (npr. trenuci pojave prekida) su nepredvidivi; ostali elementi, čak i ako su teorijski predvidivi, mogu se predvideti samo u krajnje trivijalnim situacijama; iole realnije situacije su toliko složene da je predviđanje potpuno neizvodljivo

Mart 2020.

Copyright 2020 by Dragan Milićev

86



# Sinhronizacija procesa

- ❖ Primer od ranije - uporedno sabiranje dimenzija 3D vektora u tri uporedne niti:

```
#include <pthread>

typedef double Vector3D[3];

struct VectorDimAdderParams {
 const Vector3D* vect; Vector3D* res; int size, dim;
};

void vectorDimAdder (void* params) {
 VectorDimAdderParams* p = (VectorDimAdderParams*)params;
 (*p->res)[p->dim] = 0.0;
 for (int i=0; i<p->size; i++)
 (*p->res)[p->dim] += p->vect[i, p->dim];
}

void matrixAdder (const Vector3D vect[], Vector3D res, int size) {
 VectorDimAdderParams args0 = {vect, &res, size, 0}; pthread_t adder0;
 pthread_create(&adder0, nullptr, &vectorDimAdder, &args0);

 VectorDimAdderParams args1 = {vect, &res, size, 1}; pthread_t adder1;
 pthread_create(&adder1, nullptr, &vectorDimAdder, &args1);

 VectorDimAdderParams args2 = {vect, &res, size, 2}; pthread_t adder2;
 pthread_create(&adder2, nullptr, &vectorDimAdder, &args2);

 pthread_join(&adder0, NULL);
 pthread_join(&adder1, NULL);
 pthread_join(&adder2, NULL);
 ... // Do something with res
}
```

Uslovna sinhronizacija: pre nego što niti deca završe, rezultat nije spreman u *res*, tek kada niti deca završe, može se koristiti vrednost rezultata u *res*. U suprotnom, bez ove sinhronizacije, vrednost pročitana iz *res* bila bi potpuno nedefinisana i nepredvidiva

Mart 2020.

Copyright 2020 by Dragan Milićev

89

## Sinhronizacija procesa

- ❖ Da bi se rešili ovakvi problemi, potrebno je u određenim situacijama *ograničiti* neodređenost u načinu preplitanja instrukcija uporednih procesa
- ❖ Uvođenje ograničenja u pogledu načina preplitanja akcija uporednih procesa ili načina njihovog napredovanja, koja onda izvršno okruženje ili OS moraju zadovoljiti tokom uporednog izvršavanja procesa, naziva se *sinhronizacija* (*synchronization*)
- ❖ Jedan tip sinhronizacije predstavlja *uslovna sinhronizacija* (*conditional synchronization*): neki proces ne sme da nastavi izvršavanje iza neke tačke, tj. ne sme da izvršava neke akcije ukoliko neki drugi proces nije uradio nešto, ili ukoliko nije ispunjen neki uslov, ili ukoliko neki proces ili podatak nije u nekom potrebnom stanju i slično
- ❖ Jedan jednostavan primer uslovne sinhronizacije procesa već smo videli: proces ili nit roditelj kreira procese ili niti decu, koja uporedo obavljaju neke operacije, a roditelj čeka da se oni (ili neko od njih) završi i eventualno isporuči rezultat:
  - ❖ sistemski pozivi tipa *wait* za procese
  - ❖ sistemski pozivi tipa *join* za niti

Mart 2020.

Copyright 2020 by Dragan Milićev

88

# Sinhronizacija procesa

- ❖ Postoji još jedna karakteristična vrsta sinhronizacije, odnosno situacije za koju je potrebna sinhronizacija
- ❖ Pretpostavimo da dva uporedna procesa pristupaju nekoj deljenoj promenljivoj ili strukturi podataka:
  - ❖ ako su u pitanju procesi, mogu da pristupaju promenljivoj u deljenoj memorijskoj oblasti
  - ❖ ako su u pitanju niti, mogu da pristupaju statičkoj promenljivoj u zajedničkom adresnom prostoru
- ❖ Neka je deljena promenljiva  $x$  jednostavnog celobrojnog tipa i neka joj je inicijalna vrednost 0, a oba procesa treba da izvrše sledeću jednostavnu operaciju inkrementiranja te promenljive:  
$$x := x + 1$$
- ❖ Na picoRISC procesoru, ali i na mnogim drugim, može se pretpostaviti da će ova naredba biti prevedena u sekvencu instrukcija poput ovakve:  

```
load r0, x
inc r0
store r0, x
```
- ❖ Može se možda pretpostaviti da procesor izvršava svaku instrukciju atomično (ne prihvata prekide dok se ona ne završi), odnosno da se operacije upisa i čitanja u skalarnu promenljivu u memoriji obavljaju jednim, atomičnim ciklusom na magistrali, pa će više procesora izvršavati instrukcije *load* i *store* nedeljivo, bez preplitanja, ali se to ne može pretpostaviti za celu ovu grupu instrukcija

Mart 2020.

Copyright 2020 by Dragan Milićev

91

## Sinhronizacija procesa

- ❖ Drugi primer - uslovna sinhronizacija kod ograničenog bafera:

- ❖ proizvođač ne sme da krene u operaciju smeštanja elementa u bafer ako je bafer pun

```
void BoundedBuffer::append (Data* d) {
 while (count==N);
 buffer[tail] = d;
 tail = (tail+1)%N;
 count++;
}
```

Uslovna sinhronizacija - čekanje

Uslovna sinhronizacija - signal

- ❖ potrošač ne sme da krene u operaciju uzimanja elementa iz bafera ako je bafer prazan

```
Data* BoundedBuffer::take () {
 while (count==0);
 Data* d = buffer[head];
 head = (head+1)%N;
 count--;
 return d;
}
```

Uslovna sinhronizacija - čekanje

Uslovna sinhronizacija - signal

- ❖ Opšti slučaj: jedan proces *čeka* na ispunjenje uslova da bi nastavio izvršavanje od neke tačke, a drugi proces *signalizira* ispunjenje tog uslova u odgovarajućem trenutku

Mart 2020.

Copyright 2020 by Dragan Milićev

90

# Sinhronizacija procesa

- ❖ Drugi primer: jedan proces računa koordinate pozicije na koju treba pomeriti neki kursor na ekranu, a drugi proces čita te koordinate i pozicionira kursor na tu poziciju:

```
type Coord = record x, y : integer end;
shared var coord : Coord := {0,0};
```

```
process writer;
var nextCoord : Coord;
loop
 compute(nextCoord);
 coord := nextCoord;
end loop;

process reader;
var nextCoord : Coord;
loop
 nextCoord := coord;
 moveTo(nextCoord);
end loop;
```

- ❖ Čak i ako se pretpostavi da je operacija čitanja ili upisa u jednu skalarnu promenljivu (x ili y) atomična, čitanje ili upis u strukturu dve takve promenljive vrlo verovatno nije. Zato se može dogoditi sledeći scenario:

| Proces writer upisuje u coord | Vrednost coord | Proces reader čita iz coord |
|-------------------------------|----------------|-----------------------------|
| x:=1                          | 1, 0           |                             |
| y:=1                          | 1, 1           |                             |
|                               |                | x=1                         |
|                               |                | y=1                         |
| x:=2                          | 2, 1           |                             |
|                               |                | x=2                         |
|                               |                | y=1                         |
| y:=2                          | 2, 2           |                             |

Mart 2020.

Copyright 2020 by Dragan Milićev

Dogodilo se preplitanje, pa je reader pročitao nekorektno (2, 1) pre nego što je writer završio upis i u y

93

## Sinhronizacija procesa

- ❖ Zato su mogućí sledeći scenariji preplitanja ovih instrukcija dva uporedna procesa. Prvi slučaj, bez ikakvog preplitanja:

```
Process 1:
load r0, x ; r0:=0
inc r0 ; r0:=1
store r0, x ; x:=1
```

```
Process 2:

load r0, x ; r0:=1
inc r0 ; r0:=2
store r0, x ; x:=2
```

- ❖ U ovom slučaju tzv. *serializovanog izvršavanja* (*serialized execution*), kada se ove sekvence instrukcija ne prepliću, rezultat - vrednost promenljive x biće 2, kao kada se ovi procesi izvršavaju sekvencijalno i nezavisno, jedan, pa drugi

- ❖ U drugom slučaju, zbog asinhronne promene konteksta ili paralelnog izvršavanja na više procesora, dešava se preplitanje:

```
Process 1:
load r0, x ; r0:=0
inc r0 ; r0:=1
store r0, x ; x:=1
```

```
Process 2:

load r0, x ; r0:=0
inc r0 ; r0:=1
store r0, x ; x:=1
```

- ❖ Rezultat - vrednost promenljive x u prvom slučaju biće 2, a u drugom 1, i ta vrednost je nepredvidiva, jer zavisi od nepredvidivih faktora koji su ranije opisani (elemenata platforme izvršavanja)
- ❖ Logička ispravnost korektnog konkurentnog programa ne sme da zavisi od ovakvih nepredvidivih faktora, jer je takav rezultat nepredvidiv

Mart 2020.

Copyright 2020 by Dragan Milićev

92



# Sinhronizacija procesa

- ❖ Prema tome, postoje sekvence instrukcija, odnosno sekcije koda uporednih procesa koje treba izvršavati *neделиivo* (*indivisibly*), *atomično* (*atomically*), ili, kako se najispravnije kaže, *izolovano* (*isolated*), tako da njihov efekat bude takav kao da drugih procesa nema, odnosno kao da tokom njihovog izvršavanja nema interakcije sa drugim procesima i njihovog uticaja
- ❖ Ovo su tipično sekcije koda koje obavljaju (složene, neatomične) operacije čitanja i /ili upisa u deljene promenljive, kao u pokazanim primerima
- ❖ Ovakve sekcije koda uporednih procesa nazivaju se *kritične sekcije* (*critical section*)
- ❖ Očigledno je potrebna sinhronizacija koja obezbeđuje izolaciju, odnosno isključenje uporednog izvršavanja kritičnih sekcija: ako jedan proces uđe u kritičnu sekciju, drugi procesi ne smeju da budu u svojim kritičnim sekcijama koje su sa tom u potencijalnom konfliktu, niti da uđu u njih
- ❖ Ovakva sinhronizacija naziva se *međusobno isključenje* (*mutual exclusion*)
- ❖ Ove pojmove uveo je čuveni Edsger W. Dijkstra 1965. godine u prvom, antologijskom radu koji se bavi konkurentnim programiranjem ("Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, 8 (9))
- ❖ Konflikt koji nastaje ukoliko nije obezbeđena potrebna sinhronizacija (međusobno isključenje ili uslovna sinhronizacija) naziva se *utrkivanje* (*race condition*): proces se "zatrčao", pretekao je neki drugi proces, pa je krenuo da radi operacije koje ne bi trebalo da radi, jer nisu ispunjeni uslovi za to

Mart 2020.

Copyright 2020 by Dragan Milićev

95

## Sinhronizacija procesa

- ❖ Treći primer: operacije *append* i *take* ograničenog bafera inkrementiraju, odnosno dekrementiraju promenljivu *count*, ako više proizvođača, odnosno potrošača, uporedo izvršava te funkcije, može se dogoditi konflikt poput onog u prvom primeru

```
void BoundedBuffer::append (Data* d) {
 while (count==N);
 buffer[tail] = d;
 tail = (tail+1)%N;
 count++;
}
```

- ❖ Četvrti primer: višenitni kernel, dešava se sledeći scenario:

- ❖ procesor uđe u izvršavanje kernel koda za promenu konteksta, sačuva kontekst tekućeg procesa
- ❖ procesor pređe na izvršavanje koda kernela koji premešta PCB tekućeg procesa u neki red (spremnih, suspendovanih), ali uradi samo delimičnu promenu strukture (ovo prelancavanje po pravilu zahteva više elementarnih operacija, npr. promenu nekoliko pokazivača u listama)
- ❖ pre nego što završi celu promenu, dogodi se prekid, procesor pređe u prekidnu rutinu - kod kernela, koji krene da pristupa istim ovim strukturama podataka koje su u nekonzistentnom stanju, jer prethodno započeta transformacija nije završena!
- ❖ Isti ovaj scenario može se dogoditi i tako što jedan procesor krene u promenu neke strukture kernela u memoriji (npr. red spremnih procesa), a onda i drugi procesor paralelno krene da radi to isto i napravi konflikt poput onih već opisanih

Mart 2020.

Copyright 2020 by Dragan Milićev

94

# Uposleno čekanje

- ❖ Ovakav pristup, u kom proces čeka na ispunjenje uslova izvršavajući petlju u kojoj stalno iznova izračunava vrednost uslova i ispituje tu vrednost, i vrti se u praznoj petlji dok uslov ne bude ispunjen, naziva se *uposlano čekanje* (*busy waiting*)
  - ❖ Očigledan nedostatak je u tome što se proces izvršava, tj. troši procesorsko vreme na izvršavanje "jalovih" instrukcija koje stalno ispituju uslov koji može biti ispunjen mnogo kasnije
  - ❖ Uslovna sinhronizacija se jednostavno rešava uposlenim čekanjem, ali šta je sa međusobnim isključenjem?
  - ❖ Opšti, generički oblik izvršavanja kooperativnih procesa: proces u svom izvršavanju naizmenično i ciklično prolazi kroz sledeće dve faze:
    - ❖ vrši neko svoje privatno izračunavanje, pristupajući samo svojim privatnim podacima (nedeljenim), bez ikakve interakcije sa drugim procesima - nekritična sekcija
    - ❖ izvršava kod kritične sekcije, odnosno pristupa deljenim podacima, vrši interakciju sa drugim procesima
- ```
process writer;      process reader;
var nextCoord : Coord; var nextCoord : Coord;
loop                loop
  compute(nextCoord);  nextCoord := coord;
  coord := nextCoord;  moveTo(nextCoord);
end loop;            end loop;
```
- Privatno izračunavanje, nekritična sekcija
- Interakcija, kritična sekcija
- ❖ Potrebno je pre ulaska u kritičnu sekciju izvršiti neki *ulazni protokol* (*entry protocol*) koji će obezbediti međusobno isključenje, a na izlazu eventualno neki *izlazni protokol* (*exit protocol*)

Mart 2020.

Copyright 2020 by Dragan Milićev

97

Uposleno čekanje

- ❖ Kako bi se mogla implementirati sinhronizacija samo do sada poznatim konceptima sekvencijalnog programiranja?
- ❖ Opšti slučaj uslovne sinhronizacije rešava se jednostavno:

```
shared var cond : boolean := false;

process waiting;
  ""
  while not cond do null;
  cond := false;
  ""
end;

""
end;
```
- ❖ Uslovna sinhronizacija kod ograničenog bafera:

```
void BoundedBuffer::append (Data* d) {
  while (count==N);
  buffer[tail] = d;
  tail = (tail+1)%N;
  count++;
}
```

Uslovna sinhronizacija - čekanje

Uslovna sinhronizacija - signal

Mart 2020.

Copyright 2020 by Dragan Milićev

96

Uposleno čekanje

- ❖ Drugi pokušaj rešavanja - isto kao u prethodnom, samo su najava ulaska u kritičnu sekciju i uposleno čekanje zamenili mesta:

```
shared var flag1, flag2 : boolean := false;
```

```
process P1
```

```
begin
```

```
loop
```

```
while flag2 = true do null;
```

```
flag1 := true;
```

```
<critical section>
```

```
flag1 := false;
```

```
<non-critical section>
```

```
end
```

```
end P1;
```

```
process P2
```

```
begin
```

```
loop
```

```
while flag1 = true do null;
```

```
flag2 := true;
```

```
<critical section>
```

```
flag2 := false;
```

```
<non-critical section>
```

```
end
```

```
end P2;
```

- ❖ Problem - moguće je sledeći scenario: oba procesa pročitaju fleg onog drugog, zaključče da nije postavljen i prolaze u kritičnu sekciju - nije obezbeđeno međusobno isključenje!

- ❖ Ovakva neregularna situacija naziva se *utrkiom*ije (*race condition*): proces se “zatrićao”, pretekao nekog drugog, krenuo da radi nešto što ne bi trebalo, jer *nije obezbeđena sinhronizacija*

Mart 2020.

Copyright 2020 by Dragan Milićev

99

Uposleno čekanje

- ❖ Prvi pokušaj rešavanja - dva procesa koji se međusobno isključuju:

```
shared var flag1, flag2 : boolean := false;
```

```
process P1
```

```
begin
```

```
loop
```

```
flag1 := true;
```

```
while flag2 = true do null;
```

```
<critical section>
```

```
flag1 := false;
```

```
<non-critical section>
```

```
end
```

```
end P1;
```

```
process P2
```

```
begin
```

```
loop
```

```
flag2 := true;
```

```
while flag1 = true do null;
```

```
<critical section>
```

```
flag2 := false;
```

```
<non-critical section>
```

```
end
```

```
end P2;
```

- ❖ Osnovno pitanje: da li ovo rešenje obezbeđuje potrebnu sinhronizaciju, odnosno međusobno isključenje? Da. Dokazati! (Upuistro: pretpostaviti suprotno, da ne obezbeđuje, tj. da su oba procesa ušla u kritičnu sekciju i doći do kontradikcije)
- ❖ Problem - moguće je sledeći scenario: oba procesa postavse svoje flegove (“podignu zastavice”) na *true*, a onda se oba “zaglave” u petljama uposlenog čekanja neograničeno
- ❖ Ovakva neregularna situacija naziva se *živa blokada* (*livelock*): procesi se izvršavaju, ali su zaglavljeni u petljama uposlenog čekanja, jer su se međusobno uslovili - jedan čeka na uslov koji treba da ispuni drugi, i obratno (ili više takvih u krug)

Mart 2020.

Copyright 2020 by Dragan Milićev

98

Uposleno čekanje

- ❖ Rešenje - Petersonov algoritam (G. L. Peterson, "Myths About the Mutual Exclusion Problem", *Information Processing Letters*, 12(3) 1981):

```
shared var turn : integer := 1, flag1, flag2 : boolean := false;
process P1
begin
  loop
    flag1 := true; turn := 2;
    while flag2 and turn = 2 do null;
  <critical section>
    flag1 := false;
  <non-critical section>
end
end P1;

process P2
begin
  loop
    flag2 := true; turn := 1;
    while flag1 and turn = 1 do null;
  <critical section>
    flag2 := false;
  <non-critical section>
end
end P2;
```

- ❖ Da li ovo rešenje obezbeđuje međusobno isključenje? Da. Dokazati! (Uputstvo: dokaz kontradikcijom)

- ❖ Da li ovo rešenje ima problem žive blokade? Ne. Dokazati! (Uputstvo: dokaz kontradikcijom, uz pretpostavku da oba procesa uposleno čekaju, opet kontradiktoran zaključak da *turn* ima i vrednost 1 i vrednost 2) Upravo promeniš *turn* "prelama" u situaciji kada oba procesa žele da uđu u kritičnu sekciju: ona ima vrednost 1 ili 2 (neodređeno, koju godi), koja odlučuje o tome koji će proces proći

- ❖ Da li ovo rešenje ima suvišnu sinronizaciju, tj. da li nameće naizmeničnost? Ne: proces čeka samo ako i onaj drugi želi da uđe u kritičnu sekciju, a red je na njega; ako onaj drugi uopšte ne želi da uđe, nema čekanja

Mart 2020.

Copyright 2020 by Dragan Milićev

101

Uposleno čekanje

- ❖ Treći pokušaj rešavanja - uvesti redosled izvršavanja; promenljiva *turn* ukazuje na to koji proces je na redu da uđe u sekciju:
shared var turn : integer := 1;

```
process P1
begin
  loop
    while turn = 2 do null;
  <critical section>
    turn := 2;
  <non-critical section>
end
end P1;

process P2
begin
  loop
    while turn = 1 do null;
  <critical section>
    turn := 1;
  <non-critical section>
end
end P2;
```

- ❖ Da li ovo rešenje obezbeđuje međusobno isključenje? Da. Dokazati! (Uputstvo: dokaz kontradikcijom, polazeći od suprotne pretpostavke, kojom se dolazi do kontradikcije da *turn* nema ni vrednost 1 ni vrednost 2, a ne može imati nijednu treću vrednost)
- ❖ Da li ovo rešenje ima problem žive blokade? Ne. Dokazati! (Uputstvo: dokaz kontradikcijom, uz pretpostavku da oba procesa uposleno čekaju, opet kontradiktoran zaključak da *turn* ima i vrednost 1 i vrednost 2)
- ❖ Problem: ovo rešenje nameće strogu naizmeničnost ulaska procesa u kritičnu sekciju, što je suvišna sinhronizacija
- ❖ Ukoliko neki proces ne želi da uđe u kritičnu sekciju, ne treba zadržavati druge. Ako napredovanje procesa, tj. ulazak u kritičnu sekciju neće izazvati nikakav problem (ovde zato što drugi proces i ne želi da uđe u nju), ne treba ga zadržavati

Mart 2020.

Copyright 2020 by Dragan Milićev

100

Podrška hardvera

- ❖ Kako obezbediti međusobno isključenje unutar samog kernela?
- ❖ U najjednostavnijem slučaju, ceo kod kernela može se posmatrati kao kritična sekcija: kada procesor (bilo koji od potencijalno više njih) uđe u izvršavanje nekog dela koda kernela, nijedan drugi procesor (pa ni on sam) ne bi smeo da uđe u isti ili bilo koji drugi deo kernela, kako ne bi napravio konflikte na strukturama podataka kernela
- ❖ To znači da je pri svakom ulasku u kernel kod, na svim mestima, potrebno uraditi neki ulazni protokol, “zaključavanje” ulaska u kritičnu sekciju:
`lock();`
a na svakom izlasku izlazni protokol, “otključavanje”:
`unlock();`

Mart 2020.

Copyright 2020 by Dragan Milićev

103

Uposleno čekanje

- ❖ Problemi ovog rešenja:
 - ❖ ovi procesi moraju da znaju jedan za drugog, tj. njihov kod zavisi od postojanja onog drugog (uočiti upotrebu deljenih promenljivih *flag1*, *flag2* i *turn*)
 - ❖ ovo je rešenje za dva procesa; postoji i opštije rešenje za n procesa, ali je ono složenije, i svakako je neophodno znati taj ograničen broj procesa n
 - ❖ U praktičnim primenama potrebno je obezbediti međusobno isključenje tako da kod procesa ne zavisi od postojanja drugih, i tako da je njihov broj neograničen; primer su kritične sekcije samog kernela
 - ❖ U svakom slučaju, uposleno čekanje ima taj nedostatak što procesi nepotrebno troše procesorsko vreme, izvršavajući petlje uposlenog čekanja

Mart 2020.

Copyright 2020 by Dragan Milićev

102

Podrška hardvera

- ❖ Međutim, ovo nije dovoljno u (simetričnim) multiprocesorskim sistemima: jedan procesor može da uđe u kritičnu sekciju (kod kernela), maskira prekide (i time se “zaštiti od samog sebe”), ali ništa ne sprečava neki drugi procesor da uradi to isto i uđe u kod kernela
- ❖ Za međusobno isključenje izvršavanja na više procesora potrebna je podrška hardvera: atomičnost na jednom nivou apstrakcije ne može se napraviti “ni iz čega”, uvek se mora napraviti korišćenjem nekog koncepta ili mehanizma koji obezbeđuje atomičnost na nižem nivou apstrakcije. U ovom slučaju, pošto se radi o softveru na najnižem nivou apstrakcije, tj. kodu kernela, potrebna je podrška hardvera
- ❖ Kako su multiprocesorski sistemi odavno u širokoj upotrebi, procesori odavno imaju takvu podršku u vidu instrukcija koje mogu da imaju različit, ali vrlo sličan oblik i semantiku; dva osnovna tipa ovakvih instrukcija su:

- ❖ *test-and-set*
- ❖ *stamp*

Mart 2020.

Copyright 2020 by Dragan Milićev

105

Podrška hardvera

- ❖ Posmatrajmo najpre samo jedan procesor koji je krenuo u izvršavanje koda kernela kao posledica sistemskog poziva, izuzetka ili prekida
- ❖ Šta je to što može uzrokovati da taj isti procesor prekinе tekuće izvršavanje i pređe na izvršavanje nekog drugog ili ponovo istog dela koda kernela i tako napravi konflikt?
- ❖ Jedino spoljašnji, asinhroni prekid, jer sam kod kernela neće to uraditi eksplicitno ili implicitno, odnosno sinhrono, samom instrukcijom:
 - ❖ taj kod nije pisan tako da sam sebi pravi konflikte, tj. da “nekontrolisano skače” iz jedne u drugu kritičnu sekciju, odnosno da prekida neku započetu operaciju pre njenog završetka
 - ❖ ukoliko bi neka instrukcija kernela napravila neki izuzetak, to se može smatrati fatalnim problemom u radu kernela, odnosno njegovom neispravnosti, i od toga po pravilu nema oporavka - ceo sistem otkazuje
- ❖ Prema tome, dovoljno za vreme izvršavanja kritične sekcije ignorisati prekide, što znači da:
 - ❖ *lock()* podrazumeva maskiranje spoljašnjih prekida
 - ❖ *unlock()* podrazumeva demaskiranje spoljašnjih prekida
- ❖ Kako procesori po pravilu implicitno maskiraju prekide pri obradi izuzetka/sistemskog poziva/prekida, a po povratku restauriraju stanje registara (pa i registra za maskiranje prekida), ove radnje često nije ni neophodno eksplicitno raditi
- ❖ Druga varijanta: ne maskirati prekide, već samo odložiti njihovu obradu, odnosno ne ulaziti ponovo u kernel kod odmah na pojavu prekida. Slično kao što i procesor radi - zapamti pojavu signala zahteva za prekid, pa ga obradi tek kada završi tekuću instrukciju, tako i kernel može u prekidnoj rutini na asinhroni prekid samo zabeležiti da se prekid dogodio, a onda, kada obradi tekuću operaciju, sekvencijalno, na kraju obrade, proveriti šta se u međuvremenu dogodilo, tj. koji prekidi su se dogodili, i onda ih obraditi jedan po jedan, sinhrono i sekvencijalno

Mart 2020.

Copyright 2020 by Dragan Milićev

104

Podrška hardvera

- ❖ Upotreba ove instrukcije onda izlegla ovako:
 - ❖ svakoj kritičnoj sekciji, tačnije deljenoj strukturi podataka kojoj pristupa kod kritičnih sekcija koje treba međusobno isključiti, pridruži se jedna globalna, deljena promenljiva L , u deljenoj operativnoj memoriji multiprocesora; na primer, ako je ceo kernel jedna kritična sekcija, postoji jedna ovakva promenljiva
 - ❖ ova promenljiva je Bulovog tipa: vrednost *false* (0) označava da je kritična sekcija “otključana”, vrednost *true* (1) da je “zaključana” (i procesor ne može da uđe u nju, jer je neki drugi procesor već ušao u kritičnu sekciju)
 - ❖ operacija *lock* izgleda ovako (može biti parametrizovana adresom promenljive L):

```
lock(L):  
while test_and_set(L) do null;
```
 - ❖ operacija *unlock* izgleda ovako:

```
unlock(L):  
L:=0;
```

Ako je drugi procesor ušao i zaključao sekciju, *test_and_set* će vratiti *true* i upisati isto, tako da se L ne menja; ako nije, *test_and_set* vraća *false* a upisuje *true*, čime zaključava sekciju, ali je važna nedeljivost ove dve operacije, pa nijedan drugi procesor ne može uraditi isto između te dve operacije

Mart 2020.

Copyright 2020 by Dragan Milicev

107

Podrška hardvera

- ❖ Instrukcija tipa *test-and-set* atomično radi sledeće: čita i vraća vrednost sadržaja zadate (adresirane) memorijske lokacije, a u tu lokaciju postavlja vrednost 1
- ❖ Ova atomičnost obezbeđuje se na nivou hardvera procesora: kako procesor treba da izvrši ciklus čitanja i ciklus upisa u istu memorijsku lokaciju, on ta dva ciklusa obavlja jedan za drugim, ali ne oslobađa pristup magistrali između ta dva ciklusa; kada najpre zauzme magistralu, otpušta je tek kada završi i ciklus upisa, tako da drugi procesori ne mogu da izvrše cikluse na magistrali (i eventualno urade ovo isto) između te dve operacije (čitanja i upisa iste lokacije)
- ❖ Ovo “zaključavanje magistrale” je ponovo međusobno isključenje, ali ga obezbeđuje hardver, odgovarajućim protokolima pristupa magistrali, i taj mehanizam je nevidljiv za softver

Mart 2020.

Copyright 2020 by Dragan Milicev

106

Podrška hardware

- ❖ Ovaj mehanizam “vrtenja” radi zaključavanja (*spin lock*) jeste jedan oblik uposlenog čekanja. Da li je to neefikasno?
- ❖ Donekle da, ali nema druge - nekad se mora malo i sačekati. To, međutim, nije problem u praksi, jer:
 - ❖ kernel kod nije maliciozan i nema greške (barem ne one svesno napravljene), pa je trajanje izvršavanja kritične sekcije uvek ograničeno i predvidivo
 - ❖ izvršavanje kritičnih sekcija je relativno kratko (reda nekoliko stotina ili hiljada mašinskih instrukcija)
 - ❖ jedan procesor će uposleno čekati dok onaj drugi koji je zaključao sekciju ne izađe iz nje, što je relativno kratko, ali svakako ograničeno i predvidivo
- ❖ Da li se ovo rešenje može primeniti i na samo jednom procesoru? U principu da, ali nema mnogo smisla, jer ako taj jedan procesor uđe u kritičnu sekciju, a ne završi je i ne otključa, pa onda ponovo želi da uđe u nju, to može biti samo kao posledica prekida; onda će on čekati sve dok se ponovo ne dogodi prekid i promena konteksta, procesor vrati na prekinuto izvršavanje koje onda izađe iz kritične sekcije i otključa je - besmisleno

Mart 2020.

Copyright 2020 by Dragan Miličev

109

Podrška hardware

- ❖ Instrukcija tipa *swap* ponaša se slično, ona atomično zamenjuje vrednost registra i adresirane memorijske lokacije
- ❖ Ova atomičnost implementira se na potpuno isti način, zaključavanjem magistrale i obavljanjem oba ciklusa čitanja i ciklusa upisa u adresiranu memorijsku lokaciju bez oslobađanja jednom zauzete magistrale
- ❖ Upotreba ove instrukcije onda izgleda vrlo slično kao i za instrukciju *test-and-set*:
 - ❖ kritičnoj sekciji, tačnije deljenoj strukturi podataka kojoj pristupa kod kritičnih sekcija, pridruži se jedna globalna, deljena promenljiva *L*, u deljenoj operativnoj memoriji multiprocссора, sa istim značenjem kao i ranije
 - ❖ operacija *lock* izgleda ovako (može biti parametrizovana adresom promenljive *L*):

```
lock(L):  
    locked = 1;  
    while (locked) swap(locked, L);
```
 - ❖ operacija *unlock* izgleda ovako:

```
unlock(L):  
    L = 0;
```

- ❖ Operacije *test_and_set* i *swap* rade dva pristupa memoriji; moguća je mala optimizacija, jer nema potrebe raditi i čitanje i upis, ako je sekcija zaključana; tek ako se vidi da nije, može se uraditi ova zamena:

```
lock(L):  
    locked = 1;  
    while (locked) {  
        while (L);  
        swap(locked, L);  
    }
```

Sve dok je sekcija zaključana, nemoj ni da pokušavaš da zaključaš, nego čekaj (*while(L)*). Tek ako je otključana, probaj isto što i ranije, ali se u međuvremenu moglo dogoditi to da neki drugi procesor uradi isto, pa je zato potrebno ponovo ispiрати *locked* u spoljašnjoj petlji *while*

Mart 2020.

Copyright 2020 by Dragan Miličev

108

Podrška hardvera

- ❖ Iako se na ovaj način postiže bolji odziv sistema, implementacija kernela postaje značajno složenija, jer ovakav pristup nosi određene rizike i probleme
- ❖ Osnovni uzrok tih potencijalnih problema jeste potreba da se izvršavanje kritičnih sekcija ugnežđuje, tako da izvršavanje uđe u novu kritičnu sekciju pre nego što iz prethodne izađe; ovo je vrlo logično imajući u vidu prirodne potrebe za ugnežđivanjem poziva potprograma unutar kernela zbog obavljanja operacija na više struktura podataka:

```
lock(L1);  
..  
lock(L2);  
..  
unlock(L2);  
..  
unlock(L1);
```

Ove kritične sekcije, tj. pozivi *lock()* i *unlock()* mogu biti u različitim potprogramima koji pristupaju različitim strukturama podataka, a čiji se pozivi ugnežđuju
- ❖ Prvi problem koji treba rešiti je sledeći: sada se ne sme uvek i bezuslovno u svakoj operaciji *unlock* vršiti demaskiranje prekida, jer ako je ona ugnežđena, demaskiraće prekide pre vremena, dok je još uvek u toku izvršavanje spoljašnje kritične sekcije
- ❖ Ovo se može rešiti brojanjem dubine ugnežđivanja: uvede se jedan globalni brojač sa inicijalnom vrednošću 0; u svakoj operaciji *lock* on se inkrementira, a u operaciji *unlock* dekrementira; na taj način on broji nivo (dubinu) ugnežđivanja; prekid se demaskira u operaciji *unlock* samo ako je ovaj nivo 0 (izlazak iz krajnje spoljašnje sekcije)

Mart 2020.

Copyright 2020 by Dragan Milićev

111

Podrška hardvera

- ❖ Već je rečeno da je najjednostavniji pristup u konstrukciji kernela taj da se ceo kernel kod smatra kritičnom sekcijom, tako da se međusobno isključenje obavlja na svakom ulazu u kernel kod (kod obrade svakog sistemskog poziva, izuzetka i prekida)
- ❖ Na taj način kernel ne može vršiti preotimanje dok se izvršava bilo koji deo koda kernela - kaže se da je kernel bez preotimanja (*non-preemptive*)
- ❖ Naprednije izvedbe kernela omogućavaju preotimanje i tokom izvršavanja koda kernela, osim u kritičnim skecijama koje se prave na manjim celinama, sa finijom granularnošću:
 - ❖ identifikuju se strukture podataka kojima pristupaju različiti delovi kernela, odnosno kojima se može pristupati iz različitih tokova kontrole (uključujući i pristup sa različitih procesora)
 - ❖ operacije koje pristupaju tim strukturama definišu se kao kritične sekcije
 - ❖ svakoj takvoj deljenoj strukturi podataka se pridruži promenljiva za zaključavanje (*L* u ranijim primerima), a na ulazu u operacije koje pristupaju toj strukturi, i koje predstavljaju kritične sekcije, kao i na izlazu iz njih izvrše opisane operacije *lock* odnosno *unlock*

Mart 2020.

Copyright 2020 by Dragan Milićev

110

Podrška hardvera

- ❖ Treba primetiti to da ovo rešenje ne zavisi od broja procesora koji izvršavaju uporedne tokove kontrole i koji ulaze u kritične sekcije koje se međusobno isključuju
 - ❖ Da li se onda opisana rešenja mogu koristiti za implementaciju međusobnog isključenja korisničkih procesa?
 - ❖ Ne, jer kritične sekcije korisničkih procesa, odnosno kod koji se izvršava u njima ničim nije kontrolisan:
 - ❖ taj kod može otkazati, odnosno generisati izuzetak, ili se proces može ugasi; treba onda rešiti ovakvu situaciju sekcija koje je proces zaključao, a nije otključao
 - ❖ taj kod može biti, zbog slučajne greške ili malicioznog koda, neograničeno dugo trajanja
 - ❖ čak i ako je ograničen, on može trajati previše dugo
- Za to vreme, dok je kritična sekcija zaključana, prekidi su maskirani, pa je procesor neosetljiv na događaje spolja, ne reaguje na njih. Osim toga, uposlono čekanje bespotrebno troši procesorsko vreme, a može da traje predugo
- ❖ Zbog toga se za međusobno isključenje i uslovnu sinhronizaciju korisničkih procesa koriste drugi, apstraktniji koncepti koje obezbeđuje OS, dok se opisane tehnike koriste samo za implementaciju (manjih i kontrolisanih) kritičnih sekcija samog kernela

Mart 2020.

Copyright 2020 by Dragan Milićev

113

Podrška hardvera

- ❖ Drugi problem potiče od različitih redosleda ugnežđivanja kritičnih sekcija:

Jedan tok kontrole/ procesor izvršava:	Drugi tok kontrole/ procesor izvršava:
lock(L1);	lock(L2);
“	lock(L1);
lock(L2);	“
“	unlock(L1);
unlock(L1);	“
	unlock(L2);
- ❖ Scenario: prvi tok uđe u kritičnu sekciju $L1$ i zaključa je; drugi uđe u kritičnu sekciju $L2$ i zaključa je, i sada oba toka kontrole čekaju da uđu u onu drugu sekciju koja je trajno zaključana
- ❖ Ovi tokovi kontrole će se *kružno blokirati*, u ovom slučaju u živu blokadu, jer vrše uposlono čekanje
- ❖ U opštijem slučaju, n ovakvih tokova kontrole se može kružno blokirati: prvi drži zaključanu sekciju $L1$ i čeka da uđe u sekciju $L2$; drugi drži zaključanu sekciju $L2$ i čeka da uđe u sekciju $L3$ itd, n -ti drži zaključanu sekciju L_n i čeka da uđe u sekciju $L1$
- ❖ Jedno (ali ne i jedino, niti uvek primenljivo) rešenje može da bude uvođenje uvek istog, monotonom redosleda ugnežđivanja kritičnih sekcija
- ❖ O ovom problemu i načinima njegovog rešavanja više detalja u predmetu OS2

Mart 2020.

Copyright 2020 by Dragan Milićev

112

Semafori

- ❖ Moguća je i nešto drugačija, ali potpuno ekvivalentna definicija semantike operacija nad semaforom (pokazati ekvivalenost sa prethodnom!):
 - ❖ semafor ima nenegativnu celobrojnu vrednost
 - ❖ *wait*: ako je vrednost semafora veća od 0, ta vrednost se dekrementira, a proces koji je izvršio ovu operaciju nastavlja izvršavanje; u suprotnom, ako je vrednost bila 0, proces mora da čeka na semaforu dok vrednost ne postane veća od 0, a tada je dekrementira i nastavlja
 - ❖ *signal*: vrednost semafora se uvećava za jedan
- ❖ Suštinski detalj je to da je izvršavanje ovih operacija nad semaforom atomično, tj. da su ove operacije izolovane - dok se izvršava neka operacija nad nekim semaforom, ne mogu se uporedo izvršavati bilo koje operacije nad istim semaforom, odnosno tokom tih operacija nema interakcije procesa
- ❖ Ovu atomičnost mora da obezbedi implementacija semafora; ako semafor implementira OS, kao što ovde pretpostavljamo, onda OS obezbeđuje ovu atomičnost
- ❖ Još neki nazivi za ove operacije: *wait* - *pend*, *acquire*; *signal* - *post*, *release*

Mart 2020.

Copyright 2020 by Dragan Milićev

115

Semafori

- ❖ Jedan jednostavan, efikasan i veoma upotrebljavan koncept za sinhronizaciju procesa jeste *semafor* (*semaphore*); predložio ga je E. W. Dijkstra 1962. ili 1963. godine
- ❖ Semafor je objekat, promenljiva ili apstraktan tip podataka, koji ima svoje stanje, predstavljeno celobrojnou vrednošću, kao i dve operacije koje uporedni procesi mogu da vrše nad njim:
 - ❖ *wait* (Dijkstra je originalno označavao sa *P*, od holandskih reči *proberen* - probati, *passen* - proći ili *pakken* - zgrabiti): vrednost semafora se dekrementira, i ako je nakon toga postala manja od 0, proces koji je izvršio ovu operaciju mora da čeka na semaforu; u suprotnom, proces nastavlja izvršavanje
 - ❖ *signal* (Dijkstra je originalno označavao sa *V*, od holandskih reči *verhogen* - uvećati ili *vrijgave* - otpustiti): vrednost semafora se inkrementira, a ako je pre toga bila manja od 0, jedan proces koji je čeka na tom semaforu nastavlja svoje izvršavanje

Mart 2020.

Copyright 2020 by Dragan Milićev

114

Semafori

- ❖ Implementacija semafora, *val* je vrednost semafora:

1) $val > 0$: još *val* procesa može da izvrši operaciju *wait* a da se ne blokira, nema procesa blokiranih na semaforu

2) $val = 0$: nema blokiranih na semaforu, ali će se proces koji naredni izvrši *wait* blokirati

3) $val < 0$: ima $-val$ blokiranih procesa, a *wait* izaziva blokiranje

```
procedure wait(S) begin
```

```
  val:=val-1;
```

```
  if val<0 then begin
```

```
    suspend the running process by putting it into the queue of S;
```

```
    take another process from the ready queue and switch context to it
```

```
  end
```

```
end;
```

```
procedure signal(S) begin
```

```
  val:=val+1;
```

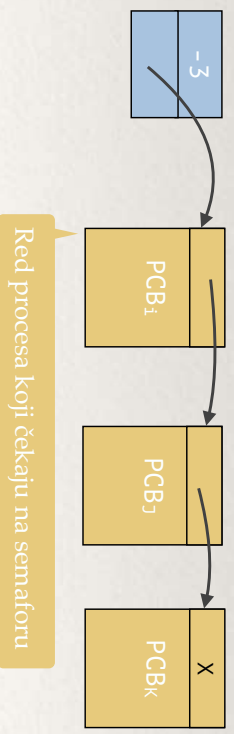
```
  if val<=0 then begin
```

```
    take one process from the suspended queue of S
```

```
    and unblock it by putting it into the ready queue
```

```
  end
```

```
end;
```



Mart 2020.

Copyright 2020 by Dragan Milićev

117

Semafori

- ❖ Kako implementirati čekanje procesa?

- ❖ Uposlenim čekanjem? Moguće, ali neefikasno - ideja jeste da se to izbegne

- ❖ *Suspension*: proces koji treba da čeka se suspenduje, tako što se on (tj. njegov PCB) ne vraća u skup spremnih, već se smešta u red procesa koji čekaju na tom semaforu; zato ovaj proces neće biti kandidat za dobijanje procesora sve dok se ne deblokira i (njegov PCB) ponovo vrati u red spremnih

- ❖ Svakom semaforu se tako pridružuje red čekanja procesa koji su blokirani na tom semaforu, a operacija *wait* je tako potencijalno blokirajuća: kada proces pozove *wait*, može se suspendovati. Proces ostaje suspendovan u redu čekanja na semaforu sve dok neki drugi proces ne izvrši *signal* kojim ga deblokira

- ❖ Važno je obezbediti da protokol (redosled) njihovog deblokiranja (otpuštanja) bude *pravedan (fair)*, kako neki proces ne bi neograničeno čekaao na semaforu zato što ga drugi procesi pretiču; ovakva pojava neograničenog čekanja procesa jer ga drugi prioritetniji pretiču naziva se *izgladnjivanje (starvation)*

- ❖ Najjednostavniji (ali ne i jedini) pravedan protokol je jednostavan FIFO protokol

- ❖ Kako implementirati atomičnost operacija na semaforu? Kao što je već opisano, tehnikama koje koriste podršku hardvera - maskiranje prekida i *spin lock*. Operacije na semaforu su kritične sekcije

Mart 2020.

Copyright 2020 by Dragan Milićev

116

Semafori

- ❖ Implementacija semafora u školskom jezgriu klasom *Semaphore* (nastavak):

```
void Semaphore::block () {
    if (setjmp(Thread::runningThread->context)==0) {
        blocked.put(Thread::runningThread);
        Thread::runningThread = Scheduler::get();
        longjmp(Thread::runningThread->context,1);
    } else return;
}

void Semaphore::unblock () {
    Thread* t = blocked.get();
    Scheduler::put(t);
}

void Semaphore::wait () {
    lock();
    if (--val<0) block();
    unlock();
}

void Semaphore::signal () {
    lock();
    if (++val<=0) unblock();
    unlock();
}
```

Mart 2020.

Copyright 2020 by Dragan Milićev

119

Semafori

- ❖ Implementacija semafora u školskom jezgriu klasom *Semaphore*:

```
class Semaphore {
public:
    Semaphore (unsigned short init=1) : val(init) {}

    void wait ();
    void signal ();

    int value () const { return val; }

protected:
    void block ();
    void unblock ();

private:
    int val;
    Queue blocked;
};

int lck = 0; // lock
```

Mart 2020.

Copyright 2020 by Dragan Milićev

118

Semafori

POSIX sistemski pozivi za operacije sa semaforima:

- ❖ "Otvaranje semafora", vraća pokazivač na deskriptor otvorenog semafora:

```
#include <fcntl.h>           /* For 0 * constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

Prvi parametar zadaje simboličko ime semafora kojim uporedni procesi mogu da identifikuju, odnosno referenciraju isti semafor pri otvaranju. Ovaj niz znakova mora početi znakom / i ograničene je dužine.

Drugi parametar definiše modalitete ove operacije. Zadaje se postavljanjem odgovarajućih bita u binarnoj predstavi. Za te potrebe definisane su (u <fcntl.h>) simboličke konstante koje predstavljaju vrednosti sa postavljenim bitima na odgovarajućim mestima.

Ukoliko je potrebno uključiti više ovakvih bita, dostavlja se argument koji je rezultat operacije *ili po bitima (bitwise or, |)*

Ako je uključen bit *O_CREATE*, onda će prvi proces koji izvrši ovakav poziv navodeći neko simboličko ime "Kreirati" semafor, ukoliko semafor sa tim simboličkim imenom već ne postoji; u suprotnom, samo će ga "otvoriti", odnosno dobiti referencu na isti taj semafor koji je drugi proces već kreirao. Na ovaj način procesi mogu da dele isti semafor. Ukoliko bit *O_CREATE* nije postavljen, ovaj sistemski poziv će vratiti grešku ako semafor ne postoji

Ako su postavljeni i *O_CREATE* i *O_EXCL*, onda će poziv vratiti grešku ako semafor sa datim imenom već postoji (način da semafor ne bude deljen, nego ekskluzivan)

Parametar *mode* definiše prava pristupa, kao za fajlove (detalji kasnije)

Parametar *value* zadaje inicijalnu vrednost semafora

- ❖ Operacije *wait* i *signal* na semaforu (ovo su samo neke funkcije, ima i drugih); vraćaju 0 u slučaju uspeha, -1 u slučaju greške:

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

Mart 2020.

Copyright 2020 by Dragan Milićev

121

Semafori

- ❖ Upotreba semafora za međusobno isključenje:

```
shared var mutex : semaphore := 1;
```

```
process P1
begin
  loop
    wait(mutex);
    <critical section>
    signal(mutex);
    <non-critical section>
  end
end P1;
```

```
process P2
begin
  loop
    wait(mutex);
    <critical section>
    signal(mutex);
    <non-critical section>
  end
end P2;
```

- ❖ Pogodnost: kod je jednostavan i jednobrazan - isti u svim procesima; taj kod više ne zavisi od toga koliko procesa ima, kao ni od koda drugih procesa
- ❖ Potencijalan problem: operacije *wait* i *signal* moraju da budu propisnu uparene:

- ❖ Šta ako se greškom izostavi *wait*? Neće biti obezbeđeno međusobno isključenje (utrkivanje)
- ❖ Šta ako se greškom izostavi *signal*? Kritična sekcija će ostati trajno zaključana

Mart 2020.

Copyright 2020 by Dragan Milićev

120

Semafori

- ❖ Međusobno isključenje se lako može i uopštiti na slučaj kada najviše N procesa sme da uđe u kritičnu sekciju, pristupi nekom resursu i slično, npr. zato što ima N raspoloživih identičnih instanci tog resursa koje procesi zauzimaju (*wait*) i oslobađaju (*signal*):
`shared var mutex : semaphore := N;`

```
process P
..
wait(mutex);
<critical section>
signal(mutex);
..
end P;
```

Prvi proces koji izvrši *wait* nalazi na vrednost N , prolazi bez čekanja i smanjuje vrednost semafora na $N-1$. Drugi proces nalazi na $N-1$, prolazi i smanjuje na $N-2$ itd. N -ti proces nalazi na vrednost 1, prolazi i smanjuje na 0. Već sledeći će morati da čeka dok neki od prethodnih ne izađe

Mart 2020.

Copyright 2020 by Dragan Milićev

123

Semafori

- ❖ Upotreba POSIX za međusobno isključenje:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

// Initialization:
const char* mutexName = "/myprogram_mutex";
sem_t* mutex = sem_open(mutexName, 0_CREAT, 0_RDWR, 1);

// Use for mutual exclusion:
sem_wait(mutex);
// Critical section
sem_post(mutex);
...

// Release the semaphore when it is no longer needed:
sem_close(mutex);
```

Kada semafor više nije potreban, potrebno ga je "zavrnuti" sistemskim pozivom *sem_close*, čime proces oslobađa referencu na semafor, a operativni sistem dealocira semafor ako ga nijedan proces više ne koristi

- ❖ Radi preglednijeg i konciznijeg zapisa, naredni primeri koriste konceptualan zapis

Mart 2020.

Copyright 2020 by Dragan Milićev

122

Semafori

- ❖ Primer sinhronizacije procesa - "pisac" i "čitalac":

```
type Coord = record x, y : integer end;

shared var coord : Coord := {0,0},
    readyToRead : semaphore := 0, readyToWrite : semaphore := 1;

process writer;
var nextCoord : Coord;
loop
    compute(nextCoord);
    readyToWrite.wait();
    coord := nextCoord;
    readyToRead.signal();
end loop;

process reader;
var nextCoord : Coord;
loop
    readyToRead.wait();
    nextCoord := coord;
    readyToWrite.signal();
    moveTo(nextCoord);
end loop;
```

- ❖ Opštiji slučaj n procesa koji rade "u lancu", svaki koristi proizvod prethodnog i prosleđuje svoj proizvod sledećem; sinhronizuju se pomoću $n-1$ semafora:

- ❖ P_1 proizvede svoj proizvod i signalizira semafor S_1 na kom čeka proces P_2
- ❖ P_2 čeka na S_1 , zatim proizvede svoj proizvod i signalizira semafor S_2 na kome čeka proces P_3
- ❖ itd, P_{n-1} čeka na S_{n-2} , zatim proizvede svoj proizvod i signalizira semafor S_{n-1} na kome čeka proces P_n

Mart 2020.

Copyright 2020 by Dragan Milićev

125

Semafori

- ❖ Uslovna sinhronizacija pomoću semafora:

```
shared var cond : semaphore := 0;
```

```
Waiting process:
```

```
..
wait(cond);
..
end;
```

```
Signalling process:
```

```
..
signal(cond);
..
end;
```

Ako prvi proces najpre naiđe na *wait*, čekaće dok drugi ne ispuní uslov i ne signalizira semafor.

Ako drugi proces prvi izvrši *signal*, povećaće vrednost semafora na 1, tako da se prvi proces neće suspendovati, nego će odmah proći *wait* bez čekanja

- ❖ Primetiti razliku u upotrebi semafora za međusobno isključenje i uslovnu sinhronizaciju:

- ❖ kod međusobog isključenja isti proces koji je semafor zatvorio operacijom *wait* i otvara taj semafor operacijom *signal* - ove dve operacije moraju biti uparene u kontekstu istog procesa
- ❖ kod uslovne sinhronizacije jedan proces čeka, a drugi signalizira deljeni semafor - ove dve operacije moraju biti uparene ali u kontekstima različitih procesa

Mart 2020.

Copyright 2020 by Dragan Milićev

124

Semafori

- ❖ Implementacija ograničenog bafera korišćenjem semafora školskog jezgra (nastavak):

```
BoundedBuffer::BoundedBuffer () :
    mutex(1), spaceAvailable(N), itemAvailable(0), head(0), tail(0) {}

void BoundedBuffer::append (Data* d) {
    spaceAvailable.wait();
    mutex.wait();
    buffer[tail] = d;
    tail = (tail+1)%N;
    mutex.signal();
    itemAvailable.signal();
}

Data* BoundedBuffer::take () {
    itemAvailable.wait();
    mutex.wait();
    Data* d = buffer[head];
    head = (head+1)%N;
    mutex.signal();
    spaceAvailable.signal();
    return d;
}
```

- ❖ Operacije *wait* i *signal* ne samo da moraju biti propisno uparene, već i njihov redosled može biti bitan. Šta će se desiti ako se samo međusobno zamene mesta susednih naredbi *spaceAvailable.wait()* i *mutex.wait()*?

Mart 2020.

Copyright 2020 by Dragan Milićev

127

Semafori

- ❖ Implementacija ograničenog bafera korišćenjem semafora školskog jezgra:

```
const int N = ...; // Capacity of the buffer
class Data;

class BoundedBuffer {
public:
    BoundedBuffer ();

    void append (Data*);
    Data* take ();

private:
    Semaphore mutex;
    Semaphore spaceAvailable, itemAvailable;

    Data* buffer[N];
    int head, tail;
};
```

Semafori za uslovnu sinhronizaciju:
spaceAvailable: semafor na kom će čekati proizvođači ukoliko je bafer pun, sve dok se u njemu ne pojavi slobodno mesto;
itemAvailable: semafor na kom će čekati potrošači ukoliko je bafer prazan, sve dok se u njemu ne pojavi element

Mart 2020.

Copyright 2020 by Dragan Milićev

126

Semafori

- ❖ U mnogim sistemima postoje varijante binarnih semafora koje se ponekad različito i zovu, a često uvode i neka ograničenja u zavisnosti od toga za koju namenu su predviđeni. Na primer:
 - ❖ *Mutex*: binarni semafor namenjen samo za međusobno isključenje kritičnih sekcija; poseduje ograničenje da samo proces koji je zatvorio semafor operacijom tipa *wait* može da ga otvori operacijom *signal*, u suprotnom se ova operacija smatra greškom; može da poseduje i neku dodatnu podršku za raspoređivanje po prioritetima u RT sistemima
 - ❖ *Događaj* (*event*) ili *signal*: služi za signalizaciju događaja, koji mogu doći i od hardvera, tj. od hardverskih uređaja koji izvršavaju uporedne radnje sa procesorom, pa se operacija *signal* može vezati i kao reakcija na spoljašnji prekid od hardvera (koji predstavlja apstraktan uporedni proces); mogu nametati i određena ograničenja, npr. to da samo proces koji je kreirao događaj i njegov je “vlasnik” može izvršiti operaciju *wait* i čekati na događaj, dok ga drugi procesi mogu signalizirati
- ❖ Mnogi sistemi podržavaju i složene operacije *wait* na dva ili više binarna semafora, pri čemu se pozivajući proces deblokira ako je signaliziran bilo koji od njih (logika *ili*) ili svaki od njih (logika *i*)

Mart 2020.

Copyright 2020 by Dragan Milićev

129

Semafori

- ❖ Do sada prikazani semafori nazivaju se često i *brojačkim* (*counting*), jer omogućavaju brojanje procesa koji prolaze *wait* u svojoj celobrojnoj vrednosti
- ❖ Za mnoge primene, kao što je pokazano, dovoljna je samo binarna, Bulova vrednost semafora:
 - ❖ kritična sekcija, podatak ili resurs je otključan ili zaključan, prolaz je otvoren ili zatvoren
 - ❖ uslov je ispunjen ili nije ispunjen
 - ❖ podatak je spreman ili nije spreman
 - ❖ događaj se dogodio ili se nije dogodio
- ❖ Zato mnogi sistemi podržavaju i posebne, *binarne semaphore* (*binary semaphore*) čija je semantika u osnovi jednostavna:
 - ❖ imaju samo dve vrednosti, 0 i 1
 - ❖ *wait*: ako je vrednost semafora 1, postavlja se na 0, a proces nastavlja; u suprotnom, proces čeka dok neki drugi proces ne izvrši *signal*
 - ❖ *signal*: ako postoje procesi koji čekaju, jedan se deblokira; u suprotnom, vrednost se postavlja na 1

Mart 2020.

Copyright 2020 by Dragan Milićev

128

Semafori

- ❖ Međutim, sa druge strane, semafori imaju i svoje nedostatke: pomalo paradoksalno, njihova najveća prednost - jednostavnost, jeste i njihova najveća mana
- ❖ Semafori su suviše jednostavan koncept, na niskom nivou apstrakcije
- ❖ Upotreba operacija *wait* i *signal* ni na koji način nije povezana sa onim što senafor "kontrolniše" i ne sugerise svrhu njegove upotrebe: za međusobno isključenje pristupa kritičnoj sekciji, za uslovnu sinhronizaciju; ove operacije nisu eksplicitno i jasno "pridružene" resursima koje kontrolišu (kritične sekcije, uslovi, deljene strukture)
- ❖ Ništa ne obavezuje programera da ove operacije koristi pregledno, pa se one lako "rasipaju" po kodu (po različitim procedurama), čak i kada se obavljaju u kontekstu jednog procesa (kod međusobnog isključenja)
- ❖ Osim toga, kod uslovne sinhronizacije, ove operacije su svakako rasute po kodu različitih procesa
- ❖ Zbog toga kod procesa koji koristi semafore lako postaje nepregledan, a time težak za razumevanje i održavanje
- ❖ Kao posledica toga, kod uporednih procesa postaje i podložan greškama koje mogu biti vrlo teške za otkrivanje:
 - ❖ operacije *wait* i *signal* moraju biti propisno uparene; u suprotnom, nastaju opisani problemi - ili izostaje sinhronizacija, ili resursi ostaju trajno zaključani
 - ❖ čak i ako su propisno uparene, redosled ovih operacija može biti bitan

Mart 2020.

Copyright 2020 by Dragan Milićev

131

Semafori

- ❖ Semafori su, sa jedne strane, veoma pogodan koncept jer su jednostavni za razumevanje i korišćenje u različite svrhe, kao i za implementaciju, a implementacija im je krajnje efikasna
- ❖ Zato su semafori podržani u praktično svim operativnim sistemima opšte namene, kao i u mnogim programskim jezicima i bibliotekama za konkurentno programiranje
- ❖ Čak i sam kernel, ukoliko je višenitni, sa preotimanjem, može koristiti semafore za sinhronizaciju u svojoj implementaciji:
 - ❖ za međusobno isključenje većih kritičnih sekcija koje pristupaju složenim strukturama podataka; naravno, za implementaciju samih semafora i operacija najužeg dela jezgra mora se koristiti podrška hardvera (maskiranje prekida i *spin lock*)
 - ❖ za uslovnu sinhronizaciju internih niti, recimo kod alokacije resursa, smeštanja i uzimanja zahteva iz redova čekanja, signaliziranja završenih operacija (npr. ulazno-izlaznih) i slično
- ❖ Osim toga, semafori se mogu koristiti u implementaciji složenijih, apstraktnijih koncepata za sinhronizaciju i komunikaciju između procesa u programskim jezicima ili izvršnim okruženjima

Mart 2020.

Copyright 2020 by Dragan Milićev

130

Semafori

- ❖ Ovakva neregularna situacija naziva se *mrtva* ili *kružna blokada* (*deadlock*)
- ❖ U opšijem slučaju, mrtva blokada nastaje kada se n procesa međusobno kružno blokira, tako što proces $P1$ drži zaključan resurs / semafor / kritičnu sekciju $R1$ a pri tom čeka na resurs $R2$, proces $P2$ drži zauzet resurs $R2$ a čeka na resurs $R3$ itd, proces Pn drži resurs Rn a čeka na resurs $R1$
- ❖ U odnosu na živu blokadu (*livelock*) razlika je u tome što se sada procesi ne izvršavaju, nego su trajno suspendovani, dok kod žive blokade neograničeno izvršavaju petlje uposlenog čekanja
- ❖ Međutim, suština nastanka oba problema je ista - u oba slučaja postoji kružni lanac procesa koji su zaglavljeni u odnosima držanja i čekanja na resurse
- ❖ Zbog toga se često ova razlika i zanemaruje, bitniji je taj kružni odnos držanja i čekanja, pa se obe ove pojave često nazivaju istim nazivom “mrtva blokada” (*deadlock*), a samo kada je bitna ova razlika koriste različiti nazivi
- ❖ Problem mrtve blokade je jedan od najozbiljnijih koji može nastupiti u konkurentnim programima, pa i operativnim sistemima
- ❖ Više detalja o uslovima nastanka mrtve blokade, kao i tehnikama i algoritmima sprečavanja, izbegavanja i detekcije mrtve blokade u predmetu OS2
- ❖ Zbog ovih nedostataka semafora, osmišljeni su apstraktniji koncepti za sinhronizaciju procesa čija je upotreba lakša, preglednija i robusnija na greške. Neke od njih razmatraćemo u predmetu OS2, a mnoge druge upoznaćete u predmetu “Konkurentno i distribuirano programiranje”

Mart 2020.

Copyright 2020 by Dragan Milićev

133

Semafori

- ❖ Primer: dva procesa imaju ugneždene kritične sekcije kontrolisane semaforima $S1$ i $S2$ čija je inicijalna vrednost 1; oba procesa propisno uparuju operacije *wait* i *signal* i čak i propisno ugnežđuju kritične sekcije:

```
process P1;  
  wait(S1);  
  wait(S2);  
  . . .  
  signal(S2);  
  signal(S1);  
end P1;  
  
process P2;  
  wait(S2);  
  wait(S1);  
  . . .  
  signal(S1);  
  signal(S2);  
end P2;
```
- ❖ Problem je što se može dogoditi sledeći scenario: $P1$ uradi *wait(S1)* i prođe bez blokade, ali spusti vrednost $S1$ na 0; $P2$ uradi *wait(S2)* i prođe bez blokade, ali spusti vrednost $S2$ na 0; onda svaki od njih uradi *wait* na onom drugom semaforu i tako ostaju neograničeno blokirani!

Mart 2020.

Copyright 2020 by Dragan Milićev

132

Druge tehnike sinhronizacije

- ❖ Do sada je za međusobno isključenje kritičnih sekcija, kod pristupa deljenim podacima, radi izbegavanja konflikata i korupcije tih podataka, primerljiv *pesimistički pristup kontroli konkurencnosti* (*pesimistic concurrency control*): zbog same mogućnosti da nastupi konflikt, unapred se obezbeđuje preventivna sinhronizacija, odgovarajućim režijskim operacijama na ulazu u kritičnu sekciju - zaključavanjem
- ❖ Ali da li je taj “pesimizam” uvek opravdan? Šta ako se konflikti dešavaju izuzetno retko, recimo zato što procesi retko uporedo pristupaju istim konkretnim podacima (objektima)? Zašto onda nepotrebno trošiti vreme na režijske operacije ako se konflikti možda uopšte neće desiti
- ❖ *Optimistički pristup kontroli konkurentnosti* (*optimistic concurrency control*) ili *optimističko zaključavanje* (*optimistic locking*) podrazumeva ponašanje kao da se konflikt neće dogoditi, tačnije, da je mala šansa da se on dogodi:
 - ❖ promena podataka obavlja se bez zaključavanja i čekanja (*no locking, no wait*),
 - ❖ ali pošto se konflikt ipak može dogoditi, sprovode se tehnike koje mogu da detektuju takav konflikt; u slučaju detektovanog konflikta, promena se otkazuje i pokušava ponovo

Mart 2020.

Copyright 2020 by Dragan Milićev

135

Druge tehnike sinhronizacije

- ❖ Do sada je naglašavano da je uposlono čekanje neefikasno, jer uzaludno troši procesorsko vreme za izvršavanje instrukcija ispitivanja uslova, i da je suspenzija efikasnija
- ❖ Generalno, to jeste tako, ali samo pod uslovom da je čekanje duže nego što traje sama suspenzija - i suspenzija, uključujući neophodnu promenu konteksta, predstavlja režijski trošak, jer procesor izvršava mnogo režijskih instrukcija
- ❖ Zato je u određenim situacijama efikasnije malo sačekati uposlono, nego odmah vršiti suspenziju, ukoliko je vreme čekanja kraće od vremena koje treba utrošiti na suspenziju i kasniju desuspenziju procesa
- ❖ Ali kako znati da li čekati i koliko čekati, a kad suspendovati proces? U opštem slučaju, to se ne može znati unapred, ali se može primeniti sledeća tehnika:
 - ❖ neka je vreme potrebno za suspenziju i promenu konteksta b
 - ❖ najpre krenuti u uposlono čekanje, sve dok uslov ne bude ispunjen ili ne prođe vreme b ; ako uslov bude ispunjen za kraće vreme, postigla se ušteda u odnosu na suspenziju
 - ❖ ako istekne vreme b , a uslov nije ispunjen, suspendovati proces; u ovom slučaju utrošeno je režijsko vreme $2b$
 - ❖ na ovaj način garantovano je utrošeno režijsko vreme u najgorem slučaju $2b$, a u povoljnijem slučaju kraće od b ; ako se koristi samo suspenzija, režijsko vreme je uvek b , a ako se koristi samo uposlono čekanje, vreme čekanja može biti neograničeno, i duže od $2b$

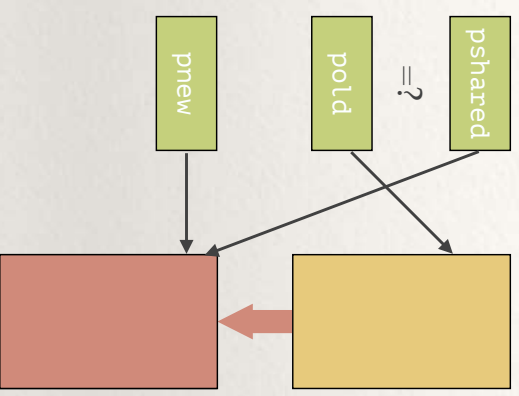
Mart 2020.

Copyright 2020 by Dragan Milićev

134

Druge tehnike sinhronizacije

- ❖ Ovaj pristup može se uopštiti i na proizvoljnu složenu strukturu podataka: na aktuelnu "verziju" strukture pokazuje neki deljeni pokazivač *pshared*, a proces vrši operaciju nad strukturom na sledeći način:
 - ❖ pročitaj vrednost pokazivača u neku privatnu promenljivu *polđ*
 - ❖ iskopiraj celu strukturu u neku svoju privatnu lokaciju na koju ukazuje privatni pokazivač *pnew*
 - ❖ izmeni svoju kopiju, tj. izvrši operaciju promene na kopiji
 - ❖ atomično uradi sledeće: ako je *pshared* i dalje jednako sa *polđ*, upiši *pnew* u *pshared*, čime aktuelna verzija postaje ova nova, izmenjena; u suprotnom, odustani od svega i kreni ispočetka
- ❖ Na najnižem nivou apstrakcije, implementacija ovog protokola zahteva podršku hardvera; na primer, Intel procesori noviji od i486 imaju instrukcije *CMPXCHG* (*compare and exchange*) za ovu namenu



Mart 2020.

Copyright 2020 by Dragan Milićev

137

Druge tehnike sinhronizacije

- ❖ Svrha međusobnog isključenja jeste u tome da se postigne izolacija, odnosno da efekat izvršavanja operacija nad deljenom promenljivom od strane više procesa ima serijalizovani efekat, tj. isti rezultat kao da su one izvršene sekvencijalno, jedna po jedna
- ❖ Na primer, neka je operacija nad deljenim podatkom (kritična sekcija) sledeća:
$$x := x+1$$
- ❖ Protokol optimističkog pristupa podrazumeva sledeće:
 - ❖ pročitaj vrednost deljene promenljive *x* u neku svoju privatnu, nedeljenu promenljivu, npr. u registar *R1*
 - ❖ pripremi novu vrednost za *x* u nekoj drugoj privatnoj promenljivoj, npr. u *R2*
 - ❖ atomično uradi sledeće: ako je vrednost *x* i dalje jednaka onom što je pročitano u *R1*, upiši novopripremljenu vrednost *R2* u *x*;
 - ❖ u suprotnom, detektovan je konflikt - neki drugi proces je u međuvremenu promenio vrednost *x*, pa bi upis vrednosti iz *R2* narušio serijalizaciju; zato se cela operacija otkazuje i ponavlja ispočetka
- ❖ Ovakav protokol garantuje serijalizovanost operacija. Ključan zahtev jeste atomičnost ovog poslednjeg koraka, provere i upisa, jer da nije tako, neki drugi proces bi mogao da izmeni podatak između te dve operacije
- ❖ Kao i ranije, ova atomičnost mora da se obezbedi na nižem nivou apstrakcije, npr. na nivou hardvera, atomičnom instrukcijom *poređenja i upisa* (*compare and set*)

Mart 2020.

Copyright 2020 by Dragan Milićev

136

Modeli međuprocесne komunikacije

- ❖ *Sinhronizacija* je samo jedan vid interakcije uporednih procesa
- ❖ Drugi vid je *međuprocесna komunikacija* (*interprocess communication, IPC*) - razmena informacija između procesa
- ❖ Ova dva pojma su povezana, jer se sinhronizacija najčešće i radi zbog logičke ispravnosti međuprocесne komunikacije, a konstrukt za međuprocесnu komunikaciju po pravilu podrazumevaju neku eksplicitnu ili implicitnu sinhronizaciju
- ❖ Postoje dva fundamentalna logička modela, dve paradigme međuprocесne komunikacije:
 - ❖ *deljena promenljiva* ili *deljeni objekat* ili *deljeni podatak* (*shared variable/object/data*): postoji deljeni podatak ili objekat kom mogu pristupati uporedni procesi, tako da neki od njih upisuju u taj podatak, a neki od njih iz njega čitaju, i na taj način razmenjuju informacije
 - ❖ *razmena poruka* (*message passing*): jedan proces eksplicitno, npr. sistemskim pozivom ili jezičkim konstruktom, zahteva slanje poruke određinom procesu ili procesima; proces primalac, ili više njih, eksplicitno traže prijem poruke

Mart 2020.

Copyright 2020 by Dragan Milićev

139

Druge tehnike sinhronizacije

- ❖ Optimistički pristup ima svoje prednosti u odnosu na pesimistički:
 - ❖ nema režijskih troškova za zaključavanje i suspenziju ili čekanje
 - ❖ lakše se rešavaju situacije u kojima proces koji je ušao u kritičnu sekciju otkáže, kod pesimističkog pristupa, potrebno je detektovati i rešiti ovu situaciju, tj. otključati resurse koje je taj proces bio zaključao; kod optimističkog pristupa nema nikakve potrebe za tim, jer proces koji je započeo operaciju i otkazao ionako ništa nije uradio nad deljenim strukturama, kao da nije ni postojao; pesimistički prístup može dovesti i do mrtve blokade
- ❖ Naravno, optimistički pristup ima svoju cenu: veliki broj ponovljenih pokušaja u situacijama čestih konflikata, kada se performanse naglo degradiraju i procesi mnogo zaustavljaju zbog neuspelih pokušaja, ponovljenih pokušaja i tako pravljenja novih konflikata. Može dovesti i do izglednijavanja. U takvim situacijama bolje je i efikasnije primeniti pesimistički pristup sa eksplicitnim zaključavanjem
- ❖ Optimistički pristup se mnogo primenjuje u praksi, u situacijama kada je on efikasan, a to su uslovi veoma retkih konflikata
- ❖ Na primer, sistemi za upravljanje bazama podataka odavno podižavaju, često i kao podrazumevani režim, optimističku kontrolu konkurentnosti pristupa deljenim podacima od strane uporednih procesa, računajući na retke konflikte, ali pružaju i mogućnost eksplicitnog zaključavanja za situacije kada su konflikti česti
- ❖ Ethernet prokol primenjuje sličnu strategiju, ali na nivou predajnika za slanje poruka preko zajedničke magistrale - zajedničkog mrežnog kablova. Nema nikakvih mehanizama zauzimanja i međusobnog isključenja pristupa magistrali, već svaki predajnik kreće da šalje poruku čim se poruka pojavi, ali istovremeno i "osluškuje" signal na magistrali; ako signal koji "čuje" nije isti kao onaj koji šalje, detektuje se konflikt, jer to znači da je i neki drugi predajnik uradio isto; tada se predajnik povlači neko slučajno vreme i potom pokušava slanje ponovo

Mart 2020.

Copyright 2020 by Dragan Milićev

138

Modeli međuprocесne komunikacije

- ❖ Razmena poruka podrazumeva odgovarajuće konstrukte jezika, bibliotečne ili sistemske pozive za slanje poruke i za prijem poruke
- ❖ Na primer, POSIX sistemski pozivi za slanje poruka preko tzv. *priključnice (socket)* - više detalja u predmetu OS2:
 - ❖ slanje poruke - niza bajova na koji ukazuje *buf* dužine *len* na priključnicu sa identifikatorom *sock*:
`ssize_t send(int sock, const void *buf, size_t len, int flags);`
 - ❖ prijem poruke sa priključnice sa identifikatorom *sock* u prostor na koji ukazuje *buf*:
`ssize_t recv(int sockfd, void *buf, size_t len, int flags);`
- ❖ Način na koji se razmena poruka između procesa implementira na istom računaru:
 - ❖ proces pošiljalac poziva sistemski poziv za slanje poruke
 - ❖ kernel po pravilu kopira poruku u svoj deo memorije, u svoje bafere
 - ❖ kada proces primalac zatraži prijem poruke, kernel kopira sadržaj poruke na mesto na koje je proces tražio
- ❖ Treba primetiti to da ovaj mehanizam za same procese izgleda apsolutno isto i u slučaju da se oni izvršavaju na različitim računarima - oni ne vide nikakvu razliku; naravno, implementacija je drugačija, jer kernel mora da razmenjuje poruke sa kernelom na drugom računaru, koristeći svoje module koji implementiraju odgovarajuće komunikacione protokole

Mart 2020.

Copyright 2020 by Dragan Milićev



141

Modeli međuprocесne komunikacije

- ❖ Iako ove dve paradigme jako podsećaju na arhitekturne paradigme i analogne su njima - multiprocesorskom sistemu (više procesora sa zajedničkom memorijom) i distribuiranom sistemu (više procesora bez zajedničke memorije, ali povezanih mrežom preko koje razmenjuju poruke) - ovde se radi o *logičkim paradigmatima* koje implementira softver, a ne o arhitekturnim, hardverskim paradigmatima
- ❖ Ova dva nivoa su različiti nivoi apstrakcije i implementacije (hardver odnosno softver), i u principu su nezavisni:
 - ❖ model deljene promenljive se lako implementira na multiprocesorskom sistemu, ali se može, nešto teže, implementirati i na distribuiranom sistemu
 - ❖ model razmene poruka se može jednostavno implementirati i na multiprocesorskim i na distribuiranim sistemima
- ❖ Model deljene promenljive je do sada već i podrazumevan i korišćen u svim primerima: deljena promenljiva ili struktura podataka, ograničeni bafer itd.
- ❖ Načini na koje procesi na istom računaru mogu da dele podatke:
 - ❖ podaci koje dele niti sa istim adresnim prostorom (statički i dinamički podaci)
 - ❖ deljena promenljiva kao koncept koji podržava programski jezik (zapravo opet između niti)
 - ❖ memorija koju proces logički deli sa drugim procesima i podaci organizovani u njoj
 - ❖ resursi koje proces dobije od operativnog sistema na korišćenje: semafor, fajl i drugi
- ❖ Na distribuiranom sistemu se ovakav koncept mora obezbediti korišćenjem razmene poruka između kernela na udaljenim računarima, a procesima stvoriti privid da pristupaju deljenom objektu; primer: pristup udaljenom fajlu

Mart 2020.

Copyright 2020 by Dragan Milićev

140

Modeli međuprocесne komunikacije

❖ Primer 1:

- ❖ procesi mogu razmenjivati informacije preko deljenog fajla - jedan proces upisuje u fajl, drugi iz njega čita; konceptualno, ovo je komunikacija po modelu deljenog objekta
- ❖ fajl može biti na udaljenom računaru, što je za proces transparentno, pa je za implementaciju pristupa tom fajlu potrebno da kernel na jednom računaru razmenjuje poruke sa kernelom na drugom računaru - model razmene poruka
- ❖ kada se pogleda implementacija kernela i njegovog modula za komunikaciju, mogu se uočiti uporedne kernel niti koje međusobno razmenjuju podatke preko ograničenih bafera - model deljenog objekta itd.

❖ Primer 2:

- ❖ aplikacije, koje se izvršavaju kao uporedni procesi, možda i na različitim računarima, pristupaju zajedničkoj bazi podataka, tako da neki čitaju, a neki upisuju podatke u bazi; ovo je model deljenih podataka, jer baza predstavlja složenu deljenu strukturu podataka
- ❖ međutim, na nivou implementacije, pristup do samih podataka u bazi ostvaruje poseban proces sistema za upravljanje bazom podataka (*database management system, DBMS*), pa klijentski procesi sa procesom baze razmenjuju poruke po odgovarajućem protokolu - model razmene poruka
- ❖ proces DBMS je, radi skalabilnosti i uporedne obrade, implementiran sa mnogo niti (na stotine njih), one opet međusobno komuniciraju preko bafera - model deljenog objekta
- ❖ sami podaci su smešteni u fajlove, često na udaljenim računarima, pa se opet koristi razmena poruka itd.

❖ Više detalja o još nekim konceptima međuprocесne komunikacije u predmetu OS2

Mart 2020.

Copyright 2020 by Dragan Milićev

143

Modeli međuprocесne komunikacije

- ❖ Oba modela međuprocесne komunikacije su podjednako rasprostranjena i u upotrebi u softverskom inženjerstvu: po pravilu se svaki konkretan problem može rešiti upotrebom i jednog i drugog modela, ali se vrlo često jedan problem lakše rešava jednom paradigmom, a neki drugi onom drugom
- ❖ Izbor jedne ili druge je pitanje konteksta: vrste problema, raspoloživih jezika, biblioteka, sistema i alata, ali i inženjerske procene o pogodnosti upotrebe jedne ili druge paradigme (složenost softvera, performanse, skalabilnost)
- ❖ Pitanje paradigme može biti i pitanje ugla posmatranja; na primer, ograničeni bafer se u principu posmatra kao deljeni objekat, jer mu pristupaju uporedni procesi koji u njega upisuju ili iz njega čitaju podatke, ali se on može posmatrati i kao međumedijum za indirektno slanje i prijem poruka između procesa
- ❖ Osim toga, obe ove paradigme se pojavljuju i na različitim nivoima apstrakcije, tako da se na višem nivou apstrakcije koristi jedan model, a on implementira na nižem nivou apstrakcije drugim modelom i tako dalje
- ❖ Zbog svega navedenog, oba ova modela komunikacije se u softverskoj praksi koriste praktično svakodnevno i to u najrazličitijim pojavnim oblicima, kao konstrukt programskih jezika, biblioteke za programiranje ili sistemski pozivi

Mart 2020.

Copyright 2020 by Dragan Milićev

142

Deo IV: Ulazno-izlazni podsistem

Mart 2020.

Copyright 2020 by Dragan Milićev

2

dr Dragan Milićev

redovni profesor

Elektrotehnički fakultet u Beogradu

dmilicev@etf.rs, www.rcub.bg.ac.rs/~dmilicev

Operativni sistemi

Osnovni kurs

Slajdovi za predavanja
Deo IV - Ulazno-izlazni podsistem

Za izbor slajda drži miša
uz levu ivicu ekrana

Mart 2020.

Copyright 2020 by Dragan Milićev

1

Karakteristike uređaja i usluga

- ❖ Ulazno-izlazni (U/I, *input-output*, I/O) uređaji koji se mogu priključiti na savremene računare su izuzetno raznoliki po nameni i načinu funkcionisanja, načinu i brzini prenosa podataka, načinu i kapacitetu skladištenja podataka itd.
- ❖ Kao i u svakoj složenosti i raznovrsnosti, osnovni metod za savladavanje te složenosti jeste *apstrakcija*: zanemarivanje manje bitnih različitosti, zard isticanja zajedničkih osobina i klasifikacija (grupisanje) različitih entiteta na osnovu tih zajedničkih osobina
- ❖ Na osnovu toga, U/I uređaji se klasifikuju prema različitim karakteristikama (aspektima) na određene kategorije
- ❖ Prema nameni:
 - ❖ za unos korisnikovih akcija: tastatura (*keyboard*), pokazivački uređaji - miš (*mouse*), pokazivač (*pointer*), ekran osetljiv na dodir (*touch screen*)
 - ❖ za prikaz slike: ekran (*display*), projektor (*projector*)
 - ❖ za skladištenje podataka: magnetni disk (*hard disk drive*, HDD), elektronska fleš memorija (*solid state drive*, SSD), skladište podataka (*storage*), magnetna traka (*magnetic tapes*), CD, DVD
 - ❖ štampač (*printer*)
 - ❖ za snimanje i reprodukciju zvuka: mikrofoni (*microphone*), zvučnici (*loudspeakers*), slušalice (*earphones*)
 - ❖ za mrežnu komunikaciju i povezivanje uređaja: mrežni adapter, univerzalna serijska magistrala (*Universal Serial Bus*, USB), Wi-Fi, Bluetooth, Thunderbolt
 - ❖ fotoaparati, video kamere (*photo/video cameras*)
 - ❖ mnogi, mnogi drugi uređaji

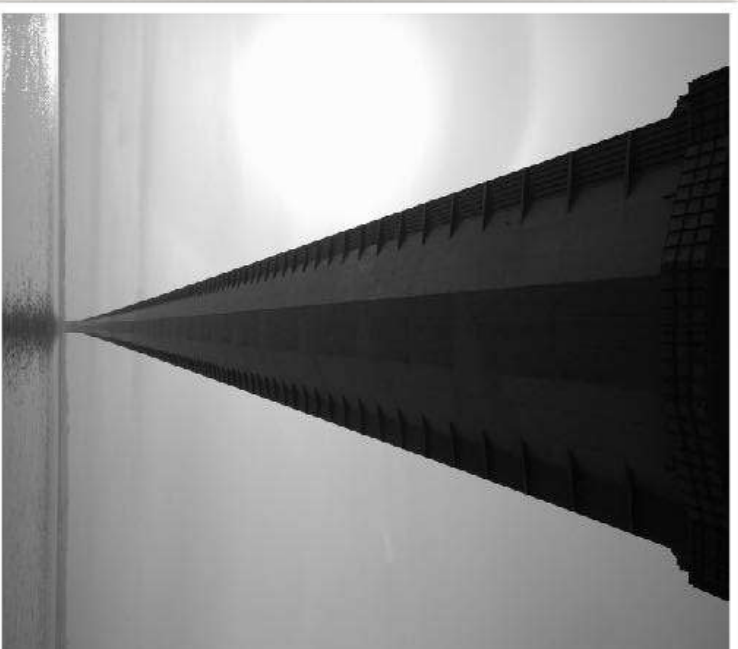
Mart 2020.

Copyright 2020 by Dragan Milićev

4

Glava 10: Interfejs U/I podсистема

- ❖ Karakteristike uređaja i usluga
- ❖ Generički pristup uređajima
- ❖ Znakovni tokovi
- ❖ Štampanje
- ❖ Razmena poruka preko priključnica
- ❖ Memorijski preslikani fajlovi
- ❖ Realno vreme
- ❖ Slika
- ❖ Zvuk



Mart 2020.

Copyright 2020 by Dragan Milićev

3

Karakteristike uređaja i usluga

- ❖ Prema načinu pristupa podacima:
 - ❖ sekvencijalni: pristup podacima je određen prirodnim, fizičkim, prostornim ili vremenskim redosledom kojim ti podaci stižu ili se šalju i ne može im se pristupiti proizvoljnim redosledom; npr. tastatura, zvučnik, mrežni adapter, ali i magnetna traka (podaci se čitaju ili upisuju po redu prolaska trake) itd.
 - ❖ sa direktnim pristupom: podacima se može pristupiti (na čitanje ili upis, u zavisnosti od vrste uređaja) u proizvoljnom redosledu, nezavisno od toga kako su oni prostorno smešteni ili vremenski pristigli; npr. diskovi, rasterski ekran
- ❖ Prema deljivosti, tj. tome da li uporedni procesi mogu slati parcijalne operacije uporedo ili ne:
 - ❖ *nedeljivi, posvećeni (dedicated)*: jedan proces mora obaviti celu operaciju sa uređajem pre nego što drugi proces obavi svoju veliku operaciju; tipičan primer je štampač: nema nikakvog smisla da jedan proces pošalje nekoliko znakova, a onda neki drugi uradi isto
 - ❖ *deljivi (share)*: može ih uporedo ili vremenski multipleksirano koristiti više procesa; npr. tastatura, ekran, miš, disk itd.
- ❖ Prema vremenskim karakteristikama prenosa podaka:
 - ❖ *sinhroni (synchronous)*: prenos vrše u pravilnim ili predvidivim vremenskim trenucima ili razmacima
 - ❖ *asinhroni (asynchronous)*: prenos vrše u nepravilnim ili nepredvidivim vremenskim trenucima ili razmacima

Mart 2020.

Copyright 2020 by Dragan Milićev

6

Karakteristike uređaja i usluga

- ❖ Prema smeru toka podataka:
 - ❖ ulazni: smer toka podataka je samo iz uređaja ka računaru (operativnoj memoriji); na primer: tastatura, miš, mikrofonski, CD-ROM itd.
 - ❖ izlazni: smer toka samo iz računara na uređaj; npr. ekran, zvučnik, štampač itd.
 - ❖ ulazno-izlazni: oba smera; osnovni primer: uređaji za skladištenje podataka koji omogućavaju i čitanje i upis podataka
- ❖ Prema jedinici prenosa podataka:
 - ❖ *znakovno orijentisani (character-oriented)*: vrše prenos malih jedinica podataka, tipično pojedinačnih znakova (ili bajtova); npr. linijski štampač, tastatura i znakovni ekran
 - ❖ *blokovski orijentisani (block-oriented)*: vrše prenos većih blokova podataka, fiksne ili promenljive dužine; npr. diskovi, mrežni adapteri itd.

Mart 2020.

Copyright 2020 by Dragan Milićev

5

Generički pristup uređajima

- ❖ Jedna važna i veoma korišćena kategorija usluga operativnih sistema vezanih za U/I operacije odnosi se na pristup fajlovima smeštenim na tim uređajima - o tome u sledećem delu
- ❖ Sistemi nalik sistemu Unix (*Unix-like systems*), a to uključuje sve varijante sistema Unix, Solaris, AIX, HP-UX, Linux, pa i Mac OS X i druge, poseduju tzv. *specijalne fajlove* (*special file*) ili *fajlove-uređaje* (*device file*) koje procesi "vide" kao virtuelne fajlove u virtuelnom direktorijumu */dev*, i sa kojima mogu da obavljaju uobičajene operacije kao sa običnim fajlovima; međutim, te operacije se zapravo preusmeravaju direktno ka odgovarajućim uređajima
- ❖ Ove "fajlove" instalira administrator sistema sistemskom komandom ili sistemskim pozivom *mknod*
- ❖ Prefiksi u nazivima ovih "fajlova" upućuju na tip uređaja, koji opet određuje način na koji će biti interpretirane U/I operacije. Na primer: *lp* (*line printer*, štampač), *tty* (*terminal*), *hda* (*hard drive*, IDE diskovi) itd.
- ❖ Ovi "fajlovi" mogu dozvoljavati znakovno orijentisane operacije ili blokovski orijentisane operacije
- ❖ Sistemski poziv koji omogućava da se ovakvim fajlovima direktno upućuju zahtevi koje onda uređaj interpretira na specifičan način:
`int ioctl (int fd, unsigned long request, ...);` *fd* je deskriptor prethodno otvorenog fajla-uređaja
- ❖ I drugi operativni sistemi, pa i Windows, imaju slične koncepte; Win32 API sistemski poziv sličan pozivu *ioctl* naziva se *DeviceControl*

Mart 2020.

Copyright 2020 by Dragan Milićev

8

Karakteristike uređaja i usluga

- ❖ Operativni sistemi na veoma različite načine obezbeđuju usluge vezane za uređaje korisnicima i njihovim procesima, kroz sistemske programe ili pozive
- ❖ Jedna važna karakteristika sistemskih poziva jeste to da li se kontrola vraća pozivaocu, odnosno da li se pozivajući tok kontrole nastavlja odmah, ili tek kada se obavi cela U/I operacija:
 - ❖ *sinhroni* (*synchronous*) ili *blokirajući* (*blocking*): pozivajući proces, odnosno tok kontrole se po potrebi suspenduje i nastavlja tek kada se zahtevana operacija završi
 - ❖ *asinhroni* (*asynchronous*) ili *neblokirajući* (*non-blocking*): pozivajući proces, odnosno tok kontrole se odmah nastavlja, sistemski poziv mu odmah vraća kontrolu, a zahtevana operacija se obavlja "odloženo", asinhrono
- ❖ Moguće su i međuvarijante, npr. tako da sistemski poziv odmah vrati kontrolu, zajedno sa informacijom (statusom) o tome da li je operacija izvršena u celini, delimično ili nije uopšte (sinhroni neblokirajući poziv)

Mart 2020.

Copyright 2020 by Dragan Milićev

7

Znakovni tokovi

- ❖ Ove uređaje proces "vidi" kao znakovno orijentisane, tekstualne fajlove, koji se identifikuju konstantama tipa *FILE** definisanim u `<stdio.h>`: *stdin*, *stdout* i *stderr*
- ❖ Ovakvi sekvencijalni, znakovno orijentisani fajlovi ili uređaji nazivaju se *znakovni tokovi* (*character streams*)
- ❖ Bibliotečne funkcije standardne biblioteke jezika C (`<stdio.h>`), ali i jezika C++ (`<iostream>`), koje rade sa znakovnim tokovima zapravo se oslanjaju na dva elementarna sistemska poziva:

```
int getc (FILE*);  
int putc (int c, FILE*);
```

getc učitava i vraća jedan znak iz zadatog ulaznog znakovnog toka
putc šalje dati znak na zadati izlazni znakovni tok
- ❖ Funkcije *getchar* i *putchar* podrazumevaju ulaz i izlaz na standardni ulazni i izlazni tok:
`getchar()` je isto što i `getc(stdin)`
`putchar(c)` je isto što i `putc(c, stdout)`
- ❖ Složenije funkcije standardne biblioteke jezika C, kao što su one iz familije *scanf* i *printf*, kao i operatorske funkcije *operator<<* i *operator>>* biblioteke jezika C++, zapravo vrše konverziju i formatizaciju vrednosti svojih argumenata, a onda se oslanjaju na sukcesivne pozive navedenih sistemskih usluga za učitavanje odnosno ispis jednog po jednog znaka na zadati znakovni tok

Mart 2020.

Copyright 2020 by Dragan Milićev

10

Znakovni tokovi

- ❖ Na sistemima nalik sistemu Unix (ali i na drugim postoje slični koncepti i važe slični principi), svakom procesu pridružena su uvek tri tzv. *standardna* znakovno orijentisana, sekvencijalna "uređaja":
 - ❖ *stdin*: standardni ulazni uređaj
 - ❖ *stdout*: standardni izlazni uređaj
 - ❖ *stderr*: standardni izlazni uređaj za uobičajeno izveštavanje o greškama u izvršavanju procesa, npr. za ispis poruka sistemskim pozivom *error*
- ❖ Ove uređaje proces dete podrazumevano nasleđuje od procesa roditelja. Na taj način CLI, kao proces, koji radi sa konzolom kao svojim *stdin* i *stdout*, usmerava standardne uređaje na istu tu konzolu, ali može i da ih preusmeri u komandnoj liniji recimo na stvarne tekstualne fajlove:

```
myprogram <input.txt >output.txt 2>error.txt
```

```
<input.txt preusmerava stdin na fajl input.txt  
>output.txt preusmerava stdout na fajl output.txt  
2>error.txt preusmerava stderr na fajl error.txt
```

Mart 2020.

Copyright 2020 by Dragan Milićev

9

Štampanje

- ❖ Prvobitni linijski štampači bili su jednostavni, sekvencijalni znakovni uređaji koji su štampali znak po znak, liniju po liniju. Zato se njima može pristupati na isti način kao i ostalim izlaznim znakovnim tokovima
- ❖ Današnji štampači omogućavaju da im se upute celi fajlovi sa sadržajem za štampu, i to u nekim standardnim formatima. Po pravilu, štampači će svakako prepoznati običan tekstualni sadržaj i odštampati ga kao običan tekst, ali prepoznaju i složenije, standardne vektorske formate opisa proizvoljnog sadržaja stranica, npr. PostScript
- ❖ Na sistemima nalik sistemu Unix, postoji sistemski program (komanda) *lpr* ili *lp* (postoje obe) koja sadržaj fajla zadatog kao argument šalje na podrazumevani štampač (štampač koji je konfigurisan kao podrazumevan u sistemu); ako argument nije zadat, ovaj program uzima sadržaj sa svog standardnog ulaza, tako da su moguće sledeće ekvivalentne varijante:

```
lpr mydoc.txt  
cat mydoc.txt | lpr
```
- ❖ Potpuno isti koncept postoji i u sistemu Windows i njegovom CLI koji se naziva *PowerShell*: umesto komande *cat* koristi se komanda *Get-Content*, a umesto komande *lpr* koristi se *Out-Printer*, dok je notacija za cevovod ista |

Mart 2020.

Copyright 2020 by Dragan Milićev

12

Znakovni tokovi

- ❖ Način na koji se programski može preusmeriti standardni ulazni ili izlazni tok procesa deteta (ovo je objašnjenje kako zapravo radi CLI kada je zadata redirekcija):

```
const char* inputFileNames = ...;  
  
int fd = open(inputFileName, O_RDONLY);  
if (fd < 0) handle_error("Cannot open input file.");  
  
int pid = fork ();  
  
if (pid == 0) {  
    dup2(fd, 0);  
    execvp(...);  
    handle_error("Cannot open exe file.");  
} ...
```

- ❖ Sistemski poziv *dup2(oldfd, newfd)* radi sledeće: kopira deskriptor fajla dat sa *oldfd* u deskriptor fajla *newfd*, tako da *newfd* zapravo ukazuje na isti fajl kao i deskriptor *oldfd*, vrednost 0 uvek označava deskriptor *stdin*, 1 označava *stdout*, a 2 *stderr*
- ❖ Na ovaj način, *stdin* procesa deteta kreiranog sa *fork* ukazuje na fajl koji je prethodno otvoren sistemskim pozivom *open* i na kog ukazuje deskriptor *fd* koji je taj sistemski poziv vratio
- ❖ Detalji mehanizma otvaranja fajlova i deskriptora fajlova biće objašnjeni u narednom delu o fajl sistemu

Mart 2020.

Copyright 2020 by Dragan Milićev

11

Razmena poruka preko priključnica

- ❖ *Priključnica (socket)* je resurs operativnog sistema koji omogućava međuprocensnu komunikaciju razmenom poruka, i to između procesa na istom ili udaljenim računarima
- ❖ *Socket API* je dostupan na praktično svim operativnim sistemima, ali i u većini programskih jezika, a implementiran je na standardan način, tako da omogućava prenos poruka preko mreže (uključujući i Internet) standardnim protokolom (TCP, ali i drugim) i između programa pisanih i na različiti jezicima ili sistemima
- ❖ Procesi na udaljenim računarima, klijentski i serverski proces, alociraju priključnicu od svog operativnog sistema, uspostave vezu (konekciju) između njih, a onda kroz njih razmenjuju poruke proizvoljnog sadržaja i dužine (niz znakova ili bajtova), sistemskim pozivima slanja i prijema poruke na svoju priključnicu

- ❖ Više detalja u predmetu OS2. Za zainteresovane npr.:

<https://www.binarytides.com/socket-programming-c-linux-tutorial/>



Mart 2020.

Copyright 2020 by Dragan Milićev

14

Štampanje

- ❖ Jedan način da se ovo uradi programski jeste korišćenje sistemskog poziva *popen* koji kreira nov proces, kao *fork*, a onda u kontekstu tog procesa izvršava program dat kao prvi argument (kao *exec*), i pravi cevovod između tekućeg procesa i tog novog procesa deteta, kao da je sve pozvano iz komandne linije. Na primer:

```
FILE* lpr = popen("lpr", "w");  
fprintf(lpr, "This text is printed.\n");  
pclose(printer);
```
- ❖ Za štampanje složenijeg sadržaja, odgovornost programa je da, tipično korišćenjem neke gotove biblioteke, pripremi sadržaj za štampu u nekom standardnom formatu (npr. PostScript ili PDF) i upiše ga u neki fajl, a posle fajl preda na štampu
- ❖ Jedan dostupan API otvorenog koda (*open source*) koji na prenosiv način omogućava štampu na različitim operativnim sistemima jeste proizvod firme Apple i naziva se CUPS. Ovaj API je složen, ali štampanje izgleda pojednostavljeno ovako:

```
#include <cups/cups.h>  
...  
int jobid = cupsPrintFile("myprinter", "filename.ps", "title", num_options, options);
```
- ❖ Operativni sistem Windows nudi usluge kojima se programski može definisati i sadržaj dokumenta za štampu (ispisivanje i iscrtavanje sadržaja stranice), kao i slanje dokumenta na štampu, upravljanje poslovima (*job*) za štamu itd. Ovi API su složeni i nazivaju se: *Print Document Package API*, *Print Spooler API*, *Print Ticket API* i *XPS Document API*

Mart 2020.

Copyright 2020 by Dragan Milićev

13

Realno vreme

- ❖ Današnji operativni sistemi opšte namene pružaju procesima i određene usluge vezane za realno vreme (mnoge od tih usluga dostupne su i kroz komande CLI). Npr. POSIX pozivi:
 - ❖ informacija o tekućem realnom vremenu:
`time_t time (time_t* tloc);` — Vraća broj sekundi koje su protekle od 1. 1. 1970. u 0 časova
postoje i bibliotečne funkcije u *<time.h>* koje vraćaju “kalendarsko” vreme, rastavljeno na datum i sat itd, a oslanjaju se na ovaj sistemski poziv: *asctime*, *ctime*, *gmtime*, *localtime*
 - ❖ suspenzija pozivajućeg procesa (“uspavljivanje”) za zadato vreme:
`unsigned int sleep (unsigned int seconds);` — Postoji i sistemski poziv sa finijom rezolucijom u nanosekundama, *nanosleep*
 - ❖ čekanje na događaje, uslove ili na sinhronizacione primitive, ali vremenski ograničeno (tzv. *timeout* kontrole, *timeout*); ako se proces ne deblokira u zadatom vremenu, biće deblokiran zbog isteka vremenske kontrole; npr. za senafore:
`int sem_timedwait (sem_t *sem, const struct timespec *abs_timeout);`
- ❖ Operativni sistemi za upotrebu u RT sistemima poseduju i druge usluge važne za ove sisteme, kao što su periodična aktivacija, kontrola prekoračenja vremenskih rokova i slično

Mart 2020.

Copyright 2020 by Dragan Milićev

16

Memorijski preslikani fajlovi

- ❖ Današnji operativni sistemi omogućavaju procesu da zatraži uslugu kojom jedan deo svog virtuelnog adresnog prostora preslika u neki fajl (*memory mapped file*):
`void* mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
`int munmap (void *addr, size_t length);`
- ❖ Sadržaj fajla koji je prethodno otvoren i čiji je deskriptor dat u *fd*, i to počev od pomeraja *offset* i dužine *length*, preslikava se u deo memorijskog prostora počev od adrese *addr*; ako je *addr* jednak *null*, kernel će sam izabrati slobodan deo virtuelnog adresnog prostora procesa; *length* mora biti umnožak veličine stranice koja se dobija pozivom *sysconf _SC_PAGE_SIZE*)
- ❖ Nakon toga, proces prosto vidi svoj deo virtuelnog adresnog prostora i pristupa mu na uobičajen način, organizujući u tom prostoru svoje podatke
- ❖ OS ove stranice, preslikane u fajl, snima u odgovarajuće delove sadržaja fajla kad ih izbacuje, a učitava u memoriju kad obrađuje stranične greške, odnosno kada proces pristupa stranicama ona nije učitana u memoriju. Sve ovo je potpuno transparentno za proces i on “ne vidi” da se sadržaj snima u fajl i učitava iz njega, jer ne vrši nikakve eksplicitne operacije
- ❖ Operativni sistemi po pravilu omogućavaju i da se ovakvi delovi memorije dele između procesa, i da na taj način komuniciraju, i to čak i bez fajla u koji su preslikani ili sa njim:
 - ❖ ako je u argumentu *flags* setovan bit *MAP_SHARED*, preslikavanje će biti deljeno između procesa koji izvrše preslikavanje u isti fajl, tako da će promene koje uradi jedan videti i oni drugi
 - ❖ ako je u argumentu *flags* setovan bit *MAP_PRIVATE*, preslikavanje će biti privatno za dati proces i on će imati svoju nezavisnu kopiju (primenjuje se tehnika kopiranja na upis), tako da promene koje on napravi drugi procesi neće videti, ali se neće ni upisivati u fajl

Mart 2020.

Copyright 2020 by Dragan Milićev

15

Zvuk

- ❖ I za usluge reprodukcije i snimanja zvuka operativni sistemi pružaju različite nivoe usluga
- ❖ U različitim sistemima postoje različite vrste podrške sa ove usluge, ili kao systemske usluge, ili kao API koji implementiraju biblioteke, a koje onda koriste systemske usluge niskog nivoa
- ❖ Na primer, Windows ima funkciju *PlaySound* koja reprodukuje zvuk zapisan u zadatom fajlu
- ❖ Na starijim verzijama sistema Linux i na nekim drugim sistemima nalik sistemu Unix, uređaj za reprodukciju zvuka postoji kao */dev/dsp*, i na njega se može poslati sadržaj fajla sa zvučnim zapisom
- ❖ Novije verzije Linux kernela koriste tzv. ALSA (*Advanced Linux Sound Architecture*) modul za pristup do usluga zvučnih kartica, a za procese postoji biblioteka koja implementira ALSA API

Mart 2020.

Copyright 2020 by Dragan Milićev

18

Slika

- ❖ Nekadašnji monitori bili su znakovno orijentisani, pa im se pristupalo kao bilo kom izlaznom znakovnom toku. Današnji monitori su rasterski, jer poseduju sliku definisanu matricom piksela, ali i oni ponekad mogu da podržavaju tekstualni režim rada u kom mogu primiti jednostavne tokove znakova i ispisivati ih linijski
- ❖ Neki operativni sistemi, poput sistema Windows ili Mac OS X, imaju GUI kao sastavni deo kernela, pa zato korisničkim procesima pružaju i usluge (kroz složen API) upravljanja resursima grafičkog interfejsa kojim upravlja kernel: prozorima, GUI kontrolama, iscrtavanjem znakova i grafičke po prozoru procesa itd.
- ❖ Kod drugih sistema, poput sistema Linux, GUI nije deo kernela, već se pravi kao izdvojena školjka
- ❖ Da bi proces, pa i školjka, pristupio ekranu, potrebno je da koristi neku biblioteku (npr: Xlib, Wayland) ili okruženje za programiranje (*framework*, npr: GTK+ ili Qt) koja implementira GUI, a ta implementacija onda koristi usluge operativnog sistema na niskom nivou
- ❖ Da bi proces, odnosno ovakva školjka pristupila uređaju za video prikaz, OS obezbeđuje mehanizam da proces pristupi posebnom delu operativne memorije koji se naziva *video bufer* (*video buffer*). Ovaj deo operativne memorije koristi i video adapter (uređaj za generisanje izlaznog video signala za ekran), a OS može obezbediti uslugu da deo virtuelnog adresnog prostora procesa preslika direktno u taj deo operativne memorije. Ovaj deo memorije koristi se kao bafer sa bitmapom jednog kadra slike (*frame*) koju onda video adapter (hardver) koristi za pretvaranje u izlazni video signal
- ❖ Na sistemu Linux, ovaj deo memorije naziva se *framebuffer*. Ovaj bafer vidi se i kao generički uređaj */dev/fb0*

Mart 2020.

Copyright 2020 by Dragan Milićev

17

Draiveri uređaja

- ❖ Ulazno-izlazni uređaji su izuzetno raznoliki po svim navedenim karakteristikama. Osim toga, proizvođači hardvera stalno osmišljavaju i proizvode nove uređaje
- ❖ Kako korisničke procese učiniti nezavisnim od varijeteta i stalne promjenljivosti uređaja? Upravo to i jeste zadatak operativnog sistema - da izoluje procese od hardvera, pružajući im apstraktan i standardizovan API
- ❖ Međutim, kako sam OS implementirati tako da podrži sve uređaje, pa i one koji se pojavljuju nakon što je OS već napravljen?
- ❖ Kao i kod svih drugih tehničkih i složenih softverskih sistema, potrebno je poštovati sve principe softverskog inženjerstva:
 - ❖ standardizacija: propisivanje i prihvatanje standarda koje proizvođači uređaja treba da poštuju, i na nivou elektrotehničkih parametara, i na nivou softverskih interfejsa, kako bi uređaji bili kompatibilni sa računarima
 - ❖ apstrakcija: zanemarivanje različitosti, a isticanje zajedničkih osobina različitih entiteta, kako bi se oni klasifikovali u grupe srodnih i posmatrali na isti način
 - ❖ modularizacija: jasno razdvajanje odgovornosti delova softvera kako bi njihove interakcije bile što jednostavnije
 - ❖ enkapsulacija: jasno definisanje interfejsa modula preko kog mu se pristupa, dok mu je implementacija sakrivena
- ❖ Osim toga, zbog prirode ovog podсистема, veoma često se arhitektura U/I podсистема organizuje kao slojevita: svaki modul koristi usluge modula "ispod" sebe, oslanjajući se na njihov interfejs, podiže nivo apstrakcije i pruža svoje usluge slojevima iznad sebe

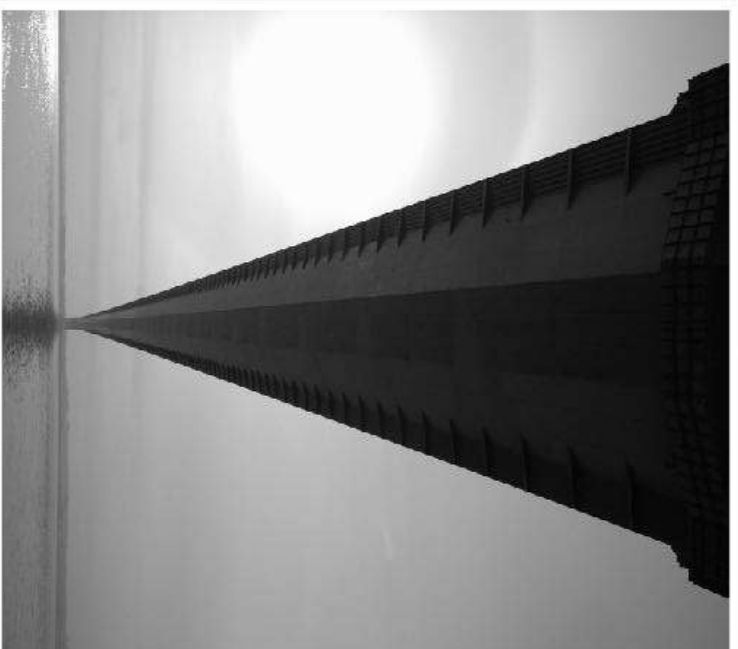
Mart 2020.

Copyright 2020 by Dragan Milićev

20

Glava 11: Implementacija U/I podсистема

- ❖ Draiveri uređaja
- ❖ Tehnike pristupa uređajima
- ❖ Upravljanje diskovima
- ❖ Keširanje
- ❖ Baferisanje
- ❖ Spuling
- ❖ Realno vreme
- ❖ Zaštita
- ❖ Performanse



Mart 2020.

Copyright 2020 by Dragan Milićev

19

Draiveri uređaja

- ❖ Kada se na računar priključi nov uređaj, ukoliko on zadovoljava generičke, standardne zahteve, OS već ima gotov standardan, generički draiver koji koristi za komunikaciju sa tim uređajem
- ❖ Ukoliko je uređaj krajnje specifičan, nestandardan, u smislu da mu generički draiver ne odgovara, proizvođač uz uređaj mora da isporučiti i draiver za dati OS. Svaki OS poseduje mehanizam kojim se nov draiver instalira u sistem. Taj draiver svakako mora da zadovolji interfejs za datu klasu (tip) uređaja koji OS očekuje
- ❖ U bilo kom od ovih slučajeva, kernel registruje taj uređaj kao dostupan, smatra ga uređajem koji pripada odgovarajućoj klasi, pa prema tome zadovoljava odgovarajući interfejs, i usmerava operacije koje ta klasa uređaja podržava na draiver za taj uređaj
- ❖ Na ovaj način se obezbeđuje da OS bude “otporan” na raznolikost i stalnu pojavu novih uređaja i da ih prihvata i koristi
- ❖ Nažalost, standardi i interfejsi za draivere važe samo za određeni OS, pa draiver nije prenosiv na drugi OS - dati uređaj mora imati draivere za svaki OS na kom se koristi

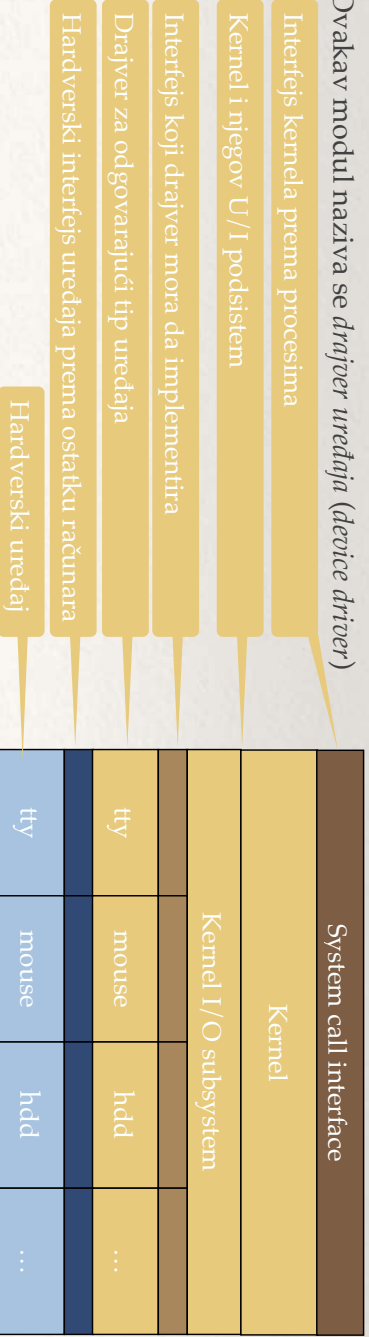
Mart 2020.

Copyright 2020 by Dragan Milićev

Draiveri uređaja

- ❖ Da bi upošle bio priključen na računar, uređaj mora da zadovolji odgovarajuće tehničke standarde, kao i da implementira odgovarajući hardverski interfejs koji omogućava njegovo priključenje, ali često i odgovarajući softverski protokol po kom računar (procesor) sa njim može da razmenjuje podatke
- ❖ Svaki uređaj pripada određenoj klasi (tipu) uređaja koju računar, ali i dati OS prepoznaje, i tako se računar i “predstavlja”: kao tastatura, miš, disk (blokovski uređaj za skladištenje podataka) itd.
- ❖ Da bi ostatak kernela bio nezavisan od uređaja, ali i da bi kernel bio spreman da prihvati stalno nove uređaje, za komunikaciju sa svakom klasom klasom uređaja postoji poseban deo, modul kernela, koji implementira odgovarajući interfejs prema ostatku kernela za datu klasu uređaja - skup operacija koje kernel poziva za ovu klasu uređaja (npr. prenos jednog znaka, prenos bloka podataka itd.)
- ❖ Taj modul zadužen je za neposrednu komunikaciju sa hardverskim uređajem, na najnižem nivou

❖ Ovakav modul naziva se *draiver uređaja (device driver)*



Mart 2020.

Copyright 2020 by Dragan Milićev

Tehnike pristupa uređajima

- ❖ Današnji U/I uređaji i protokoli komunikacije sa njima su izuzetno, izuzetno složeni, jer su razvijani i unapređivani decenijama, a njima se bave brojni veliki stručni timovi posvećeni i specijalizovani samo za određene vrste uređaja
- ❖ Zato postoje brojni standardi, kao i još brojnije implementacije tih standarda za povezivanje i komunikaciju sa U/I uređajima. Da bi proučio i specijalizovao se za svakog od njih pojedinac treba da posveti dobar deo svoje stručne karijere
- ❖ Međutim, neki fundamentalni principi su jednostavni i isti kakvi su oduvek i bili
- ❖ Ovde će oni biti ukratko prikazani na jednom jednostavnom, generičkom modelu U/I uređaja, kao i na zadatku da se prenese blok podataka određene veličine iz memorije na uređaj (izlazna operacija) ili obratno (ulazna operacija)
- ❖ Sama hardverska implementacija uređaja je, naravno, specifična, ali ovde nije od interesa. Od interesa je samo hardverski interfejs tog uređaja, tzv. kontroler uređaja (*device controller*)

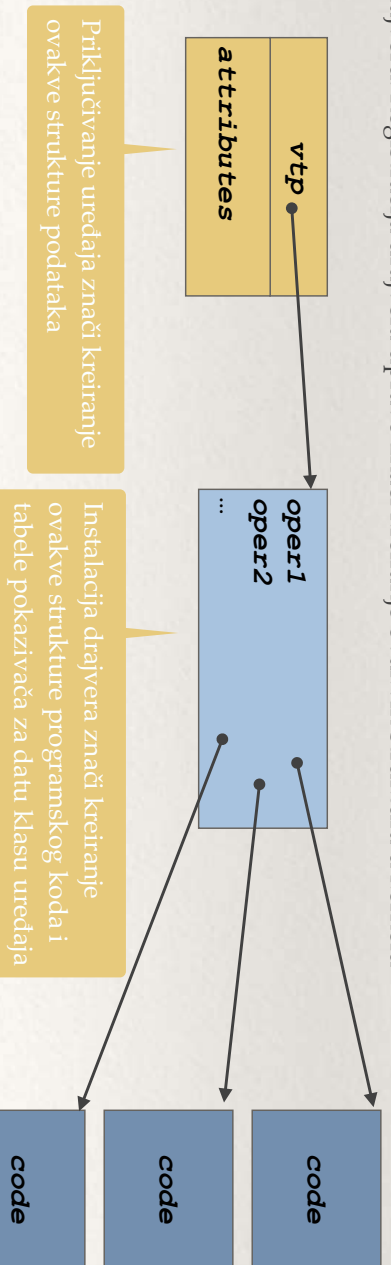
Mart 2020.

Copyright 2020 by Dragan Milićev

24

Draiveri uređaja

- ❖ Posmatrano objektno orijentisano, U/I podsistem kernela zapravo vidi uređaje kao objekte koji zadovoljavaju neki interfejs, definisan apstraktnom klasom koja ima apstraktne operacije, i poziva njihove polimorfne operacije
- ❖ Konkretno izvedene klase implementiraju ove apstraktne operacije
- ❖ Kako su sistemi tipično pravljeni na jeziku C, ovaj polimorfizam se implementira kao što je već objašnjeno, dinamičkim vezivanjem
- ❖ Draiver je zapravo softverski modul koji obezbeđuje implementaciju skupa potprograma (polimorfnih operacija) koje kernel očekuje od date klase uređaja. Konkretni uređaj se u kernelu registruje kao struktura podataka koja sadrži attribute uređaja, kao i pokazivač na tabelu pokazivača na implementacije funkcija u draiveru. Registracija draivera upravo znači kreiranje ovakvih struktura u kernelu



Mart 2020.

Copyright 2020 by Dragan Milićev

23

Tehnike pristupa uređajima

- ❖ Procesor može da pristupa ovim registrima tako što izvršava instrukcije koje čitaju vrednost ili upisuju vrednost na adresu na koju je "vezan" dati registar, odnosno na koju se on "odaziva". To znači da procesor, tokom izvršavanja date instrukcije, na adresnu magistralu računara postavlja odgovarajuću adresu, vrednost sa adrese magistrale prolazi kroz adresne dekodere, čija samo jedna linija, ona koja odgovara datoj adresi, postaje aktivna. Ta linija dolazi do odgovarajućeg registra i:
 - ❖ zajedno sa upravljačkim signalom *read* na magistrali, koji procesor postavlja tokom ciklusa čitanja, prolazi kroz logičko i kolo čiji izlaz otvara trostatičke baferne vezane na izlazima tog registra prema magistrali podataka; tako se vrednost adresiranog registra postavlja na magistralu podataka, odakle je procesor upisuje u neki svoj prihvatni registar
 - ❖ zajedno sa upravljačkim signalom *write* na magistrali, koji procesor postavlja tokom ciklusa upisa, prolazi kroz logičko i kolo čiji izlaz aktivira upravljački ulaz *load* za upis vrednosti u adresirani registar; ta vrednost koja se upisuje stiže sa magistrale podataka na koju ju je postavio procesor tokom ciklusa upisa
- ❖ Pretpostavljamo da procesor izvršava deo programa, tj. instrukcije kojima treba da prebaci niz podataka (reči ili bajtova, u zavisnosti od širine magistrale i registra podataka), iz memorije na uređaj (ako je u pitanju izlazna operacija) ili obratno (ako je u pitanju ulazna operacija)

Mart 2020.

Copyright 2020 by Dragan Milićev



26

Tehnike pristupa uređajima

- ❖ Ovaj kontroler obezbeđuje interfejs, sprengu sa ostatkom računara tako što poseduje određene registre koji su na odgovarajući način povezani na magistralu računara, tako da im procesor može pristupiti adresirajući ih preko adresa na koje se "odazivaju"
- ❖ Ovi registri pripadaju jednoj od tri karakteristične grupe, tipa:
 - ❖ *Upravljački* registar (ili više njih, *control register*): tipično vezan na magistralu računara tako da procesor u njega može samo upisivati, dok vrednost bita ovog registra tumači upravljačka jedinica kontrolera uređaja; biti (razredi) tog registra imaju odgovarajuće značenje za dati uređaj, ali u principu, preko njihovih vrednosti procesor, odnosno program koji procesor izvršava, može da zada režim rada uređaja, da mu izda komandu, parametre komande, pokrene operaciju itd. Na primer, bit *start* pokreće operaciju na uređaju (ovo je samo ilustrativan i generički primer)
 - ❖ *Statusni* registar (ili više njih, *status register*): tipično vezan na magistralu računara tako da procesor iz njega može samo čitati, dok vrednost u ovaj registar upisuje upravljačka jedinica kontrolera uređaja; biti (razredi) tog registra imaju odgovarajuće značenje za dati uređaj, ali u principu, preko njihovih vrednosti uređaj izvršava o statusu izvršene operacije, postojanju greške, rezultatu itd.
 - ❖ *Registar podataka* (ili više njih, *data register*): vezan na magistralu računara tako da procesor iz njega može čitati, ako se radi o ulaznom uređaju, i/ili upisivati, ako se radi o izlaznom uređaju; služi za prenos podataka iz uređaja i ka njemu



Mart 2020.

Copyright 2020 by Dragan Milićev

25

Tehnike pristupa uređajima

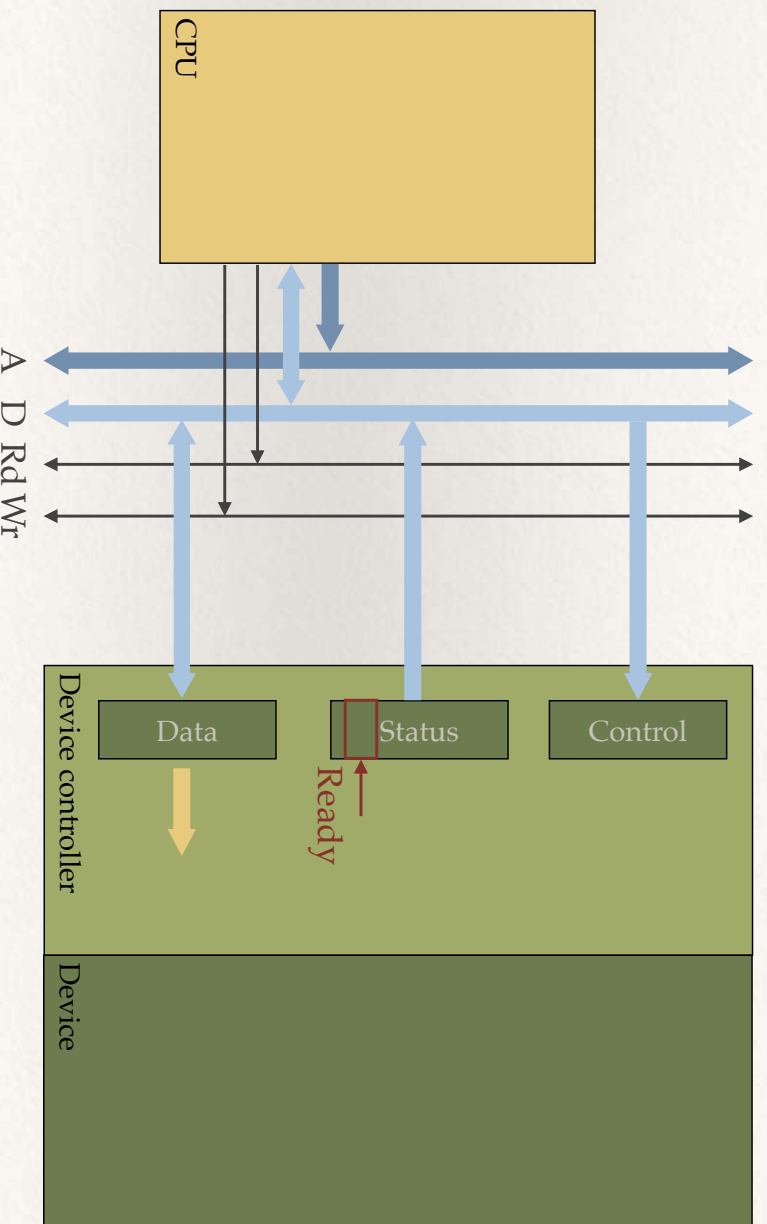
- ❖ Procesor izvršava jednu instrukciju kojom prenosi jedan podatak preko registra za podatke kontrolera uređaja:
 - ❖ ako se radi o ulaznoj operaciji, vrši operaciju čitanja iz ovog registra
 - ❖ ako se radi o izlaznoj operaciji, vrši operaciju upisa u ovaj registar
- ❖ Kada to uradi, procesor bi mogao da izvrši narednu instrukciju kojom prenosi naredni podatak na isti način. Da li to sme da uradi odmah? Ne, jer kontroler periferije možda još uvek nije završio potreban prenos:
 - ❖ ako se radi o ulaznoj operaciji, u registru za podatke je još uvek stari podatak, nov još nije stigao, tj. kontroler ga još nije upisao
 - ❖ ako se radi o izlaznoj operaciji, kontroler još uvek nije završio prenos podatka u registru za podatke, upis nove vrednosti bi poremetio taj prenos prethodnog podatka
- ❖ Zato procesor mora da sačeka da kontroler uređaja bude *spreman* za nov prenos - potrebna je sinhronizacija ova dva uređaja!
- ❖ Da bi signalizirao spremnost za nov prenos, kontroler koristi određeni bit u statusnom registru, bit spremnosti (*ready*):
 - ❖ kada procesor upiše novu vrednost u registar podataka, ili pročita vrednost iz njega, ovaj bit spremnosti se briše (isti signal za upis u registar za podatke ili otvaranje njegovih trostatičkih bafera prema magistrali koristi se za reset bita spremnosti)
 - ❖ procesor ne sme da izvrši sledeći prenos pre nego što ovaj bit bude postavljen
- ❖ Ovakva sinhronizacija između hardverskih uređaja naziva se *rukovanje (handshaking)*; zapravo se radi o klasičnoj uslovnoj sinhronizaciji, ali sada na hardverskom nivou

Mart 2020.

Copyright 2020 by Dragan Milićev

28

Tehnike pristupa uređajima



Mart 2020.

Copyright 2020 by Dragan Milićev

27

Tehnike pristupa uređajima

```
typedef volatile unsigned REG;
```

```
REG* r_control = (REG*)0x...;  
REG* r_status = (REG*)0x...;  
REG* r_data = (REG*)0x...;
```

```
struct IORquest {  
    void* buffer;  
    size_t len;  
    void (*signal)();  
};
```

...

```
const REG C_START = 0x...;  
const REG C_STOP = 0x...;  
const REG C_READY = 0x...;
```

Maska za ispitivanje bita spremnosti: binarna vrednost koja ima 1 u razredu u kom je bit spremnosti *ready*, a 0 u svim ostalim

Mart 2020.

Copyright 2020 by Dragan Milićev

30

Tehnike pristupa uređajima

- ❖ Kako procesor, odnosno program koji on izvršava, da zna da je bit spremnosti postavljen? Dva su načina:

- ❖ da čita vrednost statusnog registra kontrolera, izvršavajući instrukciju čitanja sa odgovarajuće adrese i ispituje bit spremnosti, i ponavlja to, čekajući da bit spremnosti bude postavljen - uposlono čekanje (*busy waiting*) ili *proizivnje* (*polling*)

- ❖ da izlaz razreda bita spremnosti statusnog registra bude vezan na neki ulaz zahteva za prekid procesora, tako da procesor može da izvršava neki drugi posao, a prekid signalizira spremnost kontrolera za novu operaciju

- ❖ Kod obe ove tehnike procesor je taj koji vrši prenos podatka iz memorije u registar podataka ili obratno, izvršavajući instrukcije prenosa podataka. Zato se obe tehnike nazivaju *programirani ulaz-izlaz* (*programmed input/output*)

- ❖ Naravno, nedostatak tehnike prozivjanja jeste bespotrebno trošenje procesorskog vremena na čekanje, osim ako ono ne traje sasvim kratko

- ❖ U narednim primerima biće pokazan način za prenos niza podataka zadate dužine programiranim ulazom-izlazom. Prepostavlja se da je jedan zahtev za prenosom dat u strukturi tipa *IORquest*; ovi zahtevi mogu biti složeni u red čekanja, npr. ulančavanjem u listu

- ❖ Prepostavlja se da polje *signal* u ovoj strukturi određuje šta treba uraditi kada se zahtev opsluži, npr. tako što ukazuje na funkciju koju treba pozvati, a može predstavljati i semafor koji treba signalizirati jer na njemu čeka proces (ili nit) koji je tražio tu operaciju i slično, u zavisnosti od konstrukcije ovog dela kernela

Mart 2020.

Copyright 2020 by Dragan Milićev

29

Tehnike pristupa uređajima

- ❖ Primer dela programa koji vrši prenos niza podataka programiranim ulazom-izlazom korišćenjem prekida:

```
REG* pending_req;
size_t pending_cur;

void transfer (IORequest* req) {
    REG* pending_req = req;
    pending_cur = 0;
    *r_control = C_START;
}

interrupt void device_ready () {
    REG* buf = (REG*)(pending_req->buffer);

    *r_data = buf[pending_cur++];

    if (pending_cur==pending_req->len) {
        *r_control = C_STOP;
        req->signal();
        pending_req = 0;
    }
}
```

Mart 2020.

Copyright 2020 by Dragan Milićev

32

Tehnike pristupa uređajima

- ❖ Primer dela programa koji vrši prenos niza podataka programiranim ulazom-izlazom prozivanjem:

```
void transfer (IORequest* req) {
    REG* buf = (REG*)(req->buffer);

    *r_control = C_START;

    for (size_t i=0; i<req->len; i++) {
        while (!((*r_status) & C_READY));
        *r_data = buf[i];
    }

    *r_control = C_STOP;
    req->signal();
}
```

Mart 2020.

Copyright 2020 by Dragan Milićev

31

Tehnike pristupa uređajima

- ❖ Ove radnje jednostavnog prenosa podatka iz memorije u registar podataka ili obratno, uz kontrolu petlje, suviše su banalni zadaci za tako sofisticiran uređaj kakav je procesor koji može da izvršava mnogo korisnije i složenije poslove
- ❖ Zbog toga je osmišljen poseban hardverski uređaj, tzv. *kontroler za direktan pristup memoriji* (*direct memory access controller*, *DMA controller*) koji je specijalizovan samo za prenos jednog niza (bloka) podataka iz memorije na uređaj ili obratno
- ❖ Procesor je taj koji zadaje zahtev DMA kontroleru, kao i bilo kom drugom uređaju: zadaje mu parametre zahteva upisom u odgovarajuće registre DMA kontrolera koji su namenjeni za prihvatanje tih parametara
- ❖ Parametri zahteva su tipično adresa bloka (bafera) u memoriji, dužina tog niza, režim rada i slično
- ❖ Nakon što zada parametre i pokrene operaciju, procesor može da izvršava neki drugi, nezavisan tok kontrole
- ❖ DMA za to vreme obavlja sinhronizaciju sa uređajem, bilo neposrednim "rukovanjem" (direktnom razmenom signala) sa kontrolerom uređaja ili čitanjem statusnog registra i čekanjem na bit spremnosti, kao i sam prenos podatka (čitanje ili upis u memoriju i registar kontrolera)
- ❖ Zbog ovoga i DMA kontroler obavlja cikluse na magistrali, pa je neophodna hardverska podrška za međusobno isključenje pristupa magistrali između procesora i DMA kontrolera
- ❖ Sve ovo je potpuno nevidljivo za procesor; procesoru, tj. njegovom programu je bitno samo da zna da je ceo zahtev obrađen, odnosno da je prenos celog bloka završen
- ❖ Zato DMA kontroler po pravilu generiše zahtev za prekid kada završi celu operaciju

Mart 2020.

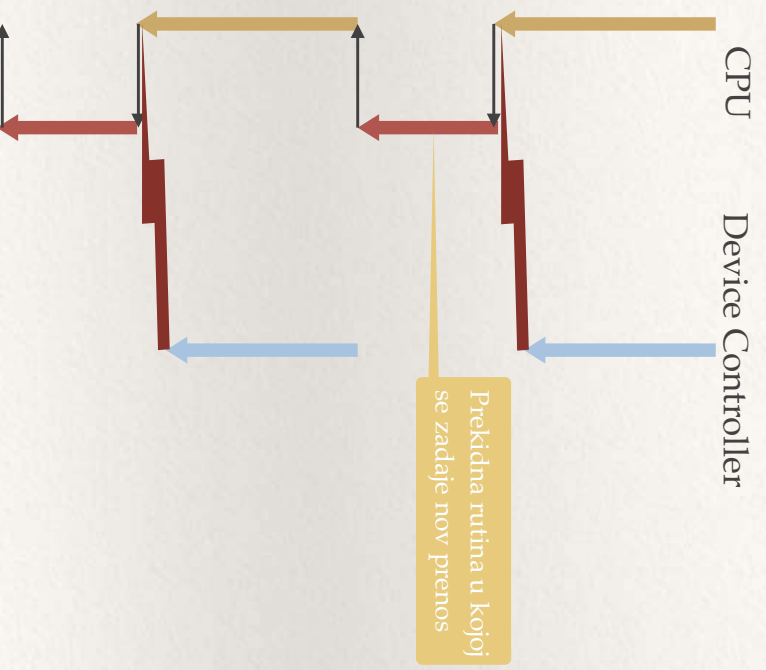
Copyright 2020 by Dragan Milićev

34

Tehnike pristupa uređajima

- ❖ Kada se koristi prekid, procesor, koji izvršava neki nezavisan tok kontrole, i kontroler uređaja obavljaju operacije paralelno, istovremeno; kada je uređaj spreman za nov prenos, generiše prekid procesoru i procesor obavlja prenos sledećeg podatka

- ❖ Zajedničko za obe tehnike jeste to što procesor, izvršavanjem instrukcija, vrši prenos iz memorije u registar podataka kontrolera ili obratno, kao i to što se on bavi kontrolom petlje (brojanje, prelazak na sledeći podatak, ispitivanje završetka petlje)



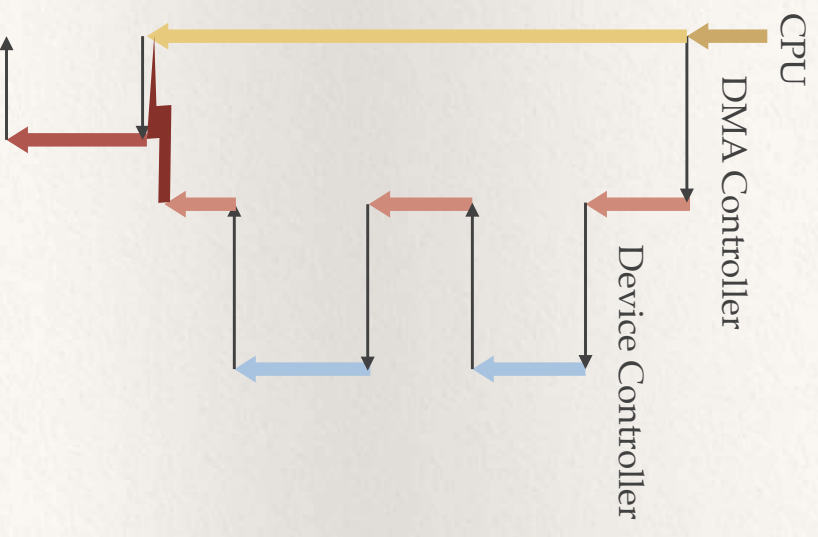
Mart 2020.

Copyright 2020 by Dragan Milićev

33

Tehnike pristupa uređajima

- ❖ Sada procesor samo inicijalizuje prenos i dalje obavlja neki nezavisan tok kontrole; DMA kontroler preuzima ceo posao sinhronizacije sa uređajem i prenosa podataka
- ❖ Računari odavno imaju DMA kontrolere kao obavezan sastavni deo svoje arhitekture, i to po pravilu više njih. Moguće su i varijante u kojoj jedan DMA kontroler, sa jedinstvenim interfejsom prema procesoru, ima više kanala, tj. može da obavlja više uporednih prenosa nezavisno, i svaki se može zadati kroz dati interfejs
- ❖ Prenos podataka korišćenjem DMA kontrolera se veoma intenzivno koristi u današnjim U/I podsystemima, a po pravilu je neizostavan kod prenosa podataka sa blokovskim uređajima



Mart 2020.

Copyright 2020 by Dragan Milićev

36

Tehnike pristupa uređajima

- ❖ Primer dela programa koji vrši prenos niza podataka korišćenjem DMA kontrolera:

```
REG* dma_control = (REG*)0x...;
REG* dma_addr = (REG*)0x...;
REG* dma_size = (REG*)0x...;

REG* pending_req;

void transfer (IORequest* req) {
    REG* pending_req = req;

    *dma_addr = req->buffer;
    *dma_size = req->len;
    *dma_control = C_START;
}

interrupt void dma_transfer_complete () {
    *dma_control = C_STOP;
    pending_req->signal();
    pending_req = 0;
}
```

Mart 2020.

Copyright 2020 by Dragan Milićev

35

Upravljanje diskovima

- ❖ Diskovima (*hard drive*) se ovde nazivaju svi blokovski ulazno-izlazni uređaji sa direktnim pristupom, priključeni na računar, koji služe za skladištenje podataka
- ❖ Ovi uređaji su posebno važni ne samo zbog toga što se na njih smešta sam operativni sistem i korisnički fajlovi (i programi i podaci), već i zbog toga što sam OS koristi diskove za smeštanje izbačenih stranica procesa
- ❖ Zbog toga je podsystem upravljanja diskovima, njegova organizacija i performanse, od posebnog značaja za funkcionisanje celog sistema
- ❖ Osnovne zajedničke karakteristike diskova:
 - ❖ to su ulazno-izlazni uređaji: omogućavaju i čitanje i upis podataka
 - ❖ sadržaj je *perzistentan* (*persistent*): jednom upisan sadržaj ostaje trajno sačuvan do ponovnog upisa na isto mesto, bez obzira na prestanak napajanja (isključivanje) računara (za razliku od operativne memorije koja nije perzistentna, *non-persistent*, *volatile*)
 - ❖ blokovski su orijentisani (*block-oriented*): omogućavaju prenos isključivo celih blokova fiksne veličine (npr. 512 bajtova); da bi se izmenio jedan jedini bajt, potrebno je učitati ceo blok sa diska u memoriju, izmeniti šta je potrebno u memoriji, a onda upisati ceo blok na disk; svaki blok na disku ima svoju *logičku adresu* - redni broj tog bloka na disku kojim se blok adresira
 - ❖ omogućavaju *direktan pristup* (*direct access*): blokovima se može pristupiti u proizvoljnom redosledu, nezavisno od toga na koji način su fizički smešteni i pozicionirani na disku

Mart 2020.

Copyright 2020 by Dragan Milićev

38

Tehnike pristupa uređajima

- ❖ Opisane tehnike koriste se na najnižem nivou apstrakcije, u najnižim slojevima kernela, odnosno drajvera uređaja, onim koji neposredno pristupaju uređajima i obavljaju stvarni prenos
- ❖ Kada se registruju, drajveri uređaja mogu tražiti od kernela korišćenje DMA kanala i / ili ulaze zahteva za prekid u vektor tabeli. Kernel zato vodi evidenciju o zauzetim i slobodnim DMA kanalima i ulazima u vektor tabelu i dodeljuje drajveru onaj koji zauzme za njega; drajver onda može da konfiguriše kontrolere prekida ili uređaja, upisujući u njihove registre ove parametre koji se onda koriste kod U / I operacija
- ❖ Za uređaje koji su obavezni sastavni deo arhitekture određenog računara (npr. PC arhitektura), rutine koje obavljaju ovakve elementarne operacije sa uređajima, na najnižem nivou, deo su koda koji se nalazi u ROM-u računara i koristi ih kernel, odnosno drajveri uređaja - tzv. *BIOS* (*Basic Input/Output System*) rutine
- ❖ Naravno, moguće su veoma različite varijante navedenih pristupa, kao i druge tehnike, koje jako zavise od konkretnih uređaja

Mart 2020.

Copyright 2020 by Dragan Milićev

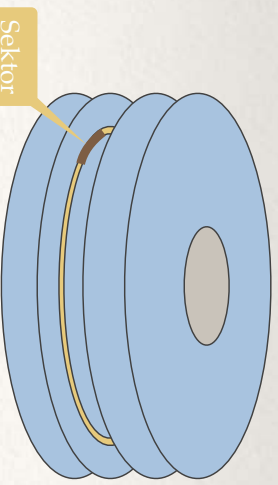
37

Upravljanje diskovima

- ❖ Uređaj magnetnog diska sadrži više koaksijalnih tankih kružnih ploča - diskova poređanih na istu osovinu koja se neprekidno okreće velikom brzinom u sredini koja je skoro vakuum (izuzetno male gustine, zbog što manjeg otpora)
- ❖ Svaki disk na svojoj površini ima magnetni sloj sa veoma finom granulacijom koji se može različito polarizovati (namagnetsati), čime se zapisuju biti
- ❖ Zapis je organizovan u *trake* ili *staze* (*track*) - koncentrične prstenove koji sadrže linearan zapis bita
- ❖ Svaka traka je podeljena na *sektore* koji sadrže zapise blokova podataka; jedna operacija viši se na jednom sektoru, odnosno bloku
- ❖ Čitanje i upis vrše magnetne glave, po jedna iznad svake površine diska ("lebdi" na vrlo malom odstojanju od površine), koje su pozicionirane na mehaničkoj ruci u obliku "češlja"; ruku pomera koračni motor (*stepper motor*) radijalno, od periferije diskova ka osovini i obratno
- ❖ Da bi određeni blok bio pročitlan ili upisan:
 - ❖ ruka mora da se pomeri tako da se pozicionira iznad trake na kojoj se nalazi traženi blok
 - ❖ diskovi moraju da se obrnu toliko da traženi blok dođe ispod glave

Mart 2020.

Copyright 2020 by Dragan Miličev



40

Izgled magnetnog diska (Wikipedia)



Upravljanje diskovima

- ❖ Prema tome, osnovne operacije sa ovakvim uređajima su samo dve (to su operacije drajvera):
 - ❖ pročitaj sadržaj bloka sa datom logičkom adresom i učitaj ga na određeno mesto u memoriji; na primer: `int readBlock (BlockNo block, void* buffer);`
 - ❖ sadržaj bloka sa određenog mesta u memoriji upiši u blok na disku sa zadatom logičkom adresom; na primer: `int writeBlock (BlockNo block, void* buffer);`
- ❖ Prema tehnološkoj izvedbi, diskovi mogu biti:
 - ❖ *magnetni* (*magnetic disk*, *hard disk drive*): zapis je magnetni, na kružnim rotirajućim diskovima kojima se pristupa mehaničkim putem
 - ❖ *elektronski*, *čvrsta memorija* (*flash*, *solid state drive*, *SSD*): memorija je elektronska, nema mehaničkih delova niti mehaničkog kretanja
- ❖ Bez obzira na tehnologiju, obe vrste diskova:
 - ❖ posmatraju se na isti način od strane operativnog sistema, kao blokovski orijentisani, perzistentni, ulazno-izlazni uređaji, i pristupa im se preko istog interfejsa
 - ❖ imaju vreme pristupa - vreme koje je potrebno da se pristupi jednom bloku podataka kada se adresira, kao i brzinu transfera - brzinu prenosa podataka; vreme pristupa je svakako značajno veće od vremena pristupa operativnoj memoriji, ali i mnogo kraće od ostalih ulazno-izlaznih uređaja
- ❖ SSD uređaji imaju uniformno vreme pristupa - ne zavisi od bloka koji se adresira, a koje je značajno kraće od vremena pristupa magnetnih diskova; međutim, oni imaju određena tehnološka ograničenja, poput ograničenog broja upisa u istu lokaciju
- ❖ Međutim, magnetni diskovi još uvek imaju značajno veće kapacitete (stotine GB ili terabajti), kao i mnogo nižu cenu po jedinici količine podataka. Zbog toga su magnetni diskovi i dalje u veoma širokoj upotrebi i značajni za računarski sistem

Mart 2020.

Copyright 2020 by Dragan Miličev

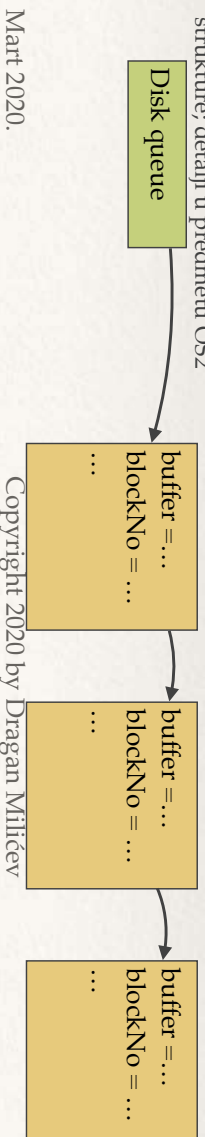
39

Upravljanje diskovima

- ❖ Procesi postavljaju različite zahteve za uslugama operativnog sistema koji se mogu ispoljiti kao zahtevi za pristup disku, tipično kroz fajl sistem, ali i drugačije (npr. zbog zamene stranica); ti zahtevi se na najnižem nivou preslikavaju u zahteve za učitavanje ili upis pojedinačnih blokova na disku
- ❖ Kada postavi takav zahtev koji ne može odmah da se opsluži, proces mora da se suspenduje
- ❖ Zahtev se smešta u red čekanja, kao zapis u strukturi koja sadrži sve elemente zahteva
- ❖ Poseban tok kontrole, interna nit kernela, može da uzima zahteve iz reda čekanja, jedan po jedan, i upućuje disku, odnosno poziva operacije drajvera koji organizuje prenos pomoću DMA kontrolera
- ❖ Kada je zahtev opslužen, odnosno cela usluga koju je proces tražio završena, proces se može deblokirati i nastaviti izvršavanje
- ❖ Kako vreme pristupa blokovima na disku nije uniformno, nije svejedno kojim redom se oni opslužuju; npr. ako bi se zahtevi opsluživali po redosledu po kom su postavljeni (FIFO), ako je u redu zahteva jedan koji se odnosi na blok na spoljašnjem cilindru, pa iza njega zahtev koji se odnosi na blok koji je na unutrašnjem cilindru (blizu osovine), pa onda ponovo neki periferni, glava bi se mnogo pomerala i vreme opsluživanja, pa time i vreme čekanja zahteva u redu bilo nepotrebno duže nego da se oni opslužuju nekim drugim redosledom

- ❖ Zato je bitan *algoritm raspoređivanja zahteva za diskom (disk scheduling algorithm)* i operativni sistemi primenjuju različite algoritme - detalji u predmetu OS2

- ❖ Postoje i konfiguracije više urešaja sa diskovima koje omogućavaju bolje performanse (zbog paralelizacije operacija na više diskova) i povećanu otpornost na otkaze (zbog redundantnih zapisa), koje su posebno važne za serverske sisteme - tzv. RAID strukture; detalji u predmetu OS2



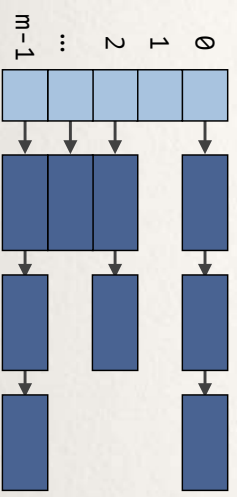
42

Upravljanje diskovima

- ❖ Zbog ovoga su blokovi logički organizovani u tzv. *cilindre (cylinder)* - skupove traka na istom poluprečniku ali na različitim diskovima; pristup blokovima na istom cilindru ne zahteva pomeranje glava, pa je pristup do njih brži, i zato se oni numerišu kao logički susedni
- ❖ Prema tome, vreme pristupa jednom bloku uključuje:
 - ❖ vreme pozicioniranja glava, koje zavisi od toga gde su glave bile (na kom cilindru) i gde se nalazi traženi blok (na kom cilindru); to znači da ovo vreme zavisi od puta koji glave treba da pređu, odnosno od međusobne udaljenosti cilindara na koje se odnose susedni zahtevi koji se opslužuju, što može biti dominantan udeo u vremenu pristupa
 - ❖ vreme koje je potrebno da prođe da se, zbog stalne rotacije, traženi blok postavi ispod glava; u najboljem slučaju, blok će se odmah pojaviti ispod glave; u najgorem, on je tek prošao i potrebna je skoro cela jedna rotacija; u srednjem, potrebno je vreme polurotacije
- ❖ Zbog ovoga, vreme pristupa bloku na magnetnom disku *nije uniformno*, jer jako zavisi od redosleda pristupa blokovima, odnosno dominantno od rastojanja cilindara na kojima se ti blokovi nalaze
- ❖ Radom celog uređaja upravlja *kontroler diska (disk controller)*, hardverski sklop na samom uređaju koju opslužuje zahteve, upravlja koračnim motorom i glavama, preslikava logičke adrese blokova u njihove fizičke pozicije, ali i kešira sadržaj blokova, detektuje i koriguje greške u zapisu, što je sve transparentno za ostatak računara - detalji u predmetu OS2

Keširanje

- ❖ Prvo pitanje jeste organizacija strukture keša, odnosno načina na koji se skladište informacije o tome koji blokovi diska se nalaze u kešu
- ❖ Jedna od varijanti jeste organizacija u formi *heš tabele* (*hash table*):
 - ❖ tabela ima m ulaza ($m \geq 1$)
 - ❖ sadržaj bloka broj b skladišti se u ulazu $hash(b)$, gde je $hash$ neka funkcija koja dobro "rasipa" ključeve (brojeve blokova), npr. jednostavan ostatak pri deljenju sa m ($b \bmod m$)
 - ❖ ako se više blokova preslikava i skladišti u istom ulazu, oni se redaju u red (npr. listu), potencijalno ograničenog kapaciteta n ($n \geq 1$); druga varijanta je ta da ceo keš, a ne svaki ulaz pojedinačno, bude ograničenog kapaciteta od n blokova



Mart 2020.

Copyright 2020 by Dragan Milićev

44

Keširanje

- ❖ Kako bi operacije sa diskovima učinio efikasnijim, OS obavezno organizuje *keš* (*cache*) blokova učitanih sa diska u svom delu operativne memorije
- ❖ Keš je struktura koja sadrži podskup blokova sa diska učitanih u memoriju. Kada neka operacija kernela traži pristup nekom bloku na disku, taj blok se najpre traži u kešu:
 - ❖ ako je on tu, pristupa se samo sadržaju u operativnoj memoriji, bez pristupa uređaju, što je značajno efikasnije
 - ❖ ako nije u kešu, pokreće se operacija njegovog učitavanja sa diska i on ostaje u kešu, kako bi bio dostupan za neke naredne operacije koje pristupaju istom bloku i koje će tako biti efikasnije
- ❖ Naravno, efikasnost se postiže tako što se najčešće istom bloku pristupa više puta, zbog lokalnosti operacija - isti sadržaj je potreban višekratno
- ❖ OS može organizovati i različite keševe, za različite potrebe, odnosno za blokove koji se koriste za različite namene, npr. keš blokova zamenjenih stranica, keš blokova fajl sistema

Mart 2020.

Copyright 2020 by Dragan Milićev

43

Baferisanje

- ❖ U implementaciji ulazno-izlaznih operacija se intenzivno koriste baferi - delovi operativne memorije koji služe za razmenu podataka između uređaja i kernela
- ❖ Osnovni razlog je već objašnjen - raspreszanje dva učesnika, u ovom slučaju uređaja (koji svakako predstavlja nezavisan, paralelan tok kontrole, ali implementiran u hardveru) i softvera kernela, ili dva toka kontrole unutar softvera. Konkretno:
 - ❖ bafer je jedini način da se obezbedi mesto u koje će uređaj, npr. DMA kontroler ili tastatura, smeštati podatke brzinom kojom on to radi, ili uzimati podatke iz njega; uređaj se ne može jednostavno "suspendovati" zbog toga što nema mesta u baferu; zbog toga se mora obezbediti bafer dovoljnog kapaciteta koji prima ceo blok koji se prenosi, ili se dodatni podaci moraju odbaciti u slučaju da je bafer pun (npr. bafer tastature - otkucani znakovi se jednostavno odbacuju ako je taj bafer pun)
 - ❖ asinhroni upis: da proizvođač ne bi bio suspendovan, odnosno obavljao sinhroni upis (ne nastavlja dalje dok proizvedeni podatak ne bude preuzet), već odmah nastavio, proizvod se mora smestiti u bafer
 - ❖ bafer amortizuje povremene razlike u brzini proizvodnje i potrošnje
 - ❖ bafer omogućava da proizvođač svoje podatke koje je pripremio u nekom svom delu memorije iskopira u bafer i posle toga ne mora više da čuva te originalne podatke, a svoj deo memorije u kom ih je pripremio može da koristi za druge potrebe ili pripremu novih proizvoda

Mart 2020.

Copyright 2020 by Dragan Milićev

46

Keširanje

- ❖ Drugo pitanje jeste to koji blok se zamenjuje (izbacuje) ako je keš (ili pojedinačan ulaz) popunjen i nema slobodnog mesta za učitavanje traženog bloka. Problem je isti kao i za algoritam zamene stranica:
 - ❖ FIFO algoritam (izbaciti onaj blok koji je najdavnije učitao) je jednostavan, ali može biti krajnje neefikasan, jer blok koji je odavno učitao može biti intenzivno korišćen i ponovo potreban u bliskoj budućnosti
 - ❖ Efikasniji algoritam je LRU (*least recently used*): izbacuje se blok koji je najdavnije korišćen, bilo u datom ulazu (lokalno) ili u celom kešu (globalno)
- ❖ Treće pitanje jeste to kada se blok čiji je sadržaj u memoriji izmenjen upisuje na disk. Mogući su različiti pristupi:
 - ❖ odmah pri svakoj izmeni, sinhrono (tzv. *store through*)
 - ❖ odloženo, asinhrono, npr. prilikom izbacivanja (tzv. *write back*)
- ❖ Izbor ove tehnike može zavisi i od vrste sadržaja u bloku, odnosno od toga da li se radi o izbačenim stranicama procesa, delu sadržaja fajla ili metapodacima fajl sistema (podacima o samim fajlovima)
- ❖ Ako sistem sadržaj keša upisuje na uređaj odloženo, npr. u trenutku zatvaranja fajla ili na eksplicitan zahtev, ta operacija snimanja nekog dela sadržaja, npr. fajla iz keša na disk, naziva se "ispiranje" (*flush*)
- ❖ I sam disk kontroler ima svoj (hardverski) keš koji je potpuno transparentan za OS, tako da ne mora da znači da će svaka operacija upućena disku zaista zahtevati pomeranje glave i pristup fizickom bloku na disku

Mart 2020.

Copyright 2020 by Dragan Milićev

45

Baferisanje

- ❖ Keševi i baferi imaju sličnosti, ali su konceptijski različiti, po definiciji:
 - ❖ keš sadrži *kopiju* podskupa podataka iz neke veće memorije u kojoj su svi originalni podaci
 - ❖ bafer služi kao posrednik i može u nekom trenutku sadržati jedinu kopiju nekog podatka
- ❖ Međutim, nije nikakav problem, i to se najčešće i radi, da se isti prostor u memoriji koristi i kao keš i kao bafer. Na primer, prostor za smeštanje bloka u kešu koristi se kao bafer kod DMA prenosa sa diskom

Mart 2020.

Copyright 2020 by Dragan Milićev

48

Baferisanje

- ❖ Osim toga, bafer omogućava i *adaptaciju interfejsa* uređaja, odnosno promenu načina pristupa jednom uređaju iz jednog oblika u drugi:
 - ❖ znakovni u blokovski ili obratno: ako proizvođač proizvodi znak po znak, a potrošač uzima cele blokove odjednom, znakove je potrebno smeštati, jedan po jedan, u bafer, dok se ne nakupi dovoljno za ceo blok koji se može pročitati odjednom; isto i u suprotnom smeru
 - ❖ blokovski sa različitim veličinama bloka: ako proizvođač proizvodi pakete (blokove) jedne veličine, a potrošač uzima blokove druge veličine, potrebno je smeštati blokove u bafer, a onda iz njih "isecati" i uzimati blokove druge veličine
- ❖ Jedna tehnika koja se takođe koristi kod sprege uređaja ili tokova kontrole koji su jako različiti po brzini transfera i jedinici prenosa je *dvostruko baferisanje* (*double buffering*). Na primer, proizvođač je jako spor i proizvodi male jedinice (npr. znakove), ali ravnomerno; potrošač je brz, uzima cele blokove, ali u naletima:
 - ❖ postoje dva bafera, A i B
 - ❖ u jednoj fazi, bafer A je ulazni bafer u koji upisuje proizvođač, a bafer B izlazni iz kog uzima potrošač
 - ❖ kada oba učesnika završe fazu - punjenje, odnosno pražnjenje celog svog bafera, baferi A i B zamenjuju uloge



Mart 2020.

Copyright 2020 by Dragan Milićev

47

Spuling

- ❖ Rešenje je još tada, kod prvih paketnih sistema, osmišljeno pod nazivom *spuling* (*spooling*), i koristi se za štampeče i slične spore nedeljive izlazne uređaje:
 - ❖ proces zatraži korišćenje uređaja posebnim sistemskim pozivom
 - ❖ OS otvori poseban fajl na određenom mestu u svom fajl sistemu u koji će preusmeriti sve izlazne operacije sa tim uređajem i odmah vrati kontrolu procesu
 - ❖ proces zato odmah može da nastavi operacije sa uređajem, koje vrši uporedo sa drugim procesima, a rezultat tih operacija se upisuje u dati fajl
 - ❖ kada proces završi operaciju sa uređajem, "zatvara" taj uređaj, a OS predaje taj fajl na obradu posebnom procesu ili niti operativnog sistema, tzv. *spulernu* (*spooler*)
 - ❖ spulter uzima jedan po jedan fajl kao posao (*job*) i njegov sadržaj šalje na dati uređaj
- ❖ Zapravo se radi o tehnici baferisanja, s tim da procesi upisuju uporedo, svaki u svoj bafer, a ulogu bafera igra fajl
- ❖ Operativni sistemi obezbeđuju korisnički interfejs, ali i API za pregled skupa poslova koji su u redu čekanja na štampaču i upravljanje tim poslovima (zaustavljanje, otkazivanje, ponavljanje i slično)

Mart 2020.

Copyright 2020 by Dragan Milićev

50

Spuling

- ❖ Posebnu kategoriju uređaja predstavljaju nedeljivi, spori izlazni uređaji. Tipičan predstavnik je štampač: nema nikakvog smisla da jedan proces pošalje nekoliko znakova na štampač, a onda neki drugi proces nekoliko svojih znakova i tako naizmenično, a da se ti znakovi štampaju odmah tim redom koji su stigli
- ❖ Jedan način da se ovo reši jeste *rezervacija* uređaja, zapravo međusobno isključenje operacija sa ovakvim uređajem:
 - ❖ pre nego što proces obavi bilo koju operaciju sa ovakvim uređajem, mora da traži njegovu alokaciju, odnosno rezervaciju sistemskim pozivom
 - ❖ ako je uređaj već zauzet, proces mora da čeka da dođe na red
 - ❖ kada dobije uređaj na korišćenje, proces može da obavlja operacije sa njim; na kraju proces oslobađa uređaj i neki drugi proces koji je čekaو može da ga dobije
- ❖ Međutim, kako operacije sa štampačem traju prilično dugo, proces može vrlo dugo da zauzima štampač, zbog čega se mnogi drugi procesi zaustavljaju i dugo čekaju. Ovo je posebno bio problem kod prvih multiprocesnih, paketnih sistema, u kojima su procesi stalno ispisivali svoje rezultate na linijski štampač tokom svog izvršavanja

Mart 2020.

Copyright 2020 by Dragan Milićev

49

Zaštita

- ❖ Kernel po pravilu sprečava da procesi neposredno pristupaju ulazno-izlaznim uređajima, već im operacije sa uređajima pruža samo kroz sistemске pozive. Kako se ta zaštita obezbeđuje:
 - ❖ ako je arhitektura računara takva da se registrima uređaja pristupa posebnim instrukcijama procesora koje se ispoljavaju kao drugačiji ciklusi na magistrali koji adresiraju te registre (instrukcije tipa *In/Out*), te instrukcije su po pravilu dostupne samo u privilegovanom režimu rada procesora, pa ih korisnički procesi ne mogu izvršavati
 - ❖ ako su ulazno-izlazni uređaji preslikani u fizički adresni prostor memorije, delovi tog adresnog prostora u kom su registri kontrolera uređaja zaštićeni su od pristupa korisničkih procesa: kernel ih ili ne mapira u njihove virtuelne adresene prostore, ili im zabranjuje pristup do njih opisanim tehnikama zaštite memorije
- ❖ Prave se i izuzeci od ovoga:
 - ❖ već pomenuti sistemski poziv *iocb* omogućava direktno slanje zahteva uređaju
 - ❖ kernel ponekad dozvoljava pristup do nekih delova memorije koji se preslikavaju u uređaje, kako bi proces mogao da im pristupi direktno; primer je video memorija (memorija bafera za video uređaj, *framebuffer*) grafičkih kartica, radi efikasnijeg prikaza video sadržaja (npr. za igrice)

Mart 2020.

Copyright 2020 by Dragan Milićev

52

Realno vreme

- ❖ Da bi OS pružio usluge vezane za realno vreme, potrebno je da ima odgovarajuću hardversku podršku
- ❖ Svi današnji računari imaju hardverski uređaj koji prati realno vreme, tzv. časovnik realnog vremena (*real time clock*), koji čuva informaciju o tekućem datumu i vremenu; baterijski je podržan, tako da održava tu informaciju i kada je računar isključen. OS može dobiti ovu informaciju čitanjem vrednosti odgovarajućih registara ovog uređaja, ali i ažurirati tu informaciju prema informacijama koje dobija npr. iz komunikacione mreže, od udaljenih servera i njihovih usluga
- ❖ Što se tiče protoka realnog vremena, za te potrebe računar ima hardverske uređaje - vremenske brojače, tzv. *tajmere* (*timer*); njihova implementacija i podrška može biti različita:
 - ❖ kranje jednostavna, tako što tajmer samo periodično generiše prekid procesoru i to je jedina predstava koju softver ima o protoku vremena
 - ❖ računar poseduje jedan ili više programabilnih tajmera; svakom od njih kernel može zadati merenje intervala vremena određene dužine, tako što u odgovarajući registar upiše odgovarajuću vrednost srazmernu dužini intervala; tajmer periodično dekrementira tu vrednost i kada ona stigne do nule, generiše prekid procesoru; kernel treba da obradi taj prekid u zavisnosti od toga šta je hteo merenjem tog intervala
- ❖ Korišćenjem ovih usluga hardvera i svakako ograničenog, malog broja hardverskih tajmera, kernel treba da obezbedi:
 - ❖ raspodelu vremena na procesoru (*time sharing*), tj. ograničenje rada procesa na procesoru pri promeni konteksta
 - ❖ usluge neograničenog broja merenja vremenskih intervala, usvajivanja i ostalih usluga koje pruža procesima
- ❖ Zato kernel treba da organizuje logičke tajmere, odnosno softverske brojače koje ažurira hardverskim uslugama

Mart 2020.

Copyright 2020 by Dragan Milićev

51

dr Dragan Milićev

redovni profesor

Elektrotehnički fakultet u Beogradu

dmilicev@etf.rs, www.rcub.bg.ac.rs/~dmilicev

Operativni sistemi

Slajdovi za predavanja

Deo V - Fajl sistem

Osnovni kurs

Za izbor slajda drži miša
uz levu ivicu ekrana

Mart 2020.

Copyright 2020 by Dragan Milićev

1

Performanse

- ❖ Ulazno-izlazne operacije bitno utiču na performanse celog sistema jer:
 - ❖ povećavaju broj promena konteksta, zbog obrade u različitim tokovima kontrole
 - ❖ uključuju mnogo kopiranja podataka između raznih bafera
 - ❖ povećavaju opterećenje magistrale računara zbog prenosa sa DMA
 - ❖ povećavaju učestanost prekida i njihove obrade
- ❖ Zato se ovaj podsystem mora pažljivo projektovati i implementirati. Neke tehnike koje se koriste sa ciljem unapređenja performansi:
 - ❖ smanjiti broj promena konteksta i njihove troškove (npr. korišćenjem kernel niti umesto posebnih sistemskih procesa za određene radnje)
 - ❖ smanjiti količinu kopiranja podataka prenosom referenci na podatke gde god je to moguće
 - ❖ smanjiti učestanost prekida povećanjem veličine blokova prenosa ili korišćenjem tehnike prozivanja – ako je operacija brza i čekanje veoma kratko, prozivanje je efikasnije nego obrada prekida ili suspenzija
 - ❖ maksimalno iskoristiti dostupne hardverske uređaje za operacije koje oni podržavaju – DMA kontrolere ili specijalizovane ulazno-izlazne procesore
 - ❖ važno je balansirati opterećenje procesora, memorije, magistrale i uređaja, tako da nijedan od njih ne postane usko grlo
- ❖ Po pravilu, U/I operacije su efikasnije kada se vrši prenos manje većih blokova i paketa podataka nego više manjih, jer je odnos režijskih troškova za prenos bloka podataka (za sinhronizaciju, kontrolu, promenu konteksta i drugo) i količine korisnog prenesenog sadržaja povoljniji za veće blokove

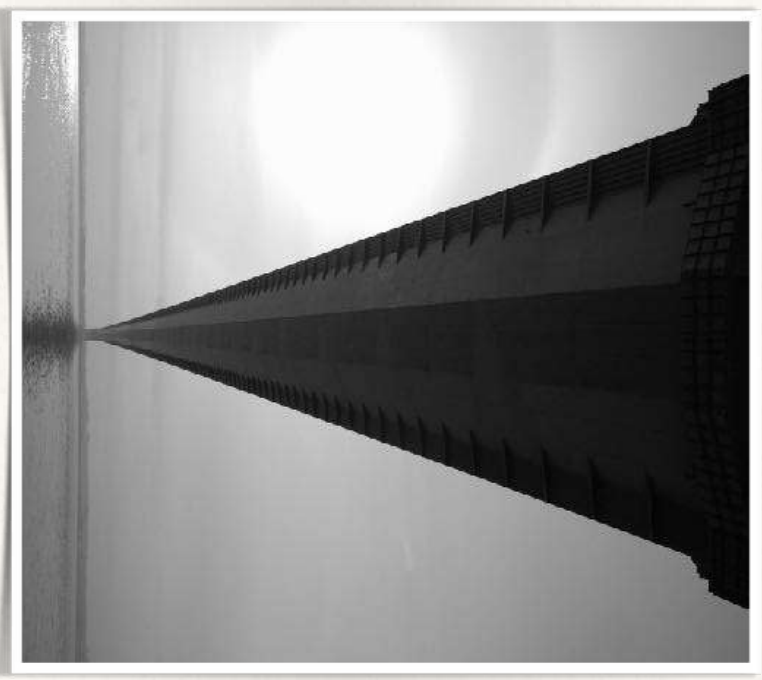
Mart 2020.

Copyright 2020 by Dragan Milićev

53

Glava 12: Interfejs fajl sistema

- ❖ Pojam fajla
- ❖ Operacije sa fajlovima
- ❖ Direktorijum
- ❖ Prava pristupa
- ❖ Uporedan pristup fajlovima
- ❖ Montiranje fajl sistema
- ❖ Pristup udaljenim fajlovima



Mart 2020.

Copyright 2020 by Dragan Milićev

3

Deo V: Fajl sistem

Mart 2020.

Copyright 2020 by Dragan Milićev

2

Pojam fajla

- ❖ Fajl je logički koncept, resurs do kog pristup obezbeđuje OS, ali se može posmatrati i kao objekat, odnosno instanca apstraktnog tipa podataka, jer ima svoje *attribute* i *operacije* koje se nad njim mogu uraditi, a koje proces traži sistemskim pozivima koje OS implementira
- ❖ Skup informacija koje OS održava o fajlu zavisi od samog sistema, ali po pravilu obuhvata sledeće:
 - ❖ jedinstven i nepromenljiv identifikator koji se interno, u implementaciji, koristi za identifikaciju fajla
 - ❖ simboličko ime fajla: niz znakova koji služi za to da korisnik i programi identifikuju fajl, ali koje se može promeniti
 - ❖ tip sadržaja fajla
 - ❖ sadržaj fajla: binarni ili znakovni sadržaj zbog kog fajl i postoji
 - ❖ informacije o sadržaju: gde se nalazi, kako je raspoređen na uređaju i koje je veličine
 - ❖ informacije o tome ko je (koji korisnik) i kada (datum i vreme) kreirao fajl, poslednji put mu pristupao, poslednji izmenio njegov sadržaj ili attribute
 - ❖ informacije o pravima pristupa do fajla: ko (koji korisnik) ima pravo da pristupi do fajla i to kojim operacijama (čitanje, upis)
- ❖ Spisak fajlova sa informacijama o njima naziva se *direktorijum* (*directory*) ili *katalog*

Mart 2020.

Copyright 2020 by Dragan Milićev

5

Pojam fajla

- ❖ *Fajl* (*file*) je logički koncept koji omogućava čuvanje sadržaja na uređajima i pristup tom sadržaju nezavisan od vrste tih uređaja
- ❖ Koncept fajla je upravo osmišljen zato da bi programe učinio nezavisnim od različitosti u načinu smeštanja podataka na uređajima, pristupa tim uređajima, ali i samih operativnih sistema koji upravljaju tim uređajima
- ❖ Fajl je tako univerzalan i jedini način da korisnički procesi, odnosno programi koje oni izvršavaju, organizuju i smeštaju podatke na uređaje, i da to rade na uniforman i prenosiv način
- ❖ Fajl je jedinstven koncept za smeštanje i programa i podataka
- ❖ Osim toga, ideja koncepta fajla jeste i u tome da njegov sadržaj interpretiraju oni programi koji su za to namenjeni i sposobni - svakako program koji je kreirao dati sadržaj fajla, ali i neki drugi
- ❖ OS u principu ne ulazi u sadržaj fajla, već samo obezbeđuje način rukovanja fajlom, njegovo smeštanje na uređaj, kao i mehanizam kojim procesi mogu da pristupaju sadržaju fajla
- ❖ Naravno, postoje i izuzeci od toga. Minimalan izuzetak jeste *exe* fajl u čiji sadržaj OS mora da ulazi i da ga interpretira, kako bi pokrenuo proces nad njim
- ❖ Osim toga, operativni sistemi interpretiraju i mnoge druge formate sadržaja, recimo preko svojih sistemskih programa koji su za to namenjeni. Na primer, interpretiraju tekstualne fajlove, PDF, HTML, audio i video formate, kao i mnoge druge
- ❖ Operativni sistemi koriste i mnoge svoje, sistemske fajlove u kojima čuvaju svoje perzistentne podatke, poput konfiguracije sistema i korisnika

Mart 2020.

Copyright 2020 by Dragan Milićev

4

Pojam fajla

- ❖ Sledeće pitanje jeste to na koji način OS može predstavljati, kroz operacije pristupa sadržaju fajla, sam sadržaj fajla (koji se svakako na kraju zapisuje binarno). Moguće su različite varijante:
 - ❖ kao jednostavan niz bajtova; OS ni na koji način ne strukturira taj sadržaj, pa se ostavlja programu da svoje interne strukture podataka pretvori u niz bajtova - tzv. *serijalizacija* (*serialization*, *marshalling*) i obratno, da iz niza bajtova izgradi svoje interne strukture podataka - tzv. *deserijalizacija* (*deserialization*, *unmarshalling*)
 - ❖ kao niz znakova - tzv. *znakovni tok* (*character stream*)
 - ❖ kao niz složenijih struktura (zapisa, *records*); ovaj pristup je primenjivan u prošlosti, sada se više ne primenjuje, jer se ova vrsta transformacije iz internog binarnog zapisa u strukture i obratno prepušta programima ili bibliotekama koje rade serijalizaciju i deserijalizaciju nekih standardnih formata zapisa (npr. XML, JSON itd), pa ta odgovornost nije više na operativnom sistemu, izmeštena je van njega
- ❖ Pošto se sadržaj fajla tipično smešta na blokove uređaje, OS mora da ovakve nizove bajtova ili znakova sadržaja fajla podeli na blokove fiksne veličine i njih rasporedi na uređaju

Mart 2020.

Copyright 2020 by Dragan Milićev

7

Pojam fajla

- ❖ OS može, ali ne mora da poznaje pojam *tipa* (*type*) fajla. Ako ga poznaje, informacija o tipu fajla koristi se na sledeći način:
 - ❖ kada se neki program instalira na sistem, tokom postupka instalacije sistemu se “prijavi” to koje sve tipove fajla taj program ume da interpretira
 - ❖ OS omogućava i način da se neki program postavi kao podrazumevan za neki tip fajla
 - ❖ OS pruža uslugu pokretanja programa za dati tip fajla, kojom zapravo kreira proces nad programom koji je registrovan za dati tip fajla i potom mu pošalje poruku da otvori odabrani fajl
 - ❖ ovu uslugu mogu zatražiti drugi procesi ili je može pokrenuti korisnik kroz GUI
- ❖ Informacija o tipu može se predstavljati na različite načine:
 - ❖ kao deo simboličkog imena fajla, tzv. *ekstenzija* (*extension*) - znakovi iza tačke u imenu fajla (DOS, Windows)
 - ❖ kao eksplicitan atribut fajla
 - ❖ u prvih nekoliko bajtova sadržaja fajla, kao tzv. *magic number*: ako prvi bajt sadržaja ima određenju vrednost (npr. 0x7F), na osnovu kojih OS prepoznaje ovu informaciju, narednih nekoliko bajtova koduju tip fajla

Mart 2020.

Copyright 2020 by Dragan Milićev

6

Operacije sa fajlovima

- ❖ POSIX sistemski poziv za otvaranje i kreiranje fajla:
`int open (const char *pathname, int flags, mode_t mode);`
- ❖ Prvi parametar zadaje simboličko ime fajla koji se otvara
- ❖ Drugi parametar, kao i inače, u svojim bitima zadaje modalitete ove operacije. Mnogo simboličkih konstanti definiše ove modalitete; uključivanje više njih vrši se logičkom operacijom *ili* po bitima ovih konstanti. Ovde su navedene samo neke:
 - ❖ `O_CREAT`: ako fajl sa datim imenom ne postoji, biće kreiran; ako ovaj fleg nije postavljen, otvaranje nepostojećeg fajla vraća grešku; ako je uključen i `O_EXCL`, otvaranje postojećeg fajla vratiće grešku
 - ❖ `O_RDONLY`, `O_WRONLY`, `O_RDWR`: barem jedna od ovih mora biti uključena i onda definiše operacije koje će proces vršiti sa fajlom, odnosno koje će mu biti dozvoljene da radi (samo čitanje, samo upis ili i čitanje i upis, respektivno)
 - ❖ `O_APPEND`: fajl se otvara za proširenje sadržaja, odnosno dodavanje na postojeći sadržaj; tekuća pozicija za upis postavlja se na kraj postojećeg sadržaja fajla i dalje upis vrši se iza te pozicije
 - ❖ `O_TRUNC`: ako je dozvoljen upis, fajl se otvara, ali mu se postojeći sadržaj briše, tako da upis počinje iznova, od praznog sadržaja
 - ❖ postoji mnogo flegova koji utiču na modalitet implementacije operacija; na primer, `O_DIRECT` zahteva sinhroni upis i čitanje, uz zaoblazanje internih keševa kernela
- ❖ Poslednji parametar *mode* ima značenje samo ako je u argumentu *flags* postavljen `O_CREAT` (fajl se kreira), u suprotnom ovaj argument može da se izostavi. Tada se ovim argumentom zadaju prava pristupa do fajla, opet postavljanjem simboličkih konstanti

Mart 2020.

Copyright 2020 by Dragan Milićev

9

Operacije sa fajlovima

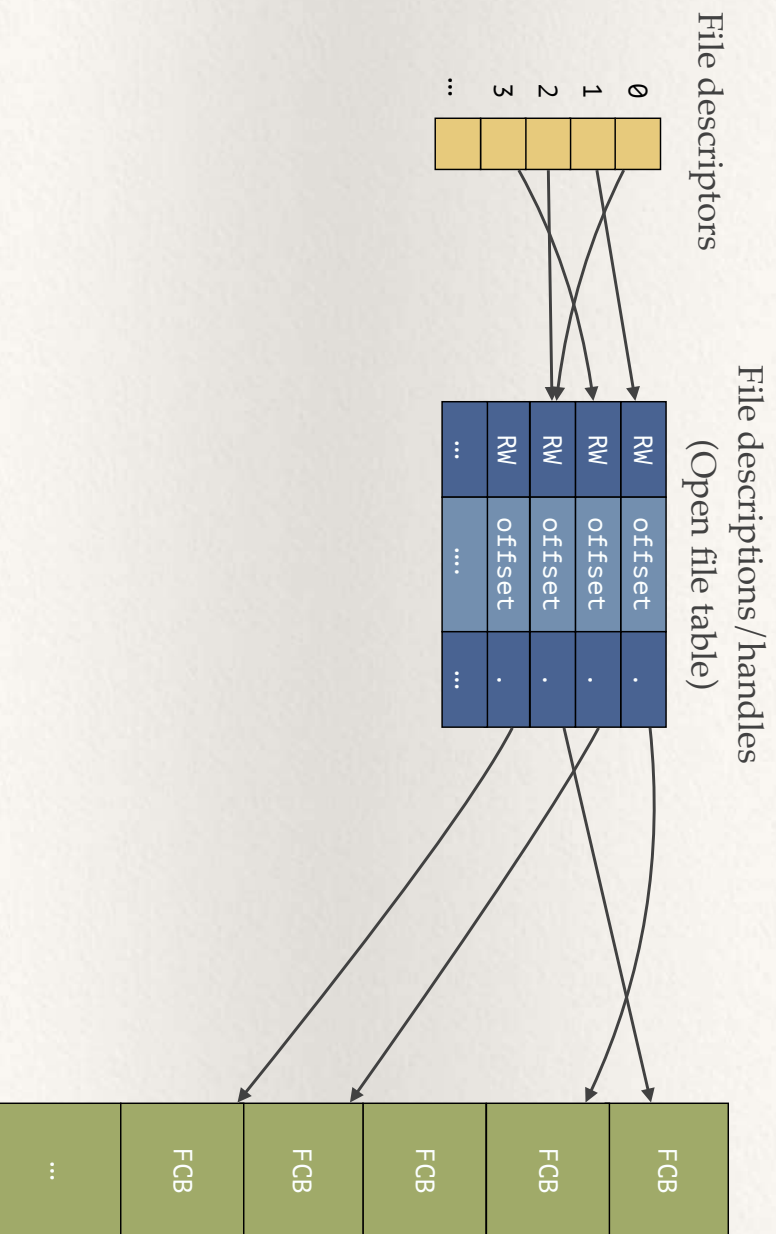
- ❖ Različiti operativni sistemi pružaju različite usluge (i različite API) za operacije sa fajlovima, ali je većina današnjih programskih interfejsa veoma slična po koncepciji i principima i ima puno sličnosti sa onim što će ovde biti prikazano
- ❖ Ovde će biti prikazane osnovne operacije sa fajlovima korišćenjem primera dva prilično slična programska interfejsa (za kompletn spisak funkcija i objašnjenja njihovog delovanja potrebno je konsultovati dokumentaciju za ove biblioteke):
 - ❖ POSIX
 - ❖ standardna biblioteka jezika C (*libc standard I/O*) za ulazno-izlazne tokove (*I/O streams*, `<stdio.h>`, `<csdio>`)
 - ❖ Prva operacija jeste *kreiranje* fajla, ali se ona najčešće objedinjuje sa operacijom *otvaranja* fajla
 - ❖ Pojam otvaranja je opšti pojam koji važi i za druge resurse kojima OS rukuje: da bi dobio ili kreirao resurs, proces mora da ga *otvori* odgovarajućim sistemskim pozivom; čak su i interfejsi (potpisi operacija) tih sistemskih poziva slični i imaju zajedničke delove (neke parametre), recimo one za proveru prava pristupa ili dozvoljene operacije (čitanje / upis)
 - ❖ Kada završi sa korišćenjem resursa, pa i fajla, proces mora da izvrši operaciju njegovog *zatvaranja*. OS tada oslobađa taj resurs i, ako ga nijedan drugi proces ne koristi, dealocira strukture u memoriji kojim ga predstavlja

Mart 2020.

Copyright 2020 by Dragan Milićev

8

Operacije sa fajlovima



Mart 2020.

Copyright 2020 by Dragan Milićev

11

Operacije sa fajlovima

- ❖ Interno, za svaki proces, kernel održava jednu tabelu (ili vektor, niz) tzv. *deskriptora fajlova* (*file descriptor*): pokazivača na strukture koje se nazivaju *opisi fajlova* (*file description*), *ručke fajlova* (*file handle*), ulazi u tabeli *otvorenih fajlova* (*open file table entry*) ili, u terminologiji implementacije kernel koda, *struct file*
- ❖ Ove strukture, opisi fajlova, slažu se u strukturu koja se tradicionalno naziva *tabela otvorenih fajlova* (*open file table*)
- ❖ Ova struktura, opis fajla, odnosno ulaz u tabeli otvorenih fajlova, pripada kontekstu procesa i sadrži, principijelno:
 - ❖ informacije o tome koje je operacije proces najavio prilikom otvaranja (flegovima *O_RDONLY*, *O_WRONLY*, *O_RDWR*)
 - ❖ *tekuću poziciju* (*current file position*, *offset*) u sadržaju fajla koja se podrazumeva kao pozicija od koje se vrši sledeća zadata operacija čitanja ili upisa sadržaja
 - ❖ pokazivač na strukturu koja sadrži attribute fajla, učitane sa uređaja, i pomoću kojih kernel obavlja operacije sa tim fajlom; ova struktura je deljena između svih procesa koji su otvorili taj fajl, globalna je za ceo sistem; ova struktura predstavlja fajl kao objekat u fajl sistemu na uređaju i naziva se tradicionalno *kontrolni blok fajla* (*file control block*, *FCB*) ili, u žargonu sistema nalik sistemu Unix, *inode* (prema nazivu strukture u C kodu kernela)
- ❖ Kada otvori fajl na zahtev procesa, kernel zauzima prvi slobodan ulaz u tabeli deskriptora datog procesa i kao rezultat uspešnog sistemskog poziva *open* vraća mali ceo broj koji predstavlja indeks tog ulaza u tabeli (u slučaju greške, vraća -1)
- ❖ Kao što je ranije rečeno, svaki proces po pokretanju ima otvorene "fajlove", odnosno deskriptore u ulazima 0, 1 i 2 za *stdin*, *stdout* i *stderr*, respektivno
- ❖ Sistemski pozivi *dup* i *dup2* kopiraju samo deskriptor u ovom vektoru, tako da dva ulaza u tom vektoru ukazuju na istu strukturu u kojoj je tekuća pozicija fajla

Mart 2020.

Copyright 2020 by Dragan Milićev

10

Operacije sa fajlovima

- ❖ Vrednosti koje ove funkcije otvaranja fajla vraćaju u slučaju uspeha koriste se kao argumenti koji identifikuju fajl pri svim ostalim operacijama sa tim fajlom i važeći su čak i ako fajl promeni svoje simboličko ime ili se premesti u drugi direktorijum
- ❖ POSIX sistemski poziv za zatvaranje fajla:
`int close (int fd);`
 - ❖ Vraća 0 u slučaju uspeha, -1 u slučaju greške
 - ❖ Ovaj poziv ne garantuje da se izmenjeni sadržaj fajla upisuje na uređaj pri zatvaranju fajla, jer se možda baferiše - nije sigurno da će kernel odmah "isprati" bafer /keš (*flush*). Na ovo je potrebno obratiti pažnju i konsultovati dokumentaciju, pa upotrebiti odgovarajuće modalitete i sistemske pozive
- ❖ C *stdio* sistemski poziv za zatvaranje fajla:
`int fclose (std::FILE* stream);`
 - ❖ Vraća 0 u slučaju uspeha, *EOF* u slučaju greške
 - ❖ Ovaj pak sistemski poziv garantuje ispiranje (*flush*) bafera /keševa

Mart 2020.

Copyright 2020 by Dragan Milićev

13

Operacije sa fajlovima

- ❖ C *stdio* sistemski poziv za otvaranje i kreiranje fajla:
`std::FILE* fopen (const char* filename, const char* mode);`
- ❖ Ova funkcija vraća pokazivač na ručku fajla, odnosno strukturu koja predstavlja ulaz u tabeli otvorenih fajlova ili *null* u slučaju greške
- ❖ Prvi parametar zadaje simboličko ime fajla koji se otvara
- ❖ Drugi parametar koduje, znacima u nizu znakova, modalitet na sledeći način:

<i>mode</i>	Značenje	Objašnjenje	Ako fajl postoji	Ako fajl ne postoji
"r"	<i>read</i>	Otvori fajl za čitanje	čitaj od početka	greška
"w"	<i>write</i>	Kreiraj fajl za upis	obriši sadržaj	kreiraj nov
"a"	<i>append</i>	Dodaj na fajl	piši na kraj	kreiraj nov
"r+"	<i>read extended</i>	Otvori fajl za čitanje / upis	čitaj od početka	greška
"w+"	<i>write extended</i>	Kreiraj fajl za čitanje / upis	obriši sadržaj	kreiraj nov
"a+"	<i>append extended</i>	Otvori fajl za čitanje / upis	piši na kraj	kreiraj nov

Mart 2020.

Copyright 2020 by Dragan Milićev

12

Operacije sa fajlovima

- ❖ POSIX sistemski pozivi za čitanje iz fajla, upis u fajl i pomeranje tekuće pozicije:
`ssize_t read (int fd, void *buffer, size_t count);`
`ssize_t write (int fd, const void *buffer, size_t count);`
`off_t lseek (int fd, off_t offset, int whence);`
 - ❖ vraćaju broj stvarno učitanih ili upisanih bajtova, odnosno novu poziciju
 - ❖ parametar *whence* definiše to u odnosu na šta se postavlja tekuća pozicija: *SEEK_SET* - u odnosu na početak sadržaja fajla, *SEEK_CUR* - za *offset* bajtova u odnosu na tekuću poziciju, *SEEK_END* - *offset* bajtova iza kraja sadržaja fajla (može biti i negativan)
- ❖ C *stdio* sistemski pozivi za čitanje iz fajla, upis u fajl i pomeranje tekuće pozicije:
`std::size_t fread (void* buffer, std::size_t size, std::size_t count, std::FILE* stream);`
`std::size_t fwrite (const void* buffer, std::size_t size, std::size_t count, std::FILE* stream);`
`int fseek (std::FILE* stream, long offset, int origin);`
 - ❖ ovde parametar *size* zadaje veličinu jednog "objekta", a *count* broj takvih objekata; svaki "objekat" se posmatra kao *size* susednih znakova tipa *unsigned char*
 - ❖ vraćaju broj stvarno učitanih ili upisanih znakova, odnosno status uspeha (0 - uspeh, -1 - greška)
 - ❖ argument *origin* ima isto značenje kao i argument *whence*

Mart 2020.

Copyright 2020 by Dragan Milićev

15

Operacije sa fajlovima

- ❖ C *stdio* API obezbeđuje već opisan pristup sadržaju fajla kao ulazno-izlaznom znakovnom toku, pa nudi skup bibliotečnih funkcija za ulaz-izlaz, zasnovanih na osnovnim *getc* i *putc*, u različitim varijantama u zavisnosti od veličine znakova i načina njihovog kodovanja, kao i za formatizovan ulaz-izlaz (*fscanf* i *fprintf*)
- ❖ Oba ova programska interfejsa podržavaju već pomenut koncept *tekuće pozicije* (*current position*) ili *pomeranja* (*offset*) koji je pridružen svakom otvorenom fajlu u kontekstu svakog procesa koji mu pristupa (svaki proces ima svoju, nezavisnu poziciju)
- ❖ Oba interfejsa takode obezbeđuju usluge za čitanje (*read*) ili upis (*write*) niza bajtova ili znakova proizvoljne dužine, počev od tekuće pozicije; nakon ovih operacija, tekuća pozicija se implicitno pomera iza pročitanoog odnosno upisanog niza znakova/bajtova u sadržaju fajla
- ❖ Postoji i operacija tipa *seek* koja eksplicitno postavlja, odnosno pomera tekuću poziciju na zadato mesto u sadržaju fajla
- ❖ Odgovarajuće operacije su dozvoljene samo ako je fajl otvoren sa tim operacijama najavljenim pri otvaranju, inače će sistem vratiti grešku
- ❖ Za detalje semantike ovih operacija potrebno je konsultovati njihovu dokumentaciju



Mart 2020.

Copyright 2020 by Dragan Milićev

14

Direktorijum

- ❖ Jedan disk (uređaj) se može logički podeliti na (jednu ili) više tzv. *particija (partition)*, disjunktih delova, odnosno skupova blokova, koji se dalje posmatraju kao nezavisne celine
- ❖ Svaka particija, ili čak više particija, moguće i na više različitih diskova, može se logički organizovati u tzv. *volumen (volume)*
- ❖ Svaki volumen OS posmatra kao nezavisan logički uređaj. Sistemi na različite načine identifikuju ove logičke uređaje-diskove, npr. slovima abecede (*c*;, *d*;, *e*; itd; *a*; i *b*; su bili korišćeni za floppy diskove koji danas više nisu u upotrebi)
- ❖ Prema tome, jedan volumen može se prostirati na jednoj ili više particija, pa i na jednom ili više fizičkih uređaja i obratno, jedan fizički uređaj može sadržati jedan ili više logičkih uređaja (diskova)
- ❖ Na svakom volumenu može se organizovati fajl sistem, inicijalizacijom struktura podataka na volumenu koje su za to potrebne. Ovaj postupak naziva se *logička formatizacija diska (formatting)*
- ❖ Particija ne mora biti formatizovana, odnosno na njoj ne mora da bude organizovan fajl sistem, već se ona može upotrebljavati za posebne namene, npr. kao prostor za zamenu stranica (*swap space*). Ovakve particije nazivaju se *presne (raw)*, za razliku od formatizovanih, koje se u žargonu nazivaju i *kuvarne (cooked)*
- ❖ Na svakom logički formatizovanom volumenu može se organizovati poseban, i to čak i drugačiji fajl sistem
- ❖ U svakom fajl sistemu potrebno je na neki način organizovati spisak svih fajlova i informacija o njima - *direktorijum (directory)*

Mart 2020.

Copyright 2020 by Dragan Milićev

17

Operacije sa fajlovima

- ❖ Pored ovih osnovnih, operativni sistemi pružaju i mnoge druge operacije nad fajlovima, kao što su:
 - ❖ promena imena fajla
 - ❖ promena drugih atributa fajla, npr. prava pristupa
 - ❖ brisanje sadržaja fajla (*truncate*): briše se samo sadržaj, ne i fajl kao entitet
 - ❖ brisanje fajla kao entiteta (fajl potpuno nestaje) i druge
- ❖ Ove ali i druge, izvedene operacije, npr. kopiranje fajla, premeštanje fajla i slično, dostupne su i kroz korisnički interfejs (CLI i GUI). Ove operacije implementiraju se kao sistemske komande ili sistemski programi koji koriste navedene elementarne sistemske pozive u svojoj implementaciji

Mart 2020.

Copyright 2020 by Dragan Milićev

16

Direktorijum

- ❖ Najjednostavniji način da se organizuje direktorijum jeste taj da postoji samo jedan jedinstven direktorijum na volumenu, u kom je spisak svih fajlova na tom volumenu. Ovakav pristup imao bi mnogo očiglednih nedostataka:
 - ❖ ovakav direktorijum bi brzo narastao na stotine i hiljade fajlova, što postaje teško za snalaženje
 - ❖ kako svaki fajl mora imati jedinstveno simboličko ime u svom direktorijumu, svi fajlovi u tom sistemu morali bi da imaju jedinstveno ime, što je nepraktično
 - ❖ pitanje je kako razvrstati i odvojiti fajlove različitih korisnika na višekorisničkom sistemu
- ❖ Nešto pogodniji pristup bio bi razvrstavanje fajlova različitih korisnika u različite direktorijume, tako da svaki korisnik ima svoj, ali jedan direktorijum u kom su svi njegovi fajlovi. I dalje bi to bilo nepraktično:
 - ❖ i dalje korisnik može da ima na stotine i hiljade fajlova u istom “džaku”, što je teško za snalaženje, veoma nepregledno
 - ❖ pitanje je gde smestiti fajlove kojima treba da pristupaju svi korisnici (npr. programi); jedan pristup je da se odvoji jedan zajednički direktorijum za takve fajlove
- ❖ Zbog svega ovoga ova rešenja su nepraktična i ne primenjuju se

Mart 2020.

Copyright 2020 by Dragan Milićev

19

Direktorijum

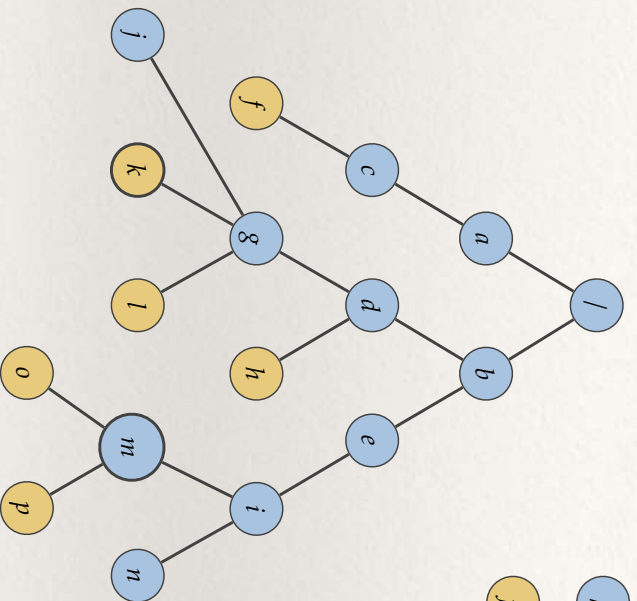
- ❖ Osnovni zadatak direktorijuma jeste da simboličko ime fajla preslika u strukturu koja čuva attribute fajla (FCB), kako bi OS mogao da pronađe tu strukturu i učita je sa uređaja prilikom otvaranja
- ❖ Prema tome, simboličko ime fajla je zapravo ključ za pretragu unutar direktorijuma, pa se može smatrati identifikatorom fajla u opsegu jednog direktorijuma
- ❖ Zbog toga ovaj identifikator mora biti jedinstven u opsegu jednog direktorijuma, ali u opštijem slučaju može postojati više identifikatora koji identifikuju isti fajl (kao entitet, koji ima svoj identitet samim postojanjem) u različitim direktorijumima
- ❖ Neke osnovne operacije sa direktorijumom su, prema tome:
 - ❖ pronađi element direktorijuma sa datim simboličkim imenom (i preslikaj u njegov FCB)
 - ❖ promeni simboličko ime elementa direktorijuma
 - ❖ dodaj element u direktorijum
 - ❖ izbaci (obriši) element iz direktorijuma
 - ❖ vrati spisak svih elemenata direktorijuma itd.
- ❖ Ove operacije podržane su sistemskim pozivima, ali i kroz korisnički interfejs (CLI i GUI), odgovarajućim komandama ili sistemskim programima

Mart 2020.

Copyright 2020 by Dragan Milićev

18

Direktorijum



I Direktorijum

f Fajl

❖ Apsolutna, puna staza do čvora k :
 $/b/d/g/k$

❖ Relativna putanja do čvora k u odnosu na čvor m :
 $../d/g/k$

Mart 2020.

Copyright 2020 by Dragan Milićev

21

Direktorijum

❖ Praktičnija organizacija jeste *hijerarhijska struktura* direktorijuma u obliku stabla:

❖ jedan direktorijum može sadržati fajlove, ali i poddirektorijume;

❖ jedan čvor u strukturi (fajl ili direktorijum) može biti sadržan samo u jednom direktorijumu (tzv. roditeljski direktorijum, *parent*)

❖ tačno jedan čvor je *koreni direktorijum* (*root*), u opštim slučajju, može biti više takvih korenih direktorijuma (šuma stabala)

❖ Ovakva struktura je pogodna jer odgovara čestom i praktičnom pristupu koji ljudi primenjuju u borbi protiv kompleksnosti, kada treba da pregledno organizuju veliki broj pojedinačnih elemenata: grupisanje u kategorije koje su organizovane hijerarhijski

❖ Ovakva organizacija ima niz važnih svojstava:

❖ svaki čvor, kao element svog roditeljskog direktorijuma, mora imati jedinstveno ime samo u opsegu tog roditeljskog direktorijuma, ali koje ne mora biti jedinstveno u celom fajl sistemu; ovo ime je *nekvantifikovano* ili *prosto* ime čvora

❖ do svakog čvora od korena dolazi jedna i samo jedna, jedinstvena putanja; ovakva putanja naziva se *apsolutna putanja* ili *puna putanja* do čvora (*absolute path*, *full path*), a pun, kvalifikovan naziv fajla sadrži sva nekvantifikovana imena čvorova duž te putanje, razdvojena nekim posebnim znakom, najčešće je to kosa crta ($/$) ili obrnuta kosa crta (\backslash)

❖ od svakog čvora do svakog drugog čvora u stablu dolazi jedinstvena putanja koja se naziva *relativna* putanja

❖ Za označavanje koraka na gore (ka roditeljskom direktorijumu) u relativnoj putanji obično se koristi poseban simbol (npr. dve tačke, $..$). Fajl sistemi često prave i uvek postojeće, podrazumevane ulaze u svakom direktorijumu, kako bi ovu navigaciju radili na uniforman način, kao pretragu ulaza u direktorijum sa datim simboličkim imenom:

❖ ulaz sa imenom $..$ označava roditeljski direktorijum

❖ ulaz sa imenom $..$ označava isti ovaj direktorijum

Mart 2020.

Copyright 2020 by Dragan Milićev

20

Direktorijum

- ❖ Hijerarhijska struktura oblika stabla je jednostavna i pravilna, ali ima jedan nedostatak: hijerarhijsku strukturu organizuje korisnik, klasifikujući svoje fajlove prema svom nađenju; međutim, ponekad (ne tako retko) potrebno je isti fajl klasifikovati prema različitim kriterijumima u više direktorijuma, što ova organizacija ne dozvoljava
- ❖ Zato se struktura može uopštiti u strukturu *usmerenog grafa bez petlji*, tj. *usmerenog acikličkog grafa* (*directed acyclic graph*, DAG). Na ovaj način isti fajl, kao entitet ili objekat, može biti klasifikovan u više direktorijuma i dostupan preko više putanja
- ❖ Ovo je ideja, koncepcija koja se na različite načine podržava i implementira u različitim operativnim sistemima

Mart 2020.

Copyright 2020 by Dragan Milićev



23

Direktorijum

- ❖ U kontekstu svakog procesa, OS čuva informaciju o tome koji direktorijum u hijerarhiji se smatra *tekućim radnim direktorijumom* (*current working directory*) tog procesa. Relativne putanje koje proces koristi u identifikaciji fajla prilikom njegovog otvaranja uzimaju se u odnosu na ovaj tekući direktorijum. Podrazumevano se tekući direktorijum može naslediti od procesa roditelja, a može se promeniti prilikom pokretanja procesa ili kasnije, odgovarajućim sistemskim pozivom; sistemi dozvoljavaju i podešavanje (kroz korisnički interfejs) podrazumevanog tekućeg direktorijuma za svaki instalirani program
- ❖ Kada proces otvara fajl, kao argument za stazu do fajla u odgovarajućem sistemskom pozivu može navesti:
 - ❖ punu putanju do fajla; ova putanja prepoznaje se tako što počinje oznakom za koreni direktorijum (npr. `"/b/d/g/mydoc"`)
 - ❖ relativnu putanju do fajla; ova putanja prepoznaje se tako što ne počinje oznakom za koreni direktorijum (npr. `"/../d/g/mydoc"`); ova relativna putanja uzima se u odnosu na tekući direktorijum procesa
- ❖ Smisao postojanja koncepta tekućeg direktorijuma je u sledećem. Kada taj pojam ne bi postojao, svaki program bi fajlove koje koristi morao da identifikuje punom putanjom, a tada bi kod programa, zbog takvih putanja ugrađenih u taj kod, bio zavisen od celokupne organizacije direktorijuma u fajl sistemu, pa bi bio neprenosiv na računar sa drugačijom strukturom direktorijuma. Postojanjem ovog koncepta, kod programa zavisi samo od podstabla u strukturi direktorijuma u kojoj se on sam nalazi, odnosno u kom je instaliran, tako da ne zavisi od pozicije u strukturi direktorijuma u kojoj je on instaliran, jer sve fajlove koji su mu potrebni može da referencira relativnim putanjama

Mart 2020.

Copyright 2020 by Dragan Milićev

22

Direktorijum

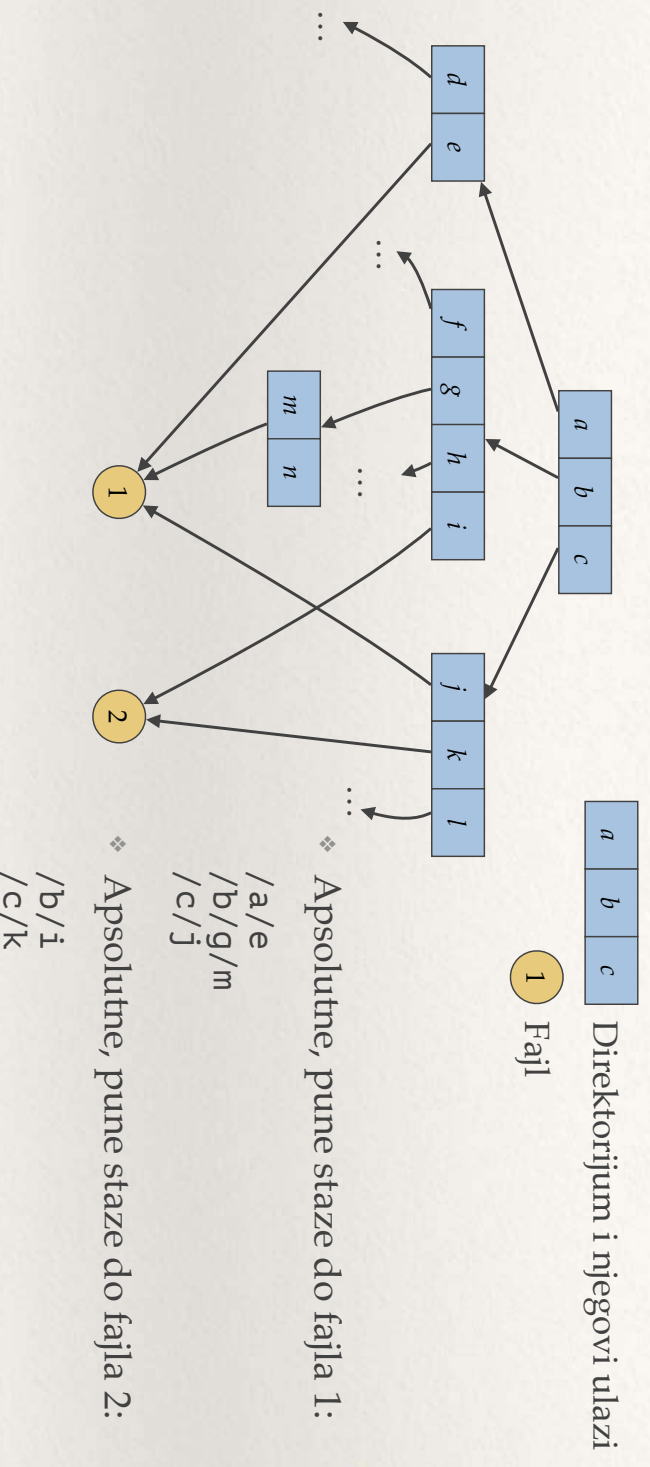
- ❖ Jedan način za implementaciju reference na fajl jeste tzv. *simbolička veza* (*symbolic link*, *symlink*): zapravo se implementira kao fajl u strukturi direktorijuma, ali posebne vrste i sa posebnim tretmanom od strane operativnog sistema; ovaj fajl sadrži niz znakova koji definiše relativnu ili apsolutnu putanju do čvora (fajla ili direktorijuma) koji referencira
- ❖ Ovo je "meká" referenca (*soft link*), jer može biti viseća ili korumpirana: ako se određišni (*target*) čvor premesti ili preimenuje, ova referenca se ne ažurira automatski, pa može postati "prekinuta" (*broken*) ili "landarava" (*dangling*)
- ❖ Suština je u tome da OS usluge u kojima procesi referenciraju fajl koji je simbolička veza zapravo preusmeravaju ka referenciranom fajlu:
 - ❖ operaciju otvaranja fajla koji je simbolička veza preusmerava na referencirani fajl
 - ❖ komande u korisničkom interfejsu preusmerava na referencirani fajl
- ❖ Na ovaj način, simbolička veza predstavlja i drugo ime za dati fajl, njegov *alijas* (*alias*)
- ❖ I ovaj koncept ima različite implementacije i pojave u različitim sistemima
- ❖ Jedan aspekt implementacije jeste to gde se smešta putanja na referencirani fajl; opcije su:
 - ❖ u samom sadržaju fajla, kao tekstualni zapis, dok poseban bit u FCB ukazuje na to da je fajl simbolička veza
 - ❖ kao atribut fajla, u strukturi FCB, što je efikasnije (tzv. *brze veze*, *fast links*)
- ❖ Drugo pitanje jeste to koje tačno operacije OS preusmerava na referencirani fajl, a koje imaju efekta na samu simboličku vezu

Mart 2020.

Copyright 2020 by Dragan Milićev

25

Direktorijum



Mart 2020.

Copyright 2020 by Dragan Milićev

24

Direktorijum

- ❖ Drugi način da se implementira struktura direktorijuma tipa DAG jesu tzv. *torde veze* (*hard link*): ova veza, odnosno referenca, predstavlja preslikavanje simboličkog imena, kao ulaza u direktorijumu, u fajl (tj. njegov FCB), s tim da se dozvoljava da više različitih simboličkih imena i ulaza u različitim direktorijumima upućuje na isti fajl kao entitet, baš kao što je nacrtano na prethodnoj slici
- ❖ Naravno, svaka operacija otvaranja fajla sa datim simboličkim imenom koji ga identifikuje otvara taj fajl kao entitet
- ❖ Na sistemima nalik sistemu Unix, tvrda veza pravi se komandom *lnk* (bez modifikatora), kao i sistemskim pozivom *link*
- ❖ Kako fajl sada može imati više imena i više staza do sebe, i svi oni predstavljaju alijase do fajla, postavlja se pitanje kada se briše sam fajl
- ❖ U većini sistema komanda koja briše čvor sa zadatim imenom zapravo samo briše dati ulaz u direktorijumu; sam fajl, kao entitet (odnosno njegov FCB) nestaje kada nestane poslednja tvrda veza ka njemu
- ❖ Ovo se implementira brojanjem referenci (*reference count*), kao standardnom tehnikom koja se primenjuje u sličnim situacijama: kad god se uspostavi nova veza, ovaj brojčak se inkrementira, a kad veza nestane, on se dekrementira; kada brojčak padne na 0, briše se i sam referencirani objekat

Mart 2020.

Copyright 2020 by Dragan Milićev

27

Direktorijum

- ❖ U sistemima nalik sistemu Unix, simbolička veza se pravi komandom *ln* sa opcijom *-s* u CLI:
`ln -s target_path link_path`
target_path je relativna ili apsolutna putanja do referenciranog (ciljnog) čvora; *link_path* je staza do simboličke veze
- ❖ Kada se napravi simbolička veza, mnoge komande (npr. *cp* za kopiranje) i operacije čitanja ili upisa koje koriste simboličku vezu OS preusmerava na referencirani fajl; komande za brisanje (*rm*, *remove*) i premeštanje (*mv*, *move*) se odnose na simboličku vezu, a ne na referencirani fajl
- ❖ Komanda za listanje (*ls*) sadržaja direktorijuma prikazuje simboličke veze pomoću strelice (*->*), npr:

```
$ ls -l mydoc
lrwxrwxrwx 1 user group 20 Apr 04 10:02 /tmp/mydoc -> /tmp/one/two
```
- ❖ I fajl sistem NTFS (*NT File System*) na sistemu Windows podržava sličan koncept simboličkih veza, kao i komandu za njihovo pravljenje *mklink*
- ❖ I sistem Mac OS podržava ovaj koncept kao koncept *alijasa*, s tim da održava simboličke veze ispravnim čak i pri premeštanju ili preimenovanju referenciranog fajla
- ❖ Sistem Windows podržava i koncept *prečice* (*shortcut*) do fajla, koji liči na koncept simboličke veze, uz sledeće razlike: sve operacije nad simboličkom vezom obrađuje sam OS i on ih preusmerava na referencirani fajl, dok to ne radi za prečice. Prečice OS vidi kao najobičnije fajlove, a odgovornost za interpretaciju prečica preuzimaju programi koji znaju da ih interpretiraju (među njima je i GUI, kao i pregledač direktorijuma i fajlova, *File Explorer*)

Mart 2020.

Copyright 2020 by Dragan Milićev

26

Prava pristupa

- ❖ U višekorisničkim sistemima potrebno je kontrolisati pristup do fajlova i direktorijuma, kako bi se sprečio neovlašćen pristup do tuđih fajlova
- ❖ Operativni sistemi primenjuju različite koncepte i tehnike zaštite pristupa, ovde će biti opisani samo neki
- ❖ Mehanizam zaštite (*protection*) ima dva elementa:
 - ❖ kako se definišu prava pristupa korisnicima do fajlova i kako se ona predstavljaju
 - ❖ kako OS proverava prava pristupa i pristup dozvoljava ili ne
- ❖ Po pravilu, OS proverava prava pristupa kada neki proces traži (sistemskim pozivom) otvaranje nekog fajla. Tom prilikom, kako je pokazano, proces najavljuje operacije koje želi da radi sa tim fajlom. OS proverava da li su mu te operacije dozvoljene i ako nisu, odbija otvaranje (vraća grešku); ako jesu, procesu će nadalje biti dozvoljene samo te operacije koje je najavio, iako bi mu možda i neke druge bile dozvoljene pravima pristupa - proces ne može da izvršava operacije koje nije najavio pri otvaranju
- ❖ Vrlo često operativni sistemi isti mehanizam kontrole prava pristupa do fajlova koriste i za direktorijume, ali i za druge resurse koje procesi dobijaju od operativnog sistema (npr: semafori, deljena memorija itd), pa potpisi odgovarajućih sistemskih poziva imaju zajedničke delove koji se odnose na proveru prava pristupa

Mart 2020.

Copyright 2020 by Dragan Milićev

29

Direktorijum

- ❖ Da li ima smisla napraviti i dalje uopštenje, pa dozvoliti i petlje u grafu koji predstavlja strukturu direktorijuma?
- ❖ Ovakav pristup imao bi mnoge nedostatke i bio bi veoma nepraktičan za upotrebu (zbunjujući), a teži i neefikasan za implementaciju. Zato se ovaj pristup ne koristi u praksi
- ❖ Upravo kako bi sprečili formiranje kružnih staza, mnogi sistemi zabranjuju kreiranje tvrdih veza ka direktorijumima; kako tvrda veza mora biti u direktorijumu, a mora referencirati samo fajl koji je uvek list, garantovano nema petlji u grafu

Mart 2020.

Copyright 2020 by Dragan Milićev

28

Prava pristupa

- ❖ Operacije zapravo izvršavaju procesi, pa se proveru vrši za procese. Međutim, svaki proces se izvršava "u ime" nekog korisnika; korisnik u čije se ime izvršava proces (*user id*, *uid*) je tako deo konteksta procesa:
 - ❖ podrazumevano, proces nasleđuje ovo svojstvo od roditeljskog procesa
 - ❖ proces koji rukuje procedurom prijavljivanja korisnika na sistem (*log in*) izvršava se sa dovoljnim privilegijama da, kada obavi autentikaciju korisnika (proveru identiteta, npr. pomoću unošenja lozinke), kreira proces koji izvršava školjku u ime tog prijavljenog korisnika
 - ❖ postoji i način da proces kreira drugi proces tako što će se taj proces izvršavati u ime nekog drugog korisnika; naravno, tom prilikom OS može zahtevati autentikaciju korisnika (proveru da on može da se predstavi kao traženi korisnik, npr. unošenjem lozinke)
- ❖ Na sistemima nalik sistemu Unix, sistemski komanda *sudo* (*superuser do*) omogućava izvršavanje komande koja je u argumentu ovog programa, odnosno pokretanje procesa sa privilegijama drugog korisnika, podrazumevano "superkorisnika" (*superuser*) koji ima sva prava (administrator sa korisničkim imenom *root*)

Mart 2020.

Copyright 2020 by Dragan Milićev

31

Prava pristupa

- ❖ Zadatak ovog dela sistema jeste to da na određeni način predstavi informacije, omogućći njihovo definisanje i održavanje, kao i upotrebu kod provere pristupa, o konceptualnoj relaciji između sledeća tri skupa:
 - ❖ skup svih korisnika registrovanih u sistemu, što je svakako evidencija koju OS vodi
 - ❖ skup svih fajlova u fajl sistemu
 - ❖ skup svih operacija koje se OS omogućava sa fajlovima
- ❖ Ako su data tri elementa, korisnik-fajl-operacija, u relaciji, onda je ta operacija nad tim fajlom dozvoljena tom korisniku
- ❖ Ekvivalentno, radi se o funkciji koja preslikava trojke (korisnik, fajl, operacija) u Bulovu vrednost koja kaže da li je ta operacija dozvoljena ili nije

Mart 2020.

Copyright 2020 by Dragan Milićev

30

Prava pristupa

- ❖ Sistemi nalik sistemu Unix primenjuju sledeći pristup:
 - ❖ svaki korisnik ima svoj jedinstven identifikator (*user id*, *uid*), ali pripada i jednoj *grupi korisnika*; svaka grupa korisnika identifikuje se jedinstvenim celim brojem (*group id*, *gid*)
 - ❖ svaki proces izvršava se u ime nekog korisnika, kao što je opisano (*uid* je deo konteksta procesa, ali isto tako i korisnikov *gid*)
 - ❖ korisnik u čije ime se izvršavao proces koji je kreirao fajl ili direktorijum (čvor), odnosno izvršio sistemski poziv kreiranja tog čvora, jeste njegov *vlasnik* (*owner*); *uid* i *gid* vlasnika su atributi čvora
 - ❖ za svaki čvor definisana su prava pristupa (jedan bit - dozvoljeno ili ne) za tri grupe korisnika:
 - owner*: prava vlasnika fajla / direktorijuma
 - group*: prava korisnika koji pripadaju istoj grupi kao vlasnik fajla / direktorijuma
 - others*: svi ostali korisnici
- i tri grupe operacija:
 - read*, *r*: sve operacije koje samo čitaju i ne menjaju sadržaj ili attribute fajla / direktorijuma
 - write*, *w*: sve operacije koje menjaju sadržaj ili attribute
 - execute*, *x*: izvršavanje fajla (poziv tipa *exec*), odnosno pretraga za direktorijume

Mart 2020.

Copyright 2020 by Dragan Milićev

33

Prava pristupa

- ❖ Jedan pristup definisanju ove relacije jeste taj da se svakom fajlu pridruži lista zapisa ovih parova (korisnik, operacija) koji definišu operacije dozvoljene korisnicima za taj fajl
- ❖ Ovakva lista naziva se *lista kontrole pristupa* (*access control list*, *ACL*)
- ❖ Kako su skupovi fajlova i korisnika, ali i operacija veliki, ovaj pristup ima niz nedostataka:
 - ❖ definisanje ovakvih informacija je naporno: za svaki fajl, svaku operaciju i svakog korisnika kom je ta operacija dozvoljena treba definisati ovakav zapis
 - ❖ za zapisivanje ovakve liste neophodne su logički neograničene, dinamičke strukture podataka, čije su održavanje i pretraga manje efikasni
- ❖ Zato se u praksi primenjuju kompaktnije varijante koje se svode na to da se skupovi o kojima se ovde radi (korisnici, fajlovi, operacije) klasifikuju u veće grupe srodnih i prava pristupa definišu za te veće skupove, odnosno sve njihove elemente

Mart 2020.

Copyright 2020 by Dragan Milićev

32

Prava pristupa

- ❖ Prava pristupa za fajl zadaju se odgovarajućim argumentom prilikom njegovog kreiranja (parametar *mode*), a mogu se kasnije i promeniti jednim od sistemskih poziva:

```
int chmod (const char *pathname, mode_t mode);  
int fchmod (int fd, mode_t mode);
```
- ❖ Iz komandne linije, prava pristupa se mogu promeniti komandom *chmod* čiji argument *mode* predstavlja niz znakova koji se sastoji iz sledećeg:
 - ❖ slovne oznake *u* (za vlasnika), *g* (za grupu) ili *o* (za ostale) koja definiše za koga se menjaju prava
 - ❖ oznake + (za uključivanje pomenutih prava), - (za isključivanje pomenutih prava), ili = (za uključivanje pomenutih, a isključivanje nepomenutih prava)
 - ❖ oznake *r*, *w* ili *x* za prava

Na primer:

```
chmod u-x, g=rw, o-w mydoc
```

Vlasniku zabrani izvršavanje, grupi dozvoli čitanje i upis, a zabrani izvršavanje, ostalima zabrani upis za fajl *mydoc*

Mart 2020.

Copyright 2020 by Dragan Milićev

35

Prava pristupa

- ❖ Ova prava mogu se videti komandom listanja sadržaja direktorijuma sa uključenom opcijom *-l*:

```
$ ls -l  
drwx----- 6 JohnDoe staff 192 Oct 27 2015 Applications  
drwx----- 7 JohnDoe staff 224 Mar 2 21:51 Desktop  
drwx----- 17 JohnDoe staff 544 Sep 13 2018 Documents  
..  
drwxr-xr-x 4 JohnDoe staff 128 Sep 27 2013 Public  
drwxr-xrwx 5 JohnDoe staff 160 Mar 21 2017 Sites
```

Naziv fajla ili direktorijuma

- ❖ Prema tome, OS prilikom otvaranja vrši proveru na sledeći vrlo jednostavan i efikasan način: ako se proces izvršava u ime vlasnika fajla (tj. *uid* procesa jednak je *uid* fajla), proveravaju se prava definisana za vlasnika; u suprotnom, ako se proces izvršava u ime korisnika koji pripada istoj grupi kao i vlasnik (tj. *gid* procesa jednak je *gid* fajla), proveravaju se prava pristupa za grupu; u suprotnom, proveravaju se prava pristupa za ostale

Mart 2020.

Copyright 2020 by Dragan Milićev

34

Uporedan pristup fajlovima

- ❖ Prvo, razlika može biti u tome kada i na koji način se fajl zaključava; moguće su sledeće varijante:
 - ❖ implicitno, prilikom operacije otvaranja fajla ili prilikom svake operacije sa fajlom; proces ne mora ništa eksplicitno da zahteva, već operacije podrazumevaju implicitno zaključavanje fajla
 - ❖ eksplicitno, na poseban zahtev procesa
- ❖ Drugo pitanje je to kakvom vrstom ključa se fajl zaključava. Moguće su varijante:
 - ❖ postoji samo jedna vrsta ekskluzivnog ključa, tako da se sve operacije procesa potpuno isključuju, odnosno proces može dobiti ključ samo ako nijedan drugi proces ne drži ključ
 - ❖ postoji više vrsta ključeva koji se ne isključuju uvek u potpunosti, a različite operacije koriste ili traže različite vrste ključeva; tipično postoje sledeće dve vrste ključeva:
 - ❖ *deljeni (shared)* ključ je potreban i dovoljan za operacije čitanja; procesi koji imaju deljeni ključ mogu vršiti operacije čitanja (i samo njih) uporedo, bez međusobnog isključenja, ali samo ako nijedan drugi proces nema ekskluzivni ključ
 - ❖ *ekskluzivni (exclusive)* ključ je potreban za operacije upisa; proces može dobiti ekskluzivan ključ samo ako nijedan drugi proces ne drži nijedan ključ (bilo koje vrste)

Mart 2020.

Copyright 2020 by Dragan Milićev

37

Uporedan pristup fajlovima

- ❖ Današnji operativni sistemi omogućavaju uporedan pristup fajlovima, tako da se fajl može koristiti i kao deljeni objekat za međuprocensnu komunikaciju: više procesa uporedo može čitati iz istog fajla i / ili upisivati u njega
- ❖ Semantika izvršavanja operacija nad fajlom od strane uporednih procesa zavisi od sistema i uvek je potrebno konsultovati dokumentaciju za sistemske pozive
- ❖ Na primer, POSIX zahteva da pojedinačne operacije sa fajlom, npr. operacije upisa u fajl i čitanja iz fajla imaju serijalizovani efekat, što znači da se pojedinačne operacije više izolovano (atomično), sa efektom kao da su izvršene u *nekom* redosledu (neodređeno kom), s tim da se te operacije uporednih procesa (ili niti) mogu preplitati
- ❖ Ponekad je potrebno obezbediti međusobno isključenje pristupa fajlu, tako da jedan proces može obaviti više operacija nad fajlom izolovano. Za to se mogu koristiti uobičajene sinhronizacione primitive (npr. semafori), kao i za ostale deljene objekte, ali sistemi često podržavaju i neposredne mehanizme *zaključavanja fajla (file locking)*, kojim se operacije nad fajlom mogu međusobno isključivati
- ❖ I kod ovih mehanizama postoje razlike u načinima podrške

Mart 2020.

Copyright 2020 by Dragan Milićev

36

Montiranje fajl sistema

- ❖ Kada se pokrene operacija montiranja, operativni sistem:
 - ❖ proverava da li je fajl sistem koji se montira podržan, odnosno da li ima prepoznatljiv format
 - ❖ organizuje strukture podataka koje predstavljaju taj fajl sistem u kernelu, preusmerava operacije sa njim na odgovarajući drajver
 - ❖ učitava informacije o tom fajl sistemu i organizuje svoje interne strukture podataka, tako da se tom fajl sistemu pristupa na isti način kao i ostalim delovima strukture direktorijuma, odnosno tako da bude deo jedinstvenog prostora imena i strukture direktorijuma
- ❖ Vrlo često postoje ograničenja u pogledu mesta na koje se u glavnoj strukturi direktorijuma mogu montirati fajl sistemi:
 - ❖ samo u koren jedinstvene hijerarhije, ili samo kao novo stablo u šumi stabala
 - ❖ samo u prazan direktorijum
- ❖ Suština ove operacije jeste u tome da nakon montiranja, direktorijumima i fajlovima iz montiranog sistema procesi pristupaju na isti način kao i svim drugim delovima jedinstvene strukture direktorijuma, imenujući fajlove apsolutnim i relativnim putanjama, pri čemu su razlike u fajl sistemima i njihov položaj na različitim uređajima potpuno transparentni za procese

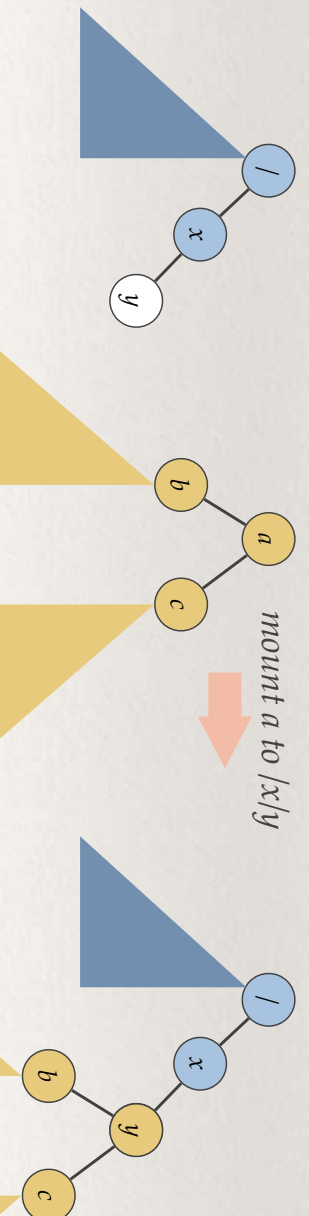
Mart 2020.

Copyright 2020 by Dragan Milićev

39

Montiranje fajl sistema

- ❖ Da bi OS procesima obezbedio pristup do fajl sistema na nekom volumenu, pa i na onom koji je spoja priključen na računar (npr. USB fleš memorija) ili se nalazi na udaljenom računaru, mora najpre izvršiti operaciju *montiranja* (*mounting*) tog fajl sistema
- ❖ OS održava jedinstvenu logičku strukturu direktorijuma u obliku stabla ili šume stabala, pri čemu su podstabla (ili stabla koja čine šumu), zapravo strukture direktorijuma sa različitim fajl sistema na različitim volumenima (pa i uređajima)
- ❖ Konceptualno, operacija montiranja znači sledeće: stablo (ili, u opštijem slučaju, podstablo) strukture direktorijuma sa datog volumena priključuje kao podstablo (ili novo stablo u šumi) jedinstvene strukture direktorijuma koje OS održava i do kog procesima pruža pristup



Mart 2020.

Copyright 2020 by Dragan Milićev

38

Pristup udaljenim fajlovima

- ❖ U prošlosti nije bio moguć direktan pristup fajlovima na drugim računarima, već su se fajlovi prenosili isključivo pomoću prenosivih medijuma (flopi diskovi, magnetne trake)
- ❖ Nastankom i dostupnošću računarskih mreža, postupno su uvođeni protokoli koji su omogućavali da se fajl sa jednog računara prenese u fajl sistem drugog računara preko mreže, kao i specijalizovan softver za to
- ❖ Jedan od prvih standardnih protokola za tu namenu, koji je i danas u upotrebi, jeste tzv. *File Transfer Protocol*, FTP. Princip rada sa ovim protokolom je sledeći:
 - ❖ u razmeni fajlova učestvuju dva računara, jedan koji igra ulogu servera i drugi koji igra ulogu klijenta
 - ❖ na oba računara izvršavaju se programi koji implementiraju ovaj protokol u navedene dve uloge (serverska i klijentska); podrška za FTP može biti ugrađena i u sam kernel, kao i u CLI, dok GUI obično implementiraju posebni programi
 - ❖ korisnik interaguje sa klijentskim programom i najpre mora da uspostavi vezu sa određitim serverskim programom, pri čemu mora da se prijavi, odnosno autentifikuje pomoću korisničkog imena i lozinke koju server proverava; za potrebe javnog pristupa fajlovima, serveri obično podržavaju i tzv. anonimno (*anonymous*) prijavljivanje bez zahtevane lozinke
 - ❖ postoji koncept tekućeg direktorijuma na klijentu i na serveru za datu sesiju, kao i komande kojim korisnik može da promeni te tekuće direktorijume; fajlovi na klijentu i na serveru imenuju se relativnim putanjama u odnosu na te tekuće direktorijume
 - ❖ na raspolaganju su i dve osnovne operacije (i mnogo drugih): *get* za prenos fajla sa servera na klijent i *put* za prenos fajla sa klijenta na server; prema tome, fajlovi se eksplicitno prenose komandama koje se izdaju u ovom softveru

Mart 2020.

Copyright 2020 by Dragan Milićev

41

Montiranje fajl sistema

- ❖ Ranije je bilo neophodno izvršiti montiranje drugih fajl sistema eksplicitnom komandom koju izdaje korisnik sa dovoljnim ovlašćenjima
- ❖ Danas se montiranje uglavnom vrši automatski, implicitno:
 - ❖ prilikom podizanja sistema, OS montira sve fajl sisteme na uređajima koji su u stalnoj konfiguraciji računara (diskovi); u jednoj varijanti, spisak fajl sistema koji se montiraju nalazi se u posebnom konfiguracionom fajlu
 - ❖ odmah po priključenju uređaja na računar pokreće se operacija montiranja fajl sistema; na primer, prilikom priključenja fleš memorije, eksternog diska, foto ili video kamere na USB priključnicu računara
 - ❖ OS može da montira i fajl sisteme udaljenih računara sa kojima je uspostavio vezu preko lokalne mreže ili nekim drugim komunikacionim kanalom (npr. Bluetooth)

- ❖ Obrnuta operacija, demontiranje (*umounting*): fajl sistem na volumenu više nije deo jedinstvene strukture direktorijuma i nije više dostupan procesima. Može se pokrenuti eksplicitno (operacija “izbaci”, *eject*) ili implicitno, samim isključivanjem uređaja

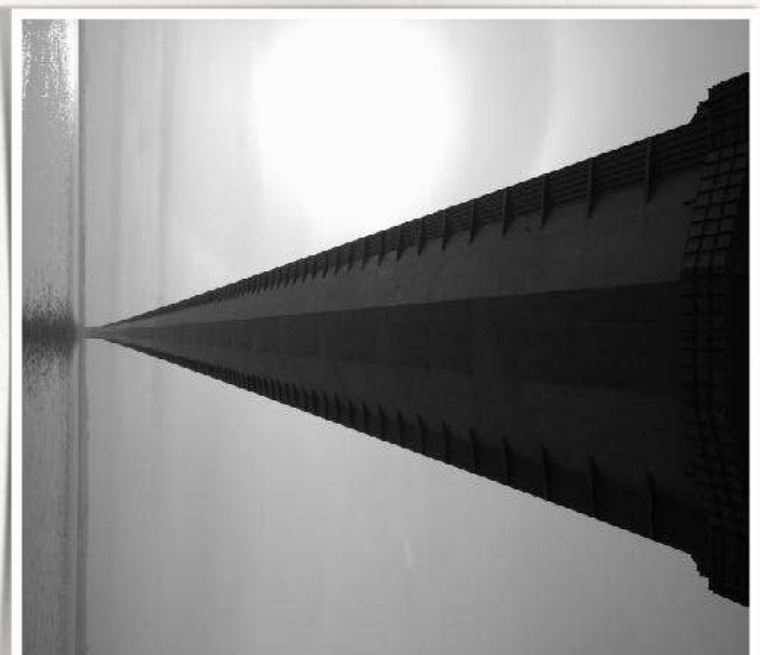
Mart 2020.

Copyright 2020 by Dragan Milićev

40

Glava 13: Implementacija fajl sistema

- ❖ Organizacija fajl podsistema
- ❖ Implementacija direktorijuma
- ❖ Metode alokacije fajla
- ❖ Rukovanje slobodnim prostorom
- ❖ Efikasnost i performanse
- ❖ Otpornost na otkaze



Mart 2020.

Copyright 2020 by Dragan Milićev

43

Pristup udaljenim fajlovima

- ❖ Današnji operativni sistemi podržavaju i mnoge druge načine za pristup do fajlova na drugim računarima i to često implicitno, bez posebne akcije korisnika:
 - ❖ prenos fajlova preko Interneta putem HTTP protokola (kroz veb brauzere), preko mejla i slično
 - ❖ pristup fajlovima na montiranim volumenima koji se nalaze na drugim računarima ili uređajima na mreži; sa tim drugim računarima ili uređajima kernel komunicira odgovarajućim standardnim protokolima tzv. mrežnog fajl sistema (*Network File System, NFS*), kojima se obezbeđuje interoperabilnost čak i između različitih operativnih sistema
- ❖ Posebna vrsta operativnih sistema ili specijalizovanog softvera (npr. za sisteme u oblaku) implementira *distribuirane fajl sisteme* (*distributed file system, DFS*). DFS omogućava jedinstven pogled na strukturu direktorijuma i fajlova koji su fizički raspoređeni na različitim računarima u distribuiranom sistemu, uz rešavanje problema partitionisanja mreže (nedostupnosti nekih čvorova zbog otkaza računara ili veze) i konzistentnosti promena sadržaja fajlova

Mart 2020.

Copyright 2020 by Dragan Milićev

42

Organizacija fajl podсистема

- ❖ Navedeni zadatak se može uraditi na neograničeno mnogo načina, od kojih su neki bolji, efikasniji, praktičniji o drugih, ali i takvih ima mnogo i svi imaju svoje dobre i manje dobre osobine
- ❖ Upravo zato i postoji mnogo implementacija fajl sistema, kako na različitim operativnim sistemima, tako čak i na istom operativnom sistemu — jedan OS može podržati mnogo formata zapisa fajl sistema na uređaju. Na primer:
 - ❖ Unix: *Unix File System* (UFS)
 - ❖ Linux: preko 40 podržanih fajl sistema, bazični je tzv. *Extended File System* (*ext2*, *ext3*)
 - ❖ Windows: NTFS, FAT, FAT32
 - ❖ Mac OS X: APFS
 - ❖ Postoje međunarodni standardi za zapise fajl sistema na prenosivim uređajima, npr. CD-ROM (ISO 9660), DVD, floppy disk itd.
- ❖ Implementacije fajl sistema jako variraju i sve imaju svoje specifičnosti. Ovde se prikazuju samo neki osnovni principi
- ❖ Ova diskusija se prvenstveno odnosi na smeštanje fajl sistema na diskove kao najčešće međijune. Primarne karakteristike diskova koje ih čine pogodnim za implementaciju fajl sistema su sledeće:
 - ❖ to su blokovski orijentisani uređaji sa mogućnošću čitanja i upisa, i sa samo dve osnovne operacije — pročitaj blok u memoriju, a nakon izmene njegovog sadržaja u memoriji, upiši ga na disk; blok tipično sadrži nekoliko sektora, a veličine sektora su od 32 B do 4 KB, tipično 512 B
 - ❖ omogućavaju direktan pristup bilo kom bloku, uz odgovarajuće vreme pristupa i transfera

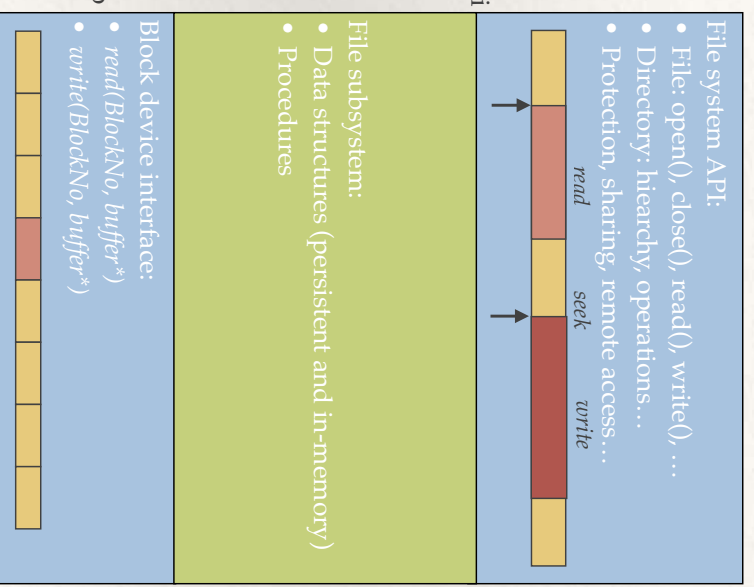
Mart 2020.

Copyright 2020 by Dragan Milićev

45

Organizacija fajl podсистема

- ❖ Fajl podсистem kernela treba da obezbedi sve opisane koncepte i one operacije sa njima koje OS pruža procesima:
 - ❖ koncept fajla, njegovih atributa i operacija sa njima: otvaranje, čitanje, upis, zatvaranje itd.
 - ❖ koncept direktorijuma, strukture direktorijuma i operacije sa njima
 - ❖ zaštitu pristupa fajlovima, uporedan pristup, udaljen pristup i sve ostalo
- ❖ Sa druge strane, na raspolaganju za implementaciju ovog interfeisa jeste jednostavan interfeis blokovskih uređaja, koji pruža nizak nivo apstrakcije — niz blokova sa pridruženim celobrojnim adresama i samo dve jednostavne osnovne operacije:
 - ❖ pročitaj blok sa datim brojem u memoriju
 - ❖ upiši blok sa datim brojem iz memorije
- ❖ Zadatak fajl podсистema jeste to da od raspoloživih apstrakcija realizuje zahtevane, da podigne nivo apstrakcije. Naravno, za ovo su potrebne odgovarajuće strukture podataka, kako one koje su zapisane na samom uređaju, tako i one koje se organizuju u operativnoj memoriji, kao i operacije nad njima



Mart 2020.

Copyright 2020 by Dragan Milićev

44

Organizacija fajl podsistema

- ❖ Kao što je već rečeno, FCB (*File Control Block*) je struktura podataka koja se zapisuje na disku za svaki fajl i sadrži attribute fajla, uključujući i informacije o tome gde je i kako smešten sadržaj tog fajla; na sistemima nalik sistemu Unix, ova struktura naziva se *inode*
- ❖ Jedno od osnovnih pitanja jeste to gde se i kako strukture FCB smeštaju na volumenu. Neki pristupi su sledeći:
 - ❖ sistemi nalik sistemu Unix: FCB se može smestiti bilo gde na disku, u bilo koji slobodan blok; direktorijum sadrži informaciju o tome gde se nalazi FCB, odnosno preslikava simboličko ime čvora u adresu njegove strukture *inode* (broj bloka u kom se nalazi)
 - ❖ NTFS: organizuje se jedna globalna tabela (vektor struktura), tzv. *Master File Table*, čiji su ulazi zapravo strukture FCB; vodi se evidencija o slobodnim i zauzetim ulazima u ovoj tabeli i FCB za nov fajl smešta u slobodan ulaz
- ❖ Drugo pitanje jeste to gde se i kako smeštaju strukture koje predstavljaju direktorijume. Neki pristupi su:
 - ❖ sistemi nalik sistemu Unix: direktorijum se predstavlja na isti način kao i fajl, ima isto svoj FCB (*inode*, zato se i zovu jednim imenom, čvor), samo što jedan bit u FCB ukazuje na to da se radi o direktorijumu; sam sadržaj ovakvog "fajla" čuva informacije o preslikavanjima simboličkih elemenata čvorova u adrese njihovih struktura FCB
 - ❖ neke posebno organizovane, drugačije strukture

Mart 2020.

Copyright 2020 by Dragan Milićev

47

Organizacija fajl podsistema

- ❖ Svaki fajl sistem najpre zahteva organizaciju određenih struktura podataka na samom volumenu. Ove strukture inicijalizuju se logičkom formatizacijom volumena
- ❖ Naravno, skup ovih struktura i način njihove organizacije upravo i zavise od vrste fajl sistema, ali ovo su neke koje po pravilu postoje na volumenu:
 - ❖ *Boot Control Block*: uvek na određenom mestu, tipično blok broj 0 na volumenu; ukoliko prvi bajt ili bajtovi u ovom bloku imaju neku definisanu vrednost (tzv. *magic number*), *bootstrap* program u ROM-u računara prepoznaje da ovaj blok sadrži *bootstrap* program za učitavanje operativnog sistema; pri podizanju računara, ovaj blok se učitava i kod u njemu izvršava za dalje učitavanje sistema; ako particija sadrži ovaj blok, naziva se *bootable*; u suprotnom, ovaj blok se ne koristi
 - ❖ *Volume Control Block*: na unapred definisanom mestu, odnosno bloku na volumenu, sadrži osnovne informacije o volumenu i globalne parametre fajl sistema na njemu, kao što su npr. ukupan broj blokova, veličinu bloka, eventualno broj slobodnih blokova, pokazivač na prvi slobodan blok u listi i slično
 - ❖ Struktura podataka kojom se implementiraju direktorijumi
 - ❖ Struktura podataka kojom se implementiraju strukture koje opisuju svaki fajl, FCB
 - ❖ Struktura podataka koja vodi evidenciju o slobodnim blokovima

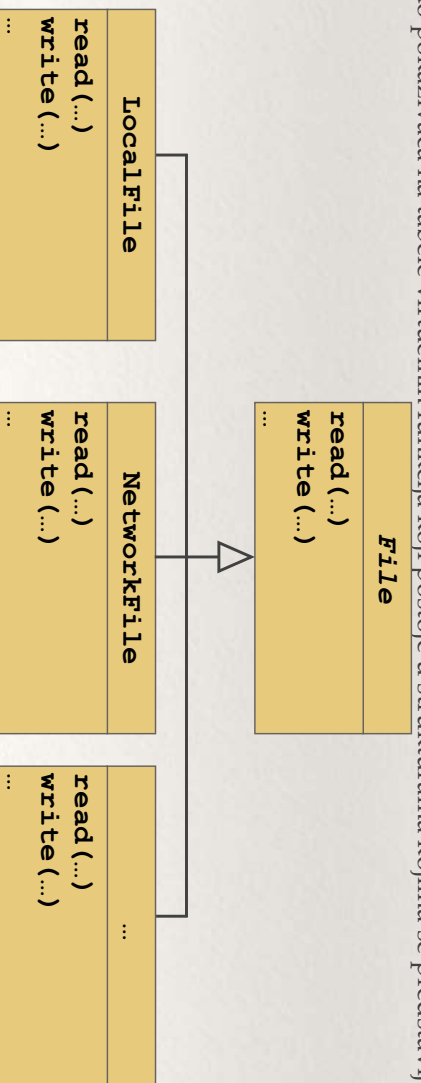
Mart 2020.

Copyright 2020 by Dragan Milićev

46

Organizacija fajl podsistema

- ❖ Kako se fajlovi mogu smeštati na različite uređaje, pa čak biti i udaljeni, da bi rešio ovu varijabilnost, OS opisane pojedinačne strukture (ulaze u tabelama koji opisuju pojedinačne objekte) zapravo vidi kao objekte koji zadovoljavaju neki interfejs (skup operacija koje se od njih zahtevaju), a koji je definisan u nekoj apstraktnoj osnovnoj klasi. Ceo ovaj skup interfejsa naziva se *virtuelni fajl sistem* (*Virtual File System, VFS*)
- ❖ Objekti su instance nekih konkretnih, izvedenih klasa koje redefinišu date operacije, čiji delovi implementacije koriste usluge odgovarajućih drajvera i drugih podsistema kernela: podsistema koji vrši operacije sa diskom priključenim na računar, podsistema mrežne komunikacije, ili komunikacije sa USB uređajem, ili nečeg potpuno drugog
- ❖ Kao i ranije, kako su kerneli tipično implementirani na jeziku C, ovo se implementira dinamičkim vezivanjem preko pokazivača na tabele virtuelnih funkcija koji postoje u strukturama kojima se predstavljaju ovi objekti



Mart 2020.

Copyright 2020 by Dragan Milićev

49

Organizacija fajl podsistema

- ❖ Kada montira fajl sistem, OS u memoriji izgrađuje strukture podataka koje koristi za izvršavanje operacija sa tim sistemom. Neke tipične strukture su sledeće:
 - ❖ tabela montiranih fajl sistema koja u svakom ulazu ima strukturu koja opisuje jedan montiran fajl sistem
 - ❖ tabela deskriptora fajlova (*file descriptor*) za svaki proces, kao deo konteksta procesa (*slika*); svaki ulaz ukazuje na opis fajla (*file description*), odnosno ulaz u tabeli otvorenih fajlova
 - ❖ *tabela otvorenih fajlova lokalnih za procese* (*local open file table, slika*), u kojoj svaki ulaz pripada kontekstu nekog procesa; ulaz je opis ili ručka fajla (*file description, file handle, struct file*) i sadrži tekuću poziciju za operacije čitanja i upisa za dati proces, operacije koje je proces najavio prilikom otvaranja, kao i pokazivač na FCB datog fajla
 - ❖ *globalna tabela otvorenih fajlova* (*global open file table*), odnosno tabela struktura FCB (*inode*) za fajlove koje neki od procesa još uvek drži otvoren, i koji je učitao sa diska, a sadrži atribute fajla i informacije o mestu i načinu organizacije njegovog sadržaja
 - ❖ keširane strukture za ulaze u direktorijumima kojima se već pristupalo; kako je zadatak direktorijuma da preslika simboličko ime u (adresu strukture) FCB u koje se preslikava to ime, pri čemu se te informacije nalaze zapisane na disku, OS kešira ona preslikavanja (ulaze u direktorijumima) koja su prethodno već bila korišćena, kako ih ne bi svaki put iznova učtao sa diska
 - ❖ keš blokova sa sadržajem fajlova kojima se nedavno pristupalo

Mart 2020.

Copyright 2020 by Dragan Milićev

48

Organizacija fajl podsistema

(Nastavak)

- ❖ U strukturi keširanih ulaza u direktorijumima traži se postojanje ulaza sa stazom `"/x/y/";` kao i malopre, ako on postoji, preslikaće u FCB koji je već učitlan. Tada se pristupa toj strukturi FCB i od nje se (pozivom odgovarajuće polimorfne operacije) traži preslikavanje ulaza "z" u identifikator ili adresu FCB na uređaju. Ovo se obavlja čitanjem sadržaja ovog čvora, kao i za svaki drugi fajl, a onda taj sadržaj interpretira u zavisanosti od implementacije direktorijuma (detalji kasnije).
Ako ovo nije slučaj, postupak se nastavlja iterativno unazad sve do korenog direktorijuma /, čiji FCB svakako jeste učitlan, a onda se ide naniže na isti način (traže se redom strukture FCB za /, x, y, z).
- Svi ulazi u direktorijumima kojima se pristupalo i za koje su izvršena preslikavanja u FCB koji su tom prilikom i učitani, smeštaju se u strukturu keširanih ulaza u direktorijumima za neke naredne pristupe
- ❖ Nakon ovog postupka, određena je adresa (broj bloka) strukture FCB traženog fajla na uređaju. Taj FCB (*inode*) se učitava korišćenjem odgovarajuće operacije draivera uređaja za učitavanje datog bloka u memoriju. Ovaj FCB učitava se u globalnu tabelu svih učitanih struktura FCB (*inode*)
- ❖ Proveravaju se prava pristupa na osnovu podešavanja zapisanih u FCB i podataka iz konteksta procesa (*uid*, *gid*), kako je ranije opisano
- ❖ Otvara se nov ulaz u tabeli otvorenih fajlova za dati proces (*struct file*) i u njoj postavljaju prava pristupa za operacije koje je proces najavio po otvaranju, a tekuća pozicija (*pomeraj*, *offset*) postavlja na 0. Inkrementira se broj referenciranja datog FCB
- ❖ Zauzima se prvi slobodan ulaz u tabeli deskriptora otvorenih fajlova datog procesa i kao rezultat sistemskog poziva vraća indeks tog ulaza. Proces koji je izdao ovaj sistemski poziv može da nastavi izvršavanje

Mart 2020.

Copyright 2020 by Dragan Milićev



51

Organizacija fajl podsistema

- ❖ Na ovaj način, fajl podsistem u kernelu može da implementira svaki sistemski poziv kao jedinstven postupak, odnosno redosled koraka, pri čemu su ti koraci pozivi polimorfnih operacija konkretnih objekata (OO pristup)
- ❖ Na primer, scenario izvršavanja sistemskog poziva za otvaranje fajla (*open*) može da izgleda ovako (pojednostavljeno, bez detalja vezanih za modalitete otvaranja i kreiranja, zaključavanje, obradu grešaka i drugo):

```
int fd = open("y/z", ...);
```
- ❖ Ukoliko je staza data argumentom apsolutna (počinje sa /), ta staza se uzima kao apsolutna staza do fajla. Ako je data staza relativna, iz konteksta tekućeg procesa uzima se staza do tekućeg direktorijuma procesa (neka je ona npr. `"/x/"`) i dodaje kao prefiks date relativne staze, čime se dobija puna staza do fajla, ovde: `"/x/y/z"`
- ❖ U strukturi keširanih ulaza u direktorijumima traži se postojanje ulaza sa punom stazom `"/x/y/z";` ukoliko takav postoji, on će odmah vratiti identifikator ili adresu FCB (*inode*) tog ulaza u tabeli koja čuva sve ove strukture, jer se tom čvoru već pristupalo. Ako to nije slučaj, mora se pronaći FCB ovog čvora, analizom strukture direktorijuma, npr. na sledeći način

Mart 2020.

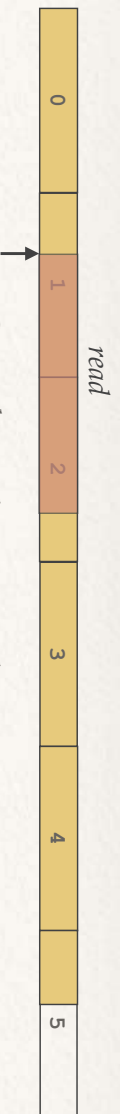
Copyright 2020 by Dragan Milićev

50

Organizacija fajl podsistema

- ❖ Princip izvršavanja operacija sa već otvorenim fajlom biće ilustrovan na primeru operacije čitanja (pojednostavljeno, bez nekih detalja kao što je obrada grešaka, obrada situacije čitanja iza kraja sadržaja i slično):

```
int status = read(fd, buf, count);
```
- ❖ Iz opisa otvorenog fajla (ulaza u tabeli opisa otvorenih fajlova) na kog ukazuje ulaz u tabeli deskriptora zadat prvim parametrom *fd*, proveriti se pravo za izvršenje ove operacije i ukoliko je ona zabranjena, vraća se odgovarajući kod greške
- ❖ Iz istog ulaza pročitati se tekuća pozicija u fajlu (pomerač, *offset*)
- ❖ Na osnovu ove tekuće pozicije, broja bajtova koje treba pročitati, kao i veličine bloka u fajl sistemu, odrede se logički brojevi blokova sadržaja fajla koje treba pročitati
- ❖ Za svaki od tih blokova, na osnovu podataka o načinu alokacije sadržaja fajla koji se nalaze u strukturi FCB (*inode*), odrede se brojevi fizičkih blokova na disku koje treba pročitati
- ❖ Svaki od tih blokova traži se u kešu, a ako nije tamo, učitava se korišćenjem operacije drajvera za čitanje bloka; učitani blok se smesti u keš blokova
- ❖ Iz učitanih blokova se deo sadržaja koji je proces tražio iskopira u bafer zadat drugim parametrom, tekuća pozicija postavi na odgovarajući vrednost, i procesu vrati kontrola i rezultat



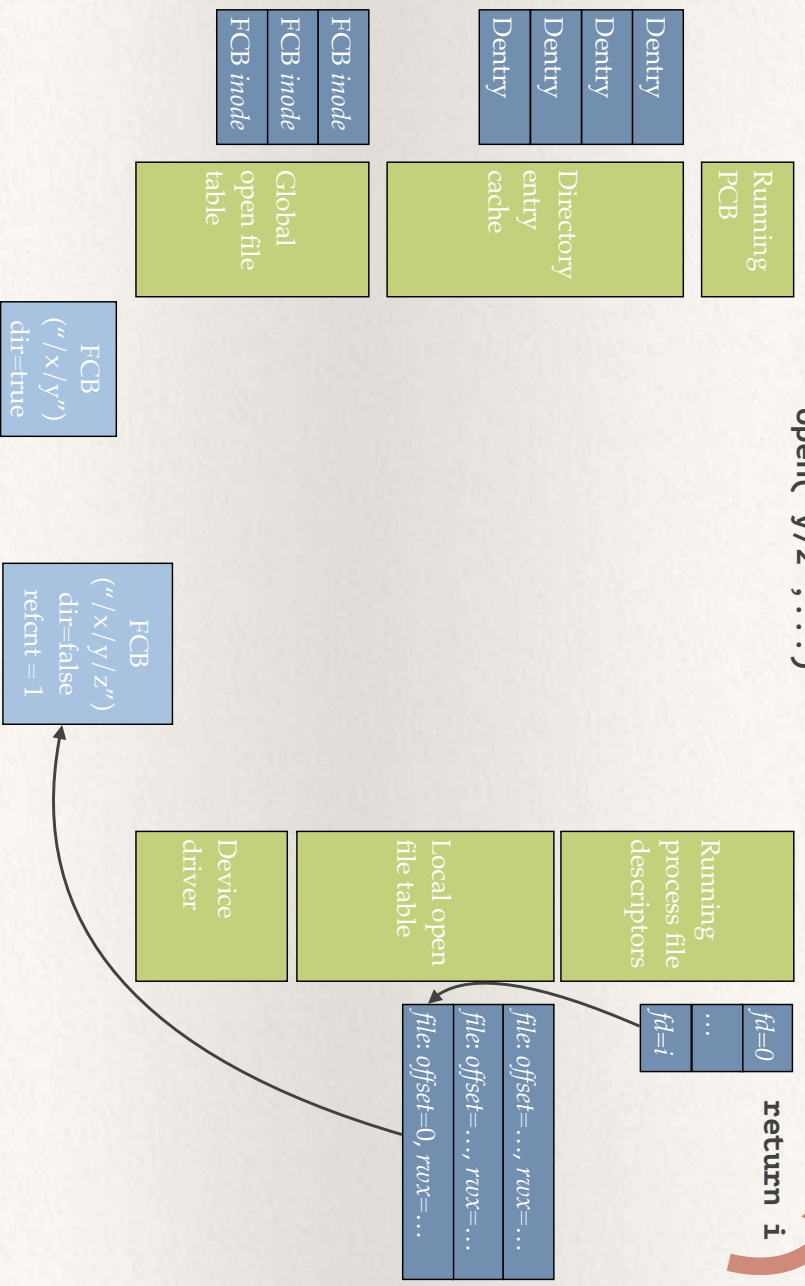
Mart 2020.

Copyright 2020 by Dragan Milićev

Organizacija fajl podsistema

`open("y/z", ...)`

`return i`



Mart 2020.

Copyright 2020 by Dragan Milićev

Implementacija direktorijuma

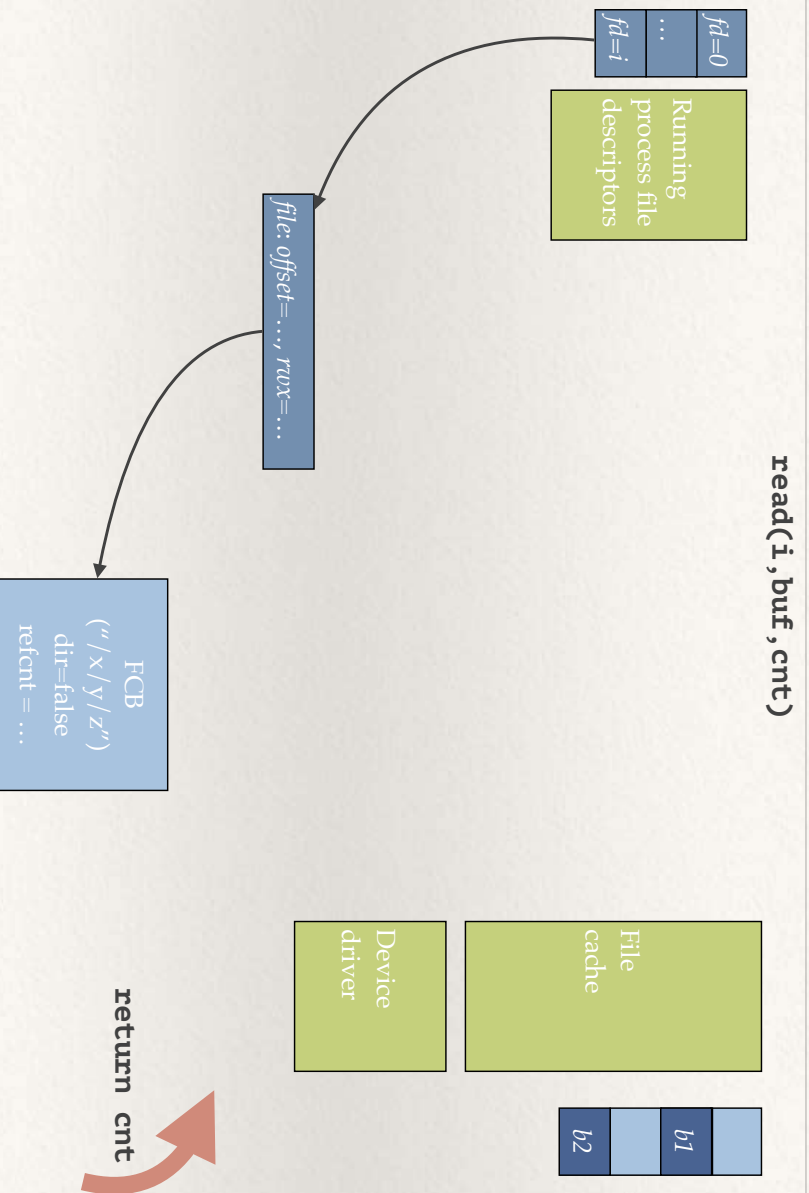
- ❖ Kao što je rečeno, direktorijumi se mogu implementirati posebnim strukturama, različito od fajlova (npr. Windows), dok se kod sistema nalik sistemu Unix, i direktorijumi i fajlovi smeštaju potpuno isto i zato nazivaju jedinstvenim imenom *čvor*: i jedni i drugi imaju svoj FCB (*inode*) i svoj sadržaj; jedan bit u FCB ukazuje na to da li čvor predstavlja direktorijum ili fajl
- ❖ Direktorijum preslikava simboličko ime u FCB, ali i to u šta tačno preslikava ime može da varira:
 - ❖ FCB se može ugraditi kao struktura u sam sadržaj direktorijuma, tako da direktorijum može da preslika ime neposredno u FCB; iako je teorijski moguće, ovo se ne radi, tada ne bi bile moguće višestruke tvrde veze na isti čvor
 - ❖ direktorijum može preslikati ime u adresu (broj bloka ili broj ulaza u globalnoj tabeli u kojoj su sve strukture FCB na volumenu) na kojoj se nalazi FCB na volumenu
 - ❖ direktorijum može preslikati ime u identifikator strukture FCB, a onda se adresa (broj bloka) gde se nalazi FCB dobija iz neke centralizovane strukture (mape), globalne za ceo volumen, preslikavanjem tog identifikatora

Mart 2020.

Copyright 2020 by Dragan Milićev

55

Organizacija fajl podsistema



Mart 2020.

Copyright 2020 by Dragan Milićev

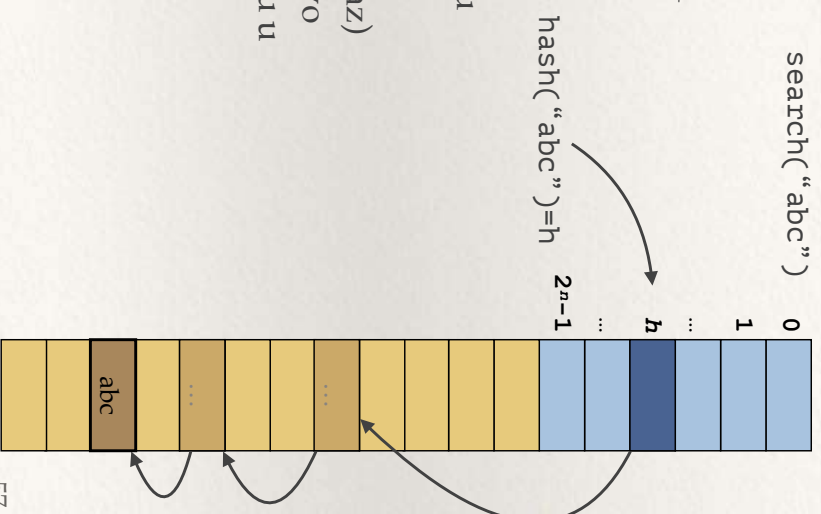
54

Implementacija direktorijuma

- ❖ Jedna takva efikasna i često korišćena struktura jeste heš tabela (*hash table*), jer je cilj upravo zapisati preslikavanje ogromnog domena ključeva (nizova znakova) u vrednosti (adrese FCB), pri čemu je skup ključeva koji se realno pojavljuju u preslikavanju mali podskup celog domena:
 - ❖ heš tabela ima 2^n ulaza
 - ❖ heš funkcija preslikava simboličko ime (niz znakova) u broj ulaza u heš tabeli
 - ❖ kolizije (pojava da se više ključeva preslikava u isti ulaz) rešavaju se ulančavanjem: svaki ulaz u tabeli je zapravo glava liste elemenata direktorijuma koji se preslikavaju u isti ulaz heš tabele
- ❖ Sada je pretraga i dalje linearna, jer se vrši u listi zapisa u jednom ulazu, ali su te liste značajno kraće (i do 2^n puta, ukoliko heš funkcija ravnomerno rasipa ključeve)

Mart 2020.

Copyright 2020 by Dragan Milićev



57

Implementacija direktorijuma

- ❖ Sledeće pitanje jeste to kako tačno implementirati strukturu podataka (mapu) kojom se realizuje preslikavanje simboličkog imena (niza znakova) u (adresu) FCB
- ❖ U suštini, bilo koja struktura podataka koja se može iskoristiti za implementaciju preslikavanja (mape) je sasvim prihvatljiva
- ❖ Sama struktura može se zapisati u sadržaj čvora koji predstavlja direktorijum
 - ❖ Jedan pristup je pomoću ulančane liste:
 - ❖ svaki ulaz u direktorijumu je jedan element liste
 - ❖ jednostavno je za implementaciju
 - ❖ može biti veoma neefikasno, jer pretraga simboličkog imena zahteva prolazak kroz celu listu, a radi se u mnogim operacijama sa direktorijumom (npr. potrebna je i pri promeni imena ali i pri kreiranju novog ulaza, radi provere jedinstvenosti novog imena)
- ❖ Moguće su i druge, naprednije strukture: sortirana lista, stablo, B-stablo

Mart 2020.

Copyright 2020 by Dragan Milićev

56

Metode alokacije fajla

- ❖ Kontinualna alokacija:
 - ❖ sadržaj fajla zauzima susedne blokove na disku (blokove sa susednim brojem)
 - ❖ FCB sadrži samo informaciju o prvom (početnom) bloku (*first_block*), kao i informaciju o veličini sadržaja u bajtovima, na osnovu čega se može izračunati i broj blokova koje zauzima
- ❖ Pogodnost:
 - ❖ jednostavan i efikasan sekvencijalan pristup
 - ❖ jednostavan i efikasan direktan pristup: logički blok broj k nalazi se na bloku broju $first_block + k$
- ❖ Problemi:
 - ❖ potrebno je pronaći slobodan prostor odgovarajuće veličine za smeštanje fajla odgovarajućim algoritmom (npr. *first fit*, *best fit*)
 - ❖ da bi se sadržaj fajla alocirao, neophodno je znati njegovu veličinu *pri samom kreiranju*, što nije uobičajeno u praksi, ili odvojiti neki procenjen prostor
 - ❖ proširenje sadržaja fajla može biti jednostavno, ako iza njega ima slobodnih blokova, ili može zahtevati relokaciju fajla ako tog prostora nema
 - ❖ postoji eksterna fragmentacija
- ❖ Problem eksterne fragmentacije može se rešavati kompakcijom. Kod starijih sistema, kompakcija je zahtevala da sistem ne bude operativan, kod novijih se može vršiti i tokom operisanja

Mart 2020.

Copyright 2020 by Dragan Milićev

59

Metode alokacije fajla

- ❖ Sledeće važno pitanje jeste to kako i gde, odnosno u koje blokove alocirati sam sadržaj fajla
- ❖ Pogodnost je to što blokovski uređaji omogućavaju direktan pristup bilo kom bloku, pa se, u principu, sadržaj fajla može smestiti u bilo koje blokove
- ❖ Kako se alociraju uvek celi blokovi za sadržaj koji pripada jednom fajlu, interna fragmentacija je neizbežna. Ona se može smanjiti manjim blokovima, ali su onda ulazno-izlazne operacije manje efikasne
- ❖ Polazna informacija o tome gde je smešten sadržaj fajla zapisuje se u strukturi FCB tog fajla
- ❖ Na raspolaganju su brojni načini alokacije, od kojih svaki ima neke svoje prednosti i nedostatke, i to tako da je ponekad neki način alokacije pogodniji za neke načine pristupa fajlu, a manje pogodan za neke druge. Viste pristupa fajlu jesu:
 - ❖ sekvencijalan pristup: karakterističan za znakovne tokove, odnosno za fajlove kojima se tekuća pozicija ne pomena eksplicitno (operacijom *seek*), već samo implicitno, tako da se sadržaju fajla pristupa redom
 - ❖ direktan pristup: pristupa se sadržaju u proizvoljnom redosledu (eksplicitne operacije *seek*)
- ❖ Ovde će biti prikazano samo nekoliko najčešće primenjivanih metoda alokacije:

- ❖ kontinualna alokacija
- ❖ ulančana alokacija
- ❖ indeksna alokacija

FCB:
first_block = 0x0d
size = 5 blocks

0	1	2	3
4	5	6	7
8	9	A	B
C	D	E	F
10	11	12	13
14	15	16	17
18	19	1A	1B

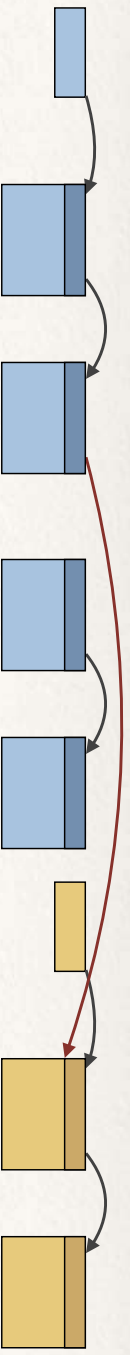
Mart 2020.

Copyright 2020 by Dragan Milićev

58

Metode alokacije fajla

- ❖ Problemi:
 - ❖ izuzetno neefikasan direktan pristup bloku: da bi se pristupilo logičkom bloku broj k sadržaja fajla, mora se pristupiti *svim prethodnim blokovima* redom, počev od prvog
 - ❖ u bloku se odvajaja prostor za smeštanje pokazivača (broja narednog bloka u listi); kod velikih diskova, sa velikim brojem blokova, i ovaj prostor može da bude nezanemarljiv u odnosu na ukupnu veličinu bloka
 - ❖ zbog greške ili oštećenja u zapisu na bloku, sadržaj fajla u tom bloku može biti oštećen (korumpiran, *corrupted*) ili potpuno izgubljen, dok je ostatak sadržaja tog fajla izgubljen i nedostupan; još gore, zbog slučajne vrednosti upisane u pokazivač, sadržaj fajla može da se nastavi listom koja pripada sadržaju drugog fajla, čime dolazi do kompromitovanja podataka
- ❖ Jedan načini rešavanja problema režijske potrošnje prostora za pokazivače jeste povećanje veličine bloka, ili alokacija više susednih blokova kao jedinice alokacije, tzv. *klastera* (*cluster* - grozd, svežanj). Međutim, povećanjem jedinice alokacije povećava se interna fragmentacija
- ❖ Problem korupcije pokazivača može se rešiti upisivanjem dodatnih, redundantnih informacija u blokove; ove informacije upisuje OS kada održava strukturu fajla, ali i proverava kada pristupa blokovima. Ako pri pristupu bloku naide na blok koji ne pripada fajlu kom se pristupa, detektuje se korupcija i pristup sprečava, a problem eventualno rešava. Na primer, u blok se mogu upisati sledeće redundantne informacije (ne bi bile potrebne da otkazi nisu mogući):
 - ❖ identifikator FCB kom pripada blok sa sadržajem
 - ❖ dvostruki pokazivači na naredni blok u listi; u ispravnom stanju, njihove vrednosti su jednake; kod korupcije, njihove vrednosti će vrlo verovatno biti različite, čime se detektuje greška



Mart 2020.

Copyright 2020 by Dragan Milićev

61

Metode alokacije fajla

❖ Ulančana alokacija (*linked allocation*):

- ❖ sadržaj fajla organizovan je kao ulančana lista blokova koji su proizvoljno raspoređeni na disku

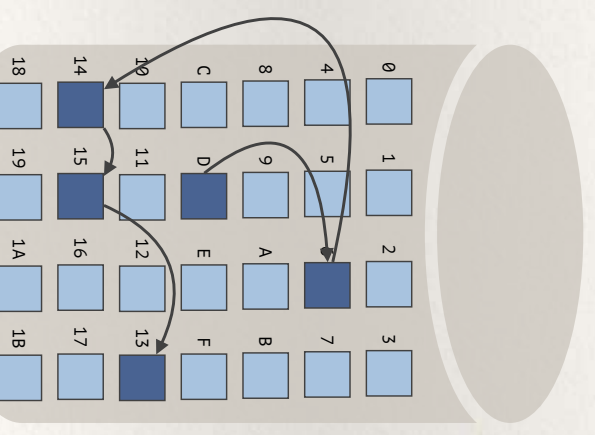
- ❖ FCB sadrži glavu liste — broj prvog bloka u listi

- ❖ svaki blok, osim sadržaja, sadrži i broj narednog bloka u listi

❖ Pogodnosti — rešava probleme kontinualne alokacije:

- ❖ jednostavan i efikasan sekvencijalan pristup
- ❖ blok se može alocirati bilo gde, nema potrebe za posebnim algoritmom (osim zbog optimizacije)
- ❖ fajl se može proširivati proizvoljno, dokle god ima slobodnih blokova
- ❖ ne mora se znati veličina sadržaja fajla pri kreiranju, kreiranje je jednostavno
- ❖ nema eksterne fragmentacije

FCB:
head_block = 0x0d
size = 5 blocks



Mart 2020.

Copyright 2020 by Dragan Milićev

60

Metode alokacije fajla

- ❖ Osnovni nedostatak ove alokacije jeste velika osetljivost na otkaze. Kod osnovne varijante ulančane alokacije, greška u jednom bloku dovodila je do ograničene štete, samo u jednom fajlu. Međutim, greška ili potpuni otkaz bloka na kom se nalazi FAT ili njen deo dovodi do otkaza celog fajl sistema
- ❖ Kako se FAT upotrebljavao u prošlosti, kada su diskovi bili osetljiviji i nisu imali ugrađene mehanizme tolerancije otkaza (otpornosti na otkaze) kakve imaju danas, ovakvi otkazi su bili veoma česti i nije bila retkost da ceo fajl sistem na disku postane neupotrebljiv
- ❖ Za oporavak od ovakvih otkaza postojali su programi koji su pokušavali da rekonstruišu sadržaj, ali je njihov učinak uvek bio ograničen i neizvesan
- ❖ Jedna mogućnost za rešavanje ovog problema jeste postojanje dvostruke kopije FAT, ali tada operacije postaju manje efikasne zbog ažuriranja dvostrukih kopija, a više prostora na disku i u operativnoj memoriji troši se na kopije FAT
- ❖ Noviji sistemi više ne upotrebljavaju ovu tehniku za hard diskove na kojima su “glavni” fajl sistemi, iako se ona i dalje koristi za manje, prenosive i eksterne uređaje

Mart 2020.

Copyright 2020 by Dragan Milićev

63

Metode alokacije fajla

- ❖ Jedna varijanta ulančane alokacije koja je bila izuzetno popularna (MS DOS, Windows, OS 2), a i dalje je u upotrebi, najviše na eksternim uređajima (eksterni diskovi, fleš memorije) jeste tzv. *File Allocation Table* (FAT):
 - ❖ na posebnom, unapred definisanom mestu na volumenu nalazi se tabela zvana FAT, ova tabela sadrži po jedan ulaz za svaki blok u prostoru za smeštaj fajlova i preslikava, jedan-na-jedan, svoje ulaze u te blokove
 - ❖ ulazi u FAT su ulančani, dok se na blokovima nalazi samo sadržaj; zapravo je sadržaj fajla i dalje ulančan, samo što su pokazivači i sadržaj razdvojeni i smeštaju se na različita mesta, a između njih postoji 1-1 preslikavanje (prethodnu sliku samo treba drugačije interpretirati, kao prikaz same FAT, a ne blokova sa sadržajem)
 - ❖ FAT se po pravilu kešira u memoriji, ili cela, ili svojim velikim delom, pa se operacije pristupa, koje i dalje podrazumevaju sekvencijalan pristup kroz elemente ulančane liste, obavljaju pristupom operativnoj memoriji, a ne blokovima na disku, što je značajno efikasnije
- ❖ Pogodnosti:
 - ❖ jednostavni algoritmi, kao i kod osnovne ulančane varijante
 - ❖ efikasan je i direktan pristup, iako je složenosti $O(n)$, podrazumeva pristupe operativnoj memoriji ako je FAT keširana

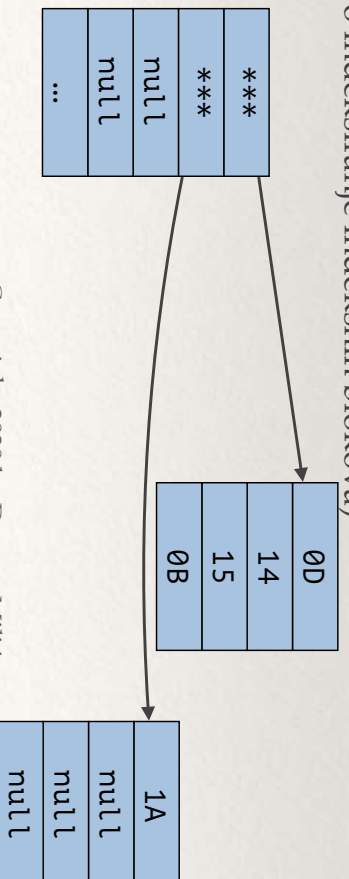
Mart 2020.

Copyright 2020 by Dragan Milićev

62

Metode alokacije fajla

- ❖ Osnovni nedostaci ove alokacije jesu potrošnja dodatnog režijskog prostora za smeštanje indeksa, kao i ograničenje veličine sadržaja fajla brojem ulaza u indeksu
- ❖ Da bi se smanjio udeo režijskog prostora za indekse, mogu se praviti manji indeksi, odnosno blokovi, ali to onda znači i manje dozvoljene veličine fajlova
- ❖ Mogući su različiti pristupi za rešavanje problema male ili ograničene veličine fajla:
 - ❖ sam indeks se može organizovati kao ulančana lista blokova, tako da se indeks može dinamički proširivati
 - ❖ indeks se može organizovati u više nivoa, kao stablo, poput PMT u više nivoa (zapravo indeksiranje indeksnih blokova)



Mart 2020.

Copyright 2020 by Dragan Milićev

65

Metode alokacije fajla

- ❖ Indeksna alokacija (*indexed allocation*):
 - ❖ svaki fajl ima svoj indeks koji se nalazi u posebnom bloku ili nekoliko susednih blokova (fiksni i unapred definisan broj)
 - ❖ broj indeksnog bloka zapisan je u FCB fajla
 - ❖ u indeksnom bloku je spisak (vektor) brojeva blokova sa sadržajem fajla, redom, poput tabele (npr. kao PMT)
- ❖ Pogodnosti:

0D	14	15	0B	1A	-1	-1	...
----	----	----	----	----	----	----	-----

- ❖ jednostavan je i efikasan direktan pristup (analogno straničenju), pa samim tim je efikasan i sekvencijalan pristup
- ❖ jednostavni su i efikasni algoritmi za ostale operacije sa fajlom (kreiranje, proširenje)
- ❖ blok se može alocirati bilo gde
- ❖ nije tako osetljiv na otkaze kao FAT
- ❖ većina pogodnosti ulančanog pristupa, samo što je direktan pristup efikasniji

FCB:
index_block = 0x06
size = 5 blocks

0	1	2	3				
4	5	6	7				
8	9	A	B				
C	D	E	F				
10	11	12	13				
14	15	16	17				
18	19	1A	1B				

Mart 2020.

Copyright 2020 by Dragan Milićev

64

Metode alokacije fajla

- ❖ Svaka metoda alokacije ima svoje prednosti i svoje nedostatke, i pogodna je za neke načine pristupa ili veličine fajlova, dok je za one druge manje pogodna ili sasvim nepodgovna
- ❖ Na primer, za male fajlove najzgodnija je kontinualna alokacija, jer nema potrošnje dodatnog prostora, a direktan pristup je izuzetno efikasan, dok je za veće fajlove pogodniji neki drugi, npr. indeksirani način sistema
- ❖ Od metode alokacije fajla, kao i od načina pristupa do sadržaja fajla umnogome zavise performanse sistema
- ❖ Zato neki sistemi omogućavaju čak i različite načine alokacije fajlova, na primer na sledeće načine:
 - ❖ prilikom kreiranja fajla, omogućavaju da se najavi kakav će pristup biti korišćen za taj fajl ili koja će maksimalna veličina sadržaja fajla biti, pa na osnovu toga odlučuju o tome koji metod alokacije će upotrebiti za fajl
 - ❖ dozvoljavaju i naknadnu konverziju fajla, tako što jedan način alokacije promene u drugi i reorganizuju strukturu sadržaja fajla; ovo se može zahtevati eksplicitno, ili čak implicitno, kada sam OS zaključi da se veličina fajla ili način pristupa do njega promenio, pa se odlučuje za drugačiji metod alokacije

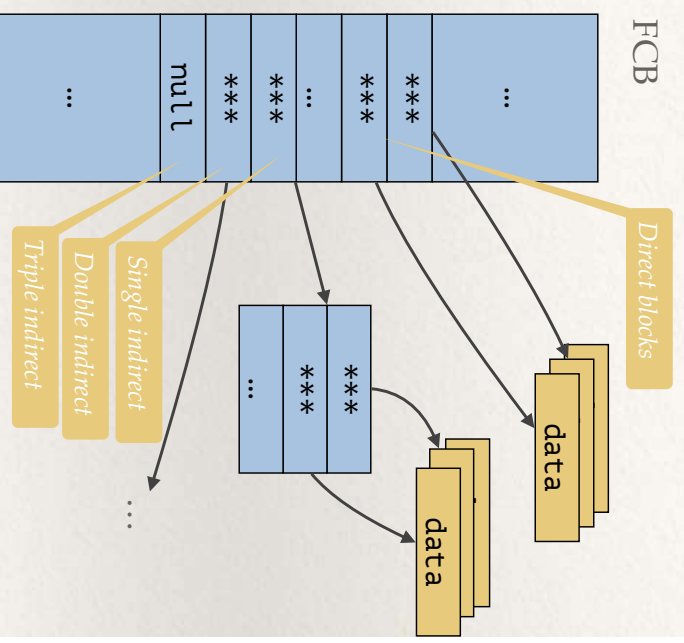
Mart 2020.

Copyright 2020 by Dragan Milićev

67

Metode alokacije fajla

- ❖ Sistemi nalik sistemu Unix primenjuju *kombinovani* varijantu, sa indeksom u jednom, dva i više nivoa:
 - ❖ u samom FCB nalazi se *indeks nultog nivoa* — određen broj ulaza koji sadrže adrese blokova sa neposrednim sadržajem; dok je sadržaj fajla dovoljno mali, koristi se samo ovaj indeks
 - ❖ kada sadržaj fajla poraste preko kapaciteta nultog indeksa, u FCB postoji ulaz koji sadrži broj bloka sa indeksom prvog nivoa (*single indirect*), u kom je spisak blokova sa sadržajem
 - ❖ i tako dalje, čak do indeksa trećeg nivoa
- ❖ Na ovaj način mali fajlovi ne zauzimaju nikakav dodatan prostor za indkse, a pristup do sadržaja je efikasan. Indekсни blokovi zauzimaju sve više prostora kako raste veličina sadržaja fajla, ali je udeo tog prostora u odnosu na veličinu sadržaja jako mali
- ❖ Osim toga, iako je veličina sadržaja fajla i dalje ograničena, to ograničenje je izuzetno veliko, tako da je praktično nebitno za većinu fajlova



Mart 2020.

Copyright 2020 by Dragan Milićev

66

Rukovanje slobodnim prostorom

- ❖ Bez ozbira na način alokacije fajlova, i nezavisno od tog načina, OS mora da vodi evidenciju o slobodnom prostoru, odnosno o slobodnim blokovima na disku:
 - ❖ kada je potrebno alocirati jedan ili više blokova, traže se blokovi iz ove evidencije i izbacuju iz nje
 - ❖ kada se obriše sadržaj fajla ili ceo fajl, oslobođeni blokovi se dodaju u ovu evidenciju
- ❖ Jedan često primenjivan pristup evidenciji slobodnog prostora jeste *bit vektor* (*bit vector*):
 - ❖ svakom bloku koji je predviđen za alokaciju fajlova pridruži se bit u nizu bajtova; preslikavanje broja bloka u tom skupu u bit u vektoru i obratno je krajnje jednostavno i efikasno (izvesti računicu!)
 - ❖ ovaj bit vektor se zapisuje na određenom, unapred definisanom mestu na volumenu, ali se svakako kešira u operativnoj memoriji
 - ❖ alokacija bloka svodi se na pretragu bita sa vrednošću 1 (ili 0) u vektoru; savremeni procesori poseduju instrukcije koje kao rezultat vraćaju broj razreda u bajtu ili reči u kom se nalazi prva jedinica / nula sleva ili zdesna, pa ova pretraga može biti izuzetno brza; slično je za dealokaciju bloka
 - ❖ ovo rešenje je pogodno jer je jednostavno, efikasno, a ne zahteva nikakve upise u same oslobođene blokove, kao neke druge metode
 - ❖ nepogodnost je to što je za smeštanje bit vektora potreban poseban prostor; na primer, za prostor od 1 TB i blokom veličine 512 B potrebno je 2 G bita, odnosno 256 MB za bit vektor

Mart 2020.

Copyright 2020 by Dragan Milićev

69

Metode alokacije fajla

- ❖ Čak i ako ne primenjuju kontinualnu alokaciju, pa ne zahtevaju da logički susedni blokovi sadržaja fajla budu i fizički susedni na disku, svi današnji sistemi se trude da logički susedne blokove sadržaja fajla alociraju i kao fizički susedne, ili makar tako da su bliski po adresi, kad god je to moguće, kako bi optimizovali pristup susednim blokovima kod pristupa sadržaju fajla (manje pomeranja glave diska)
- ❖ Sa ciljem ovakve optimizacije, operativni sistem, ili poseban program koji je za to namenjen, može vršiti premeštanje blokova sa sadržajem fajla, kako bi ih smestio što bliže jedan drugom. Ranije su ovakve operacije mogle da se rade samo dok sistem nije u upotrebi, sada mogu da se rade i dok sistem operiše
- ❖ Operativni sistemu primenjuju i mnoge druge fine i sofisticirane tehnike za organizaciju fajlova i efikasan pristup do njih

Mart 2020.

Copyright 2020 by Dragan Milićev

68

Efikasnost i performanse

- ❖ Fajl sistem, koji svakako značajno opterećuje ulazno-izlazni podsistem sa diskom, je izuzetno osetljiv deo sistema i može da postane usko grlo u funkcionisanju sistema. Zato je potrebna njegova pažljiva konstrukcija i optimizacija, a operativni sistemi primenjuju mnoge napredne tehnike za unapređenje njegove efikasnosti. Primeri tehnika:
 - ❖ alokacija susednih blokova za sadržaj istog fajla kako bi vreme pristupa bilo što kraće: čak i kada metod alokacije to ne zahteva, OS se uvek trudi da alocira susedne ili što bliže blokove na disku za sadržaj istog fajla, kao i da sadržaj fajla smesti što bliže bloku na kom je FCB tog fajla, kako bi se glava diska manje pomerala pri pristupu istom fajlu
 - ❖ prealokacija struktura FCB na disku: unapred alocirati ove strukture na volumenu kako bi kasnije operacije bile efikasnije
 - ❖ upotreba klastera različite veličine radi smanjenja interne fragmentacije: npr. za male fajlove i za kraj sadržaja fajla se upotrebljava manji klaster, dok se za sadržaj većih fajlova upotrebljava veći klaster; tako se veličina internog fragmenta drži srazmernom korisnom sadržaju, a ulazno-izlazne operacije čine efikasnijim

Mart 2020.

Copyright 2020 by Dragan Milićev

71

Rukovanje slobodnim prostorom

- ❖ Druga mogućnost jeste ulančavanje slobodnih blokova, kao kod ulančane alokacije fajlova:
 - ❖ slobodni blokovi se ulančavaju u listu, a glava liste je globalan podatak za ceo volumen
 - ❖ alokacija jednog bloka je efikasna: uzima se prvi blok sa liste, a glava pomera na sledeći (potrebno je učitavanje tog bloka); alokacija više blokova je neefikasna, jer zahteva učitavanje tih blokova radi očitavanja pokazivača
 - ❖ dealokacija je efikasna kod ulančane alokacije sadržaja fajla, jer se lanac blokova iz fajla može samo prevezati u lanac slobodnih blokova, što je efikasno ako FCB čuva i pokazivač na poslednji blok u lancu
 - ❖ rukovanje slobodnim prostorom je inherentno podržano u FAT varijanti: ulazi u FAT koji se odnose na slobodne blokove se mogu jednostavno ulančavati kao i za fajlove, ili mogu biti označeni posebnim *null* vrednostima
- ❖ Moguće su i razne druge kompaktnije varijante, na primer:
 - ❖ grupisanje slobodnih blokova, tako da prvi blok u grupi slobodnih sadrži spisak (indeks) narednih *n* slobodnih blokova iz te grupe, kao i pokazivač na sledeći takav indeksni blok naredne grupe
 - ❖ kako se često alociraju i dealociraju fizički susedni blokovi, ceo jedan kontinualan segment od nekoliko susednih slobodnih blokova može da se posmatra kao jedan element liste, tako da prvi blok u tom segmentu sadrži broj tih blokova i pokazivač na sledeći takav segment u listi

Mart 2020.

Copyright 2020 by Dragan Milićev

70

Otpornost na otkaze

- ❖ Svaka iole složenija operacija upisa u fajl po pravilu zahteva izmenu više blokova na disku, npr. i bloka u kom je FCB i više blokova sa indeksima ili sadržajem fajla. Ove operacije obavljaju se u kešu u operativnoj memoriji, a potom se tako izmenjeni blokovi u kešu snimaju na disk
- ❖ Šta će se dogoditi ako se samo deo tako izmenjenih blokova snimi, a onda nestane napajanja ili računar otkaze iz nekog drugog razloga? Strukture podataka fajl sistema na disku mogu da ostanu u nekonzistentnom stanju, korumpirane
- ❖ Ranije, kada fajl sistemi, pa i hardver (diskovi i računari) nisu imali ugrađene mehanizme otpornosti na otkaze, ovakve situacije bile su veoma česte: nestanak napajanja ili slučajno isključivanje ili reset računara (čak i zbog otkaza u softveru) bio je veoma riskantan, jer je postojala velika verovatnoća da će barem deo fajl sistema, najmanje fajl sa kojim je trenutno radio neki proces, ostati oštećen
- ❖ Za ove namene postojali su posebni sistemski programi (npr. Unix *fsck*, MS DOS *chkdsk*) koji su se pokretali i pokušavali da uoče oštećene strukture fajl sistema i poprave ih. Njihov učinak bio je zavisen od veličine oštećenja: od potpune rekonstrukcije, do nepovratnog gubitka sadržaja jednog ili više fajlova
- ❖ Današnji računari imaju ugrađene hardverske mehanizme koji ih čine otpornijim na ovakve otkaze, npr. baterijsko napajanje računara, kondenzatori koji omogućavaju napajanje disk kontrolera do završetka snimanja blokova iz njegovog keša i slično
- ❖ Osim toga, i operativni sistemi imaju složenije mehanizme koji ih čine tolerantnim na ovakve otkaze. Jedna mogućost, koja samo umanjuje veličinu štete kod ovakvih otkaza, jeste sinhrono upisivanje metapodataka na disk

Mart 2020.

Copyright 2020 by Dragan Milićev

73

Efikasnost i performanse

❖ Tehnike (nastavak):

- ❖ keširanje je praktično neizostavno
- ❖ za sekvencijalni pristup, npr. kada se fajl otvori sa najavom sekvencijalnog pristupa, ili kada OS sam prepozna da proces pristupa fajlu sekvencijalno, učitava blokove unapred, kako bi bili spremni kad budu potrebni (*read-ahead*)
- ❖ asinhroni upis podataka: operacija upisa se završi u kešu, procesu odmah vrati kontrola kako bi nastavio svoje izvršavanje, a podaci naknadno snime na disk; izuzetak mogu biti operacije promene metapodataka (samih struktura fajl sistema), kako bi se smanjila šteta pri eventualnom otkazu
- ❖ upotreba dinamičkih struktura neograničene dimenzije umesto ograničenih struktura sa statičkim dimenzijama: raniji fajl sistemi su bili orijentisani na statički alocirane i limitirane strukture podataka, pa su nametali mnoga ograničenja, počev od malog broja znakova u imenu fajla (tipično osam), pa do broja ulaza u direktorijum ili veličine sadržaja fajla; današnji sistemi orijentisani su na dinamičku alokaciju struktura, pa ili nemaju ovakva ograničenja, ili su ona veoma velika

Mart 2020.

Copyright 2020 by Dragan Milićev

72

Otpornost na otkaze

- ❖ Sistem vođenja zapisnika ovo obezbeđuje na sledeći način (principijelan i pojednostavljen opis):
 - ❖ sistem vodi tzv. *zapisnik* (*journal*) kao jednostavnu linearnu strukturu prostih zapisa na samom disku, na unapred definisanom mestu, npr. u jednom određenom bloku; pri svakom upisu jednog zapisa u zapisnik, taj blok se snima na disk sinhrono
 - ❖ na početku transakcije, u prazan zapisnik se upiše zapis da je transakcija započeta (`<start>`)
 - ❖ kada se završi promena sadržaja bilo kog bloka u kešu, bilo za metapodatke ili sadržaj fajla, i taj blok treba da se upiše na disk, taj blok se upiše na neko drugo mesto, u neki alociran slobodan blok, dok se originalni blok ne menja; u zapisnik se doda nov jednostavan zapis koji sadži adresu (broj) originalnog bloka i njegove nove verzije (`<oldBlock# , newBlock#>`)
 - ❖ kada se operacija završi do kraja, u zapisnik se upiše da je transakcija završena (`<commit>`)
 - ❖ potom se jedan po jedan blok prepisuje iz privremene kopije na originalno mesto; kada se sve ovo završi, zapisnik se obriše i operacija je završena uspešno

Mart 2020.

Copyright 2020 by Dragan Milićev

75

Otpornost na otkaze

- ❖ Jedna veoma raširena tehnika za otpornost fajl sistema na ovakve otkaze jeste tzv. *vođenje dnevnika* ili *zapisnika* (*journaling*)
- ❖ U suštini, radi se o transakcionom mehanizmu ugrađenom u fajl sistem, poput onih koje koriste sistemi za upravljanje bazama podataka
- ❖ Svaka operacija izmene fajla koju proces zahteva, npr. sistemska usluga upisa u fajl *write*, posmatra se kao transakcija i sistem obezbeđuje njenu *atomičnost*, u smislu da će efekat na perzistentne strukture na disku biti uvek garantovano ili kao da operacija nije ni započeta, ili kao da je završena u celini (nikada izmenjene samo delimično)

Mart 2020.

Copyright 2020 by Dragan Milićev

74

Otpornost na otkaze

- ❖ Bez obzira na ove hardverske i softverske mehanizme otpornosti na otkaze, otkazi se i dalje mogu dogoditi: jednostavno, hardver diska može da nepovratno otkaze (pokvari se) tako da sadržaj na disku ostaje nepovratno izgubljen
- ❖ Zbog ovoga se snažno preporučuje redovno pravljenje *rezervnih kopija* fajl sistema (*backup*): svakako je uvek više nego pametno sačuvati rezultat sopstvenog rada od nekoliko sati ili dana, kao i sve drugo što je teško ili zahtevno, dugotrajno restaurirati ako postane izgubljeno
- ❖ Danas ove stvari nije teško uraditi, jer postoje mnogi lako dostupni, jeftini i brzi načini pravljenja rezervne kopije: na eksternom disku, fleš memoriji, drugom računaru na mreži ili nekom servisu u oblaku na Internetu. Važno je samo da važan sadržaj *nikada ne bude samo na jednom uređaju*, jer tada uvek lako može postati nepovratno izgubljen

Mart 2020.

Copyright 2020 by Dragan Milićev

77

Otpornost na otkaze

- ❖ Ako se tokom izvršavanja operacije, a pre nego što transakcija upiše zapis `<commit>` u zapisnik, dogodi otkaz računara, u zapisniku će ostati samo zapisi delimično izvršene transakcije. Kada se OS ponovo podigne, on najpre gleda sadržaj zapisnika. Pošto u njemu nema zapisa `<commit>`, transakcija se odbacuje, a svi blokovi koji su čuvali pripremljene nove verzije se prosto proglašavaju slobodnim; zapisnik se briše. Na taj način, fajl sistem ostaje neizmenjen, kao da transakcija nikad nije ni započela
- ❖ Ako se otkaz dogodi nakon upisa zapisa `<commit>`, a pre nego što su svi novi blokovi prepisani u originale, pa zapisnik nije obrisan, OS će, po ponovnom podizanju, pronaći ovaj zapis u zapisniku i krenuti da prepisuje blokove ispočetka, kao i pri normalnom završetku transakcije; kako su ove operacije prepisivanja blokova idempotentne, nije nikakav problem ako su neki blokovi već bili prepisani. Kada to završi, OS briše zapisnik, tako da je efekat na fajl sistem kao da je transakcija završena u celini
- ❖ U opštem slučaju, više transakcija se može raditi uporedo i za svaku se vode zapisi u zapisniku, pri čemu zapis sadrži i identifikator transakcije

Mart 2020.

Copyright 2020 by Dragan Milićev

76

Otpornost na otkaze

- ❖ Procedura pravljenja rezervne kopije (*backup*) može biti:
 - ❖ totalna: kopiraju se svi fajlovi iz jednog fajl sistema ili njegovog dela (direktorijuma), nezavisno od toga da li su izmenjeni u odnosu na verzije na postojećoj kopiji, pa se takvi neizmenjeni fajlovi bespotrebno ponovo kopiraju
 - ❖ inkrementalna (*incremental*): kopiraju se samo izmenjeni fajlovi; ovo, naravno, može biti značajno brže
- ❖ Neki današnji operativni sistemi imaju ugrađenu podršku za pravljenje rezervne kopije, pa čak i čuvanje prethodnih verzija svih fajlova
- ❖ Na primer, Mac OS X ima tzv. *time machine*: fajl sistem čuva sve ranije verzije svakog fajla, sve dok na disku ima prostora, a onda one najstarije verzije automatski briše. Moguće je “ući u vremeplov” i restaurirati neku raniju verziju nekog fajla. Osim toga, sistem podižava i inkrementalan bekap, koji se pokreće eksplicitno ili čak i automatski, ako je na računar priključen eksterni disk koji je namenjen samo za tu svrhu i posebno formatizovan