

dr Dragan Milićev

Redovni profesor, Elektrotehnički fakultet u Beogradu

[dmilicev@etf.rs](mailto:dmilicev@etf.rs), [www.rcub.bg.ac.rs/~dmilicev](http://www.rcub.bg.ac.rs/~dmilicev)

# Operativni sistemi 2

Viši kurs

# Sadržaj

- 1 Uvod – o predmetu
- 2 Raspoređivanje procesa
- 3 Sinhronizacija i komunikacija između procesa
- 4 Upravljanje deljenim resursima
- 5 Virtuelna memorija
- 6 Upravljanje diskovima
- 7 Arhitektura operativnih sistema

# Sadržaj

- 8 Primer operativnog sistema: Linux
- 9 Primer operativnog sistema: Windows API
- 10 Primer operativnog sistema: Android
- 11 Zaključak

# Glava 1: Uvod - o predmetu

Sadržaj, ciljevi i preduslovi

Organizacija nastave

Praktičan rad

Kolokvijumi

Ispit

Literatura

Kontakti

# Sadržaj, ciljevi i preduslovi

## ◆ Sadržaj:

- Nastavak predmeta OS1: napredniji koncepti, algoritmi, problemi i rešenja operativnih sistema
- Primeri konstrukcije popularnih sistema

## ◆ Ciljevi:

- Upoznati se sa širim spektrom koncepata i principa konstrukcije operativnih sistema
- Steći potpuno znanje primenjivo na operativne sisteme uopšte, nevezano ni za jedan konkretan sistem
- Upoznati se sa principima konstrukcije konkretnih, realnih operativnih sistema

# Sadržaj, ciljevi i preduslovi

## ◆ Preduslovi:

- Svi preduslovi za OS1
- Dobro savladano gradivo predmeta OS1 i položen taj ispit!
- **Samostalan, praktičan i kontinuiran rad:  
operativni sistemi se najbolje mogu razumeti  
konstruišući sopstveni!**

# Organizacija nastave i praktičan rad

◆ Predavanja: 2 časa nedeljno

◆ Vežbe:

- 2 časa nedeljno
- zadaci za razumevanje koncepata i algoritama
- diskusija, demonstracije i konsultacije

◆ Praktičan samostalan rad:

- obavezan domaći zadatak
- student samostalno izgrađuje i integriše delove jednog malog, ali potpuno funkcionalnog operativnog sistema
- rade se i brane samostalno, usmeno i na računaru
- ulaze u konačnu ocenu sa 30%

# Kolokvijumi

- ◆ 3 kolokvijuma
- ◆ ukupno nose 40 poena: 15+15+10 poena
- ◆ složeniji, obimniji, konstruktivni zadaci
- ◆ rade se sa dozvoljenom literaturom
- ◆ svaki kolokvijum traje 1,5 sat
- ◆ može se nadoknaditi bilo koji od prva dva kolokvijuma (jedan ili oba), ali samo u jednom terminu godišnje za nadoknadu (prvi ispitni rok – ili nadoknada kolokvijuma ili ispit, ali ne oba)
- ◆ za prolaz je potrebno u zbiru prikupiti bar 40% poena sa kolokvijuma

Kol. #	SI	IR
1	U redovnom terminu 1. kol. SI	U redovnom terminu 2. kol. SI
2	U redovnom terminu 2. kol. SI	U redovnom terminu 2. kol. SI
3	U redovnom terminu jan. roka	U redovnom terminu jan. roka

# Ispit

## ◆ Pismeni ispit:

- pre pismenog ispita se mora predati kompletno urađen domaći zadatak
- može se polagati u svakom zvaničnom ispitnom roku
- ukupno nosi 30 poena
- kratka pitanja, jednostavni zadaci za proveru osnovnog znanja i razumevanja gradiva
- traje 1,5 sat
- radi se bez dozvoljene literature i bilo kakvih pomagala
- za prolaz je potrebno prikupiti bar 50% poena sa pismenog dela

## ◆ Usmeni ispit:

- odbrana domaćih zadataka
- teorijska i praktična pitanja i usmeni odgovori

# Literatura

## ◆ Referentne knjige:

- Silberschatz, A., Galvin, P., Gagne, G.: "Operating System Concepts," 7th ed., John Wiley and Sons  
[www.os-book.com](http://www.os-book.com)  
[www.wiley.com/college/silberschatz](http://www.wiley.com/college/silberschatz)
- B. Đorđević, D. Pleskonjić, N. Maček, "Operativni sistemi – koncepti", Viša elektrotehnička škola Beograd, 2004.

## ◆ Univerzitetski kursevi:

- Stanford University: [www.stanford.edu/class/cs140/](http://www.stanford.edu/class/cs140/)
- University of Washington:  
[www.cs.washington.edu/education/courses/451/](http://www.cs.washington.edu/education/courses/451/)

# Kontakti

◆ Sajt predmeta:

<http://os.etf.rs>

◆ Nastavnik: Dragan Milićev

[dmilicev@etf.rs](mailto:dmilicev@etf.rs)

[www.rcub.bg.ac.rs/~dmilicev](http://www.rcub.bg.ac.rs/~dmilicev)

◆ Asistent: Živojin Šuštran

[zika@etf.rs](mailto:zika@etf.rs)

# Glava 2: Rasporedjivanje procesa

Osnovni koncepti

Kriterijumi rasporedjivanja procesa

Algoritmi rasporedjivanja procesa

Rasporedjivanje procesa na multiprocesorima

# Osnovni koncepti

- ◆ Svrha multiprogramiranja: obezbediti da se u svakom trenutku izvršava neki proces
- ◆ Svrha raspodele vremena (*time sharing*): vršiti promenu konteksta dovoljno često tako da korisnici mogu da interaguju sa svakim programom koji se izvršava
- ◆ Na jednoprocесорском систему, увек се извршава највиše један процес (*running*); остали који су спремни за извршавање чекају свој ред у реду спремних процеса
- ◆ *Raspoređivač procesa (process scheduler)* има задатак да се наведени циљеви испуне: када се менја контекст, из скупа спремних процеса изабрати један за извршавање  
**Scheduler::put(runningThread) ;**  
**runningThread = Scheduler::get() ;**
- ◆ Кoji? – Алгоритам распоређивања

# Osnovni koncepti

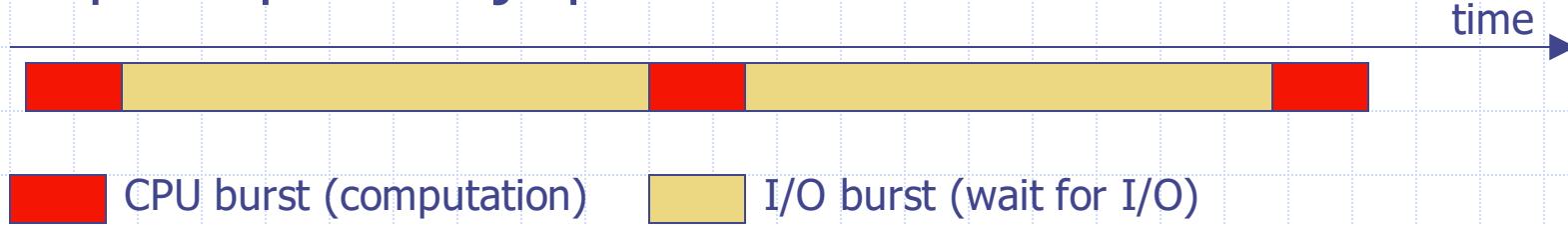
- ◆ Kod paketnih (*batch*) sistema više zadatih poslova čeka na disku:
  - *dugoročni raspoređivač* (*long-term scheduler*) ili *raspoređivač poslova* (*job scheduler*) bira one poslove za koje će kreirati procese u memoriji
  - *kratkoročni raspoređivač* (*short-term scheduler*) ili *CPU raspoređivač* (*CPU scheduler*) iz skupa spremnih procesa bira jedan kome se dodeljuje procesor
- ◆ CPU raspoređivač se aktivira veoma često (reda svakih 10-100 ms) i mora da bude brz
- ◆ Raspoređivač poslova se aktivira retko. On održava *stepen multiprogramiranja* (*degree of mutliprogramming*) – broj procesa u memoriji. Ako je stepen multiprogramiranja stabilan, raspoređivač poslova se pokreće kada se neki proces gasi

# Osnovni koncepti

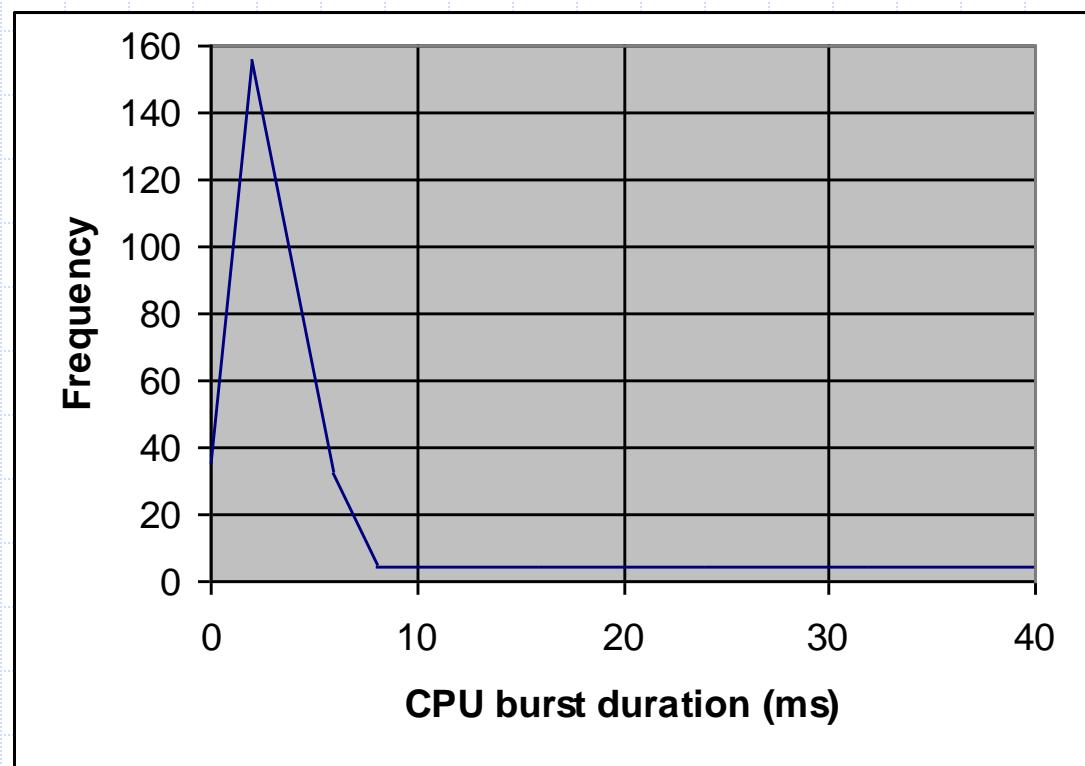
- ◆ Raspoređivač poslova mora da vodi računa o izbalansiranosti korišćenja sistemskih resursa od strane dve vrste procesa:
  - *CPU-bound* proces ima mnogo izračunavanja i retko poziva I/O operacije
  - *I/O bound* proces ima malo izračunavanja i često poziva I/O operacije
- ◆ Mnogi sistemi sa raspodelom vremena (npr. Unix, Windows) ne poseduju raspoređivač poslova
- ◆ Neki sistemi poseduju i *srednjoročni raspoređivač (medium-term scheduler)* koji bira procese za zamenu (*swapping*)
- ◆ Nadalje se uglavnom bavimo kratkoročnim (CPU) raspoređivanjem

# Osnovni koncepti

- ◆ Tipično ponašanje procesa:

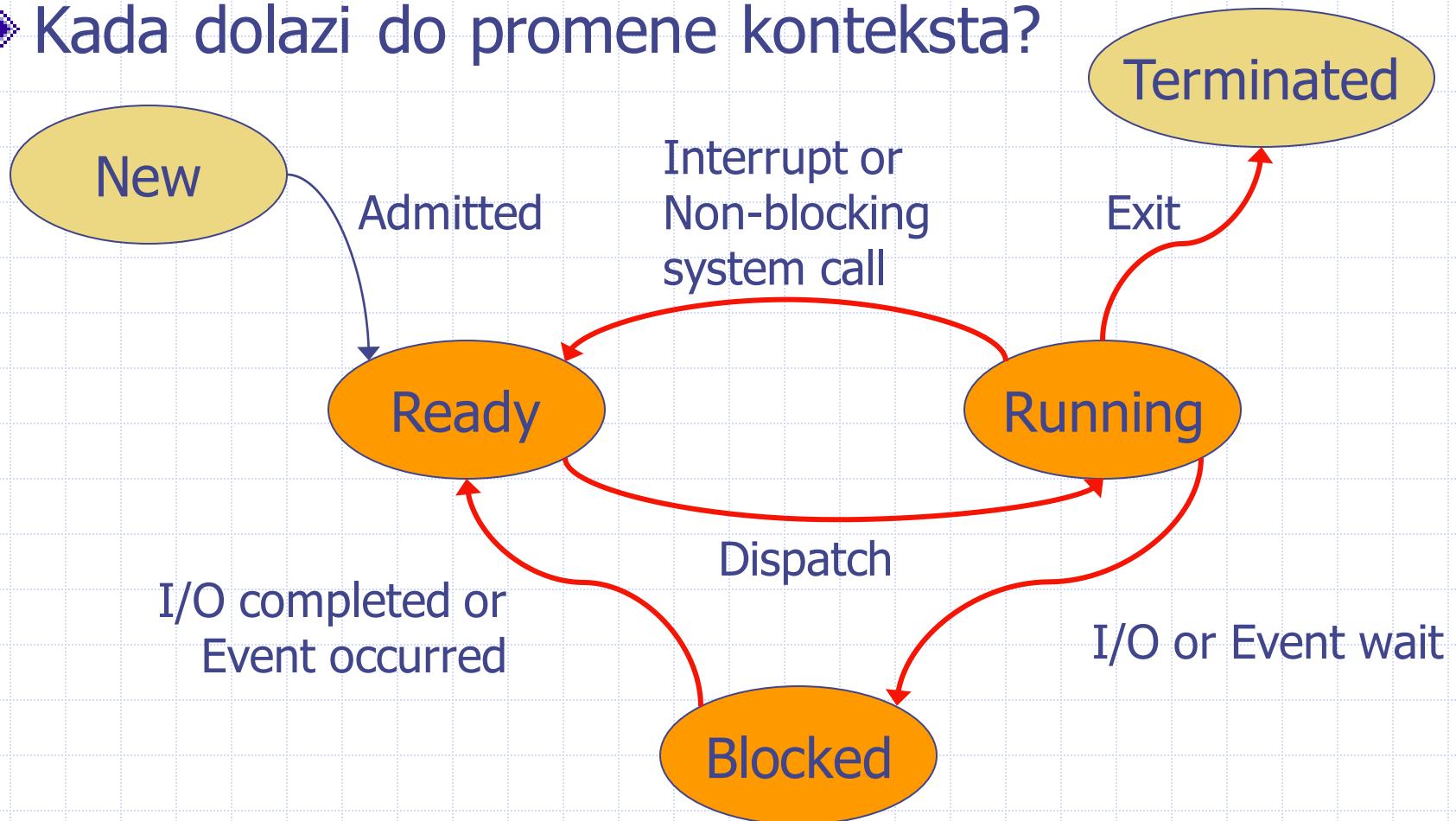


- ◆ Tipična raspodela *CPU-burst* procesa (empirijski):



# Osnovni koncepti

- ◆ Kada dolazi do promene konteksta?



# Osnovni koncepti

## ◆ Kada i kako proces (može da iz)gubi procesor (promena konteksta):

### ■ sinhrono:

- ◆ blokira se na sinhronizacionoj primitivi (eksplicitno):
  - čeka na međuprocesnu sinhronizaciju
  - čeka na I/O operaciju
  - čeka na istek zadatog vremena
- ◆ završava se (eksplicitno)
- ◆ izvršava neblokirajući sistemski poziv, ali OS dobija priliku da izvrši promenu konteksta (implicitno)
- ◆ napravi izuzetak (implicitno)

### ■ asinhrono (na prekid):

- ◆ zadovoljen uslov sinhronizacije, završena I/O operacija, isteklo vreme čekanja, isteklo vreme dodeljeno procesu (*time slice*)

# Osnovni koncepti

- ◆ Ako OS podržava samo sinhronu promenu konteksta, naziva se OS *bez preuzimanja (nonpreemptive)*
- ◆ Ako OS podržava i asinhronu promenu konteksta, naziva se OS *sa preuzimanjem (preemptive)*
- ◆ Windows 3.x – *nonpreemptive*; Windows 95+ - *preemptive*
- ◆ Mac OS X, Unix, Linux – *preemptive*
- ◆ Mnogo teže je realizovati *preemptive* nego *nonpreemptive* kernel! Zašto?
- ◆ Međutim, *preemptive* OS je efikasniji! Zašto?
- ◆ Dobro razlikovati:
  - promenu konteksta od raspoređivanja
  - algoritam (protokol) raspoređivanja od strukture podataka kojom se implementira red spremnih procesa

# Kriterijumi raspoređivanja

- ◆ Izbor algoritma raspoređivanja zavisi od prirode procesa. Svaki algoritam ima svoje karakteristike i pogodniji je za neke vrste procesa
- ◆ Kriterijumi poređenja algoritama:
  - Iskorišćenje procesora (*CPU utilization*)
  - Propusnost (*throughput*): broj završenih procesa u jedinici vremena
  - Ukupno vreme provedeno u sistemu (*turnaround time*): vreme koje protekne od kreiranja do gašenja procesa
  - Vreme čekanja (*waiting time*): vreme koje proces provede u redu spremnih procesa
  - Vreme odziva (*response time*) u interaktivnom sistemu: vreme koje protekne od korisničkog zahteva do odziva na taj zahtev
- ◆ Načini optimizacije:
  - optimizovati srednje vrednosti navedenih parametara
  - optimizovati maksimalne/minimalne vrednosti ovih parametara
  - minimizovati varijansu vrednosti: kod interaktivnih sistema, važnija je *predvidivost* odziva nego srednja vrednost njegovog odziva

# Algoritmi raspoređivanja - FCFS

- ◆ *First-Come, First Served* (FCFS): proces koji je prvi tražio CPU, prvi će ga i dobiti
- ◆ Najjednostavniji algoritam; jednostavna implementacija pomoću FIFO reda: iz reda spremnih uzima se prvi proces, a novi se stavlja na kraj
- ◆ Primer:

Proces: Vreme izvršavanja:

P1	24
P2	3
P3	3

Ako su aktivirani redom P1, P2, P3:



Vreme čekanja:  $W_1=0$ ,  $W_2=24$ ,  $W_3=24+3=27$ ,  $W_s=17$

Ako su aktivirani redom P2, P3, P1:  $W_2=0$ ,  $W_3=3$ ,  $W_1=3+3=6$ ,  $W_s=3$

# Algoritmi raspoređivanja - FCFS

- ◆ Zaključak: vreme čekanja (a time i vreme odziva) kod FCFS nije uvek minimalno i može jako da varira ako su vremena izvršavanja procesa značajno različita
- ◆ *Konvoj efekat (convoy effect)*: grupa I/O-bound procesa čeka da jedan CPU-bound proces završi svoje dugo izvršavanje i stalno tako u krug "ide za njim kao konvoj" iz reda spremnih u red čekanja na I/O – slabije iskorišćenje CPU i uređaja nego da procesi sa kraćim izvršavanjem idu prvi
- ◆ FCSF je podrazumevano *nonpreemptive* i zato nepogodan za sisteme sa raspodelom vremena

# Algoritmi raspoređivanja - SJF

- ◆ *Shortest-Job-First (SJF)*: svakom procesu pridružuje se vrednost dužine sledećeg izvršavanja (*CPU burst*); CPU se dodeljuje onom procesu koji ima najmanju ovu vrednost u redu spremnih

- ◆ Primer:

Proces:	Vreme izvršavanja:
P1	6
P2	8
P3	7
P4	3

Redosled izvršavanja prema SJF: P4, P1, P3, P2:



Vreme čekanja:  $W_1=3$ ,  $W_2=16$ ,  $W_3=9$ ,  $W_4=0$ ,  $W_s=7$

Redosled izvršavanja po FCFS: P1, P2, P3, P4:  $W_s=10,25$

# Algoritmi raspoređivanja - SJF

- ◆ Dokazivo je da je SJF *optimalan* u smislu da za dati skup procesa daje minimalno srednje vreme čekanja (izvesti dokaz!)
- ◆ Osnovni problem SJF algoritma: kako znati vrednost dužine narednog izvršavanja (*CPU burst*)?
- ◆ Kod paketne obrade, ova procena se ostavlja korisniku kao procena ograničenja vremena izvršavanja
- ◆ Kod kratkoročnog raspoređivanja, ovo se ne može *znati* unapred, može se samo *predvideti* (npr. na osnovu predistorije)
- ◆ Zbog ovog problema, iako je optimalan, SJF se upotrebljava samo kod dugoročnog raspoređivanja. Kod kratkoročnog (CPU) raspoređivanja, nije primenjiv egzaktno, već samo kao aproksimacija (na osnovu predviđenog vremena)

# Algoritmi raspoređivanja - SJF

- ◆ Način predviđanja eksponencijalnim usrednjavanjem ( $\tau_n$  - predviđeno vreme,  $t_n$  – stvarno vreme izvršavanja  $n$ ):

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n, \quad 0 \leq \alpha \leq 1$$

odnosno:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$

$\alpha=0$ : istorija nema značaja ( $\tau_{n+1} = \tau_n$ )

$\alpha=1$ : samo poslednje izvršavanje se uzima kao procena

tipično:  $\alpha=1/2$

# Algoritmi raspoređivanja - SJF

- ◆ SJF može biti i sa i bez preuzimanja. *Preemptive SJF* uzima u obzir procenu *preostalog (remaining)* vremena izvršavanja procesa koji je prekinut (*shortest remaining time*)

- ◆ Primer:

Proces:

P1

P2

P3

P4

Vreme aktivacije:

0

1

2

3

Vreme izvršavanja:

8

4

9

5



Prosečno vreme čekanja:  $W_s = 6,5$

Prosečno vreme čekanja za *nonpreemptive SJF* bilo bi 7,75

# Algoritmi raspoređivanja – *Priority Sch*

- ◆ *Raspoređivanje sa prioritetima (Priority Scheduling, PS)*: svakom procesu se dodeljuje vrednost *prioriteta*, a CPU se dodeljuje procesu sa najvišim prioritetom
- ◆ Prioritet je vrednost iz skupa sa parcijalnim uređenjem (može se porediti). Najjednostavnije – celobrojna vrednost
- ◆ Neki sistemi označavaju viši prioritet manjim celim brojem, a neki obrnuto. Ovde: manji broj – viši prioritet
- ◆ SJF je specijalni slučaj PS, pri čemu je prioritet jednak dužini narednog izvršavanja

# Algoritmi raspoređivanja – Priority Sch

## ◆ Primer:

Proces:

P1

P2

P3

P4

P5

Vreme izvršavanja:

10

1

2

1

5

Prioritet:

3

1

4

5

2



Srednje vreme čekanja: 8,2

# Algoritmi raspoređivanja – *Priority Sch*

- ◆ Prioriteti se procesima mogu dodeliti:
  - interno: sam OS dodeljuje vrednost neke ili nekih merljivih veličina kao prioritet; na primer: vremensko ograničenje, memorijski zahtevi, broj otvorenih fajlova, količnik prosečnog vremena I/O operacije i CPU izračunavanja
  - eksterno: zadaje se van OS, kao parametar kreiranja procesa; npr. važnost procesa, cena njegovog izvršavanja ili vremenska ograničenja (u RT sistemima)
- ◆ Raspoređivanje po prioritetima može biti i sa i bez preuzimanja
- ◆ Osnovni problem ovog raspoređivanja - *izgladnjivanje (starvation)*: proces koji je spreman nikada ne dolazi do procesora jer ga pretiču procesi višeg prioriteta
- ◆ Jedan način rešavanja problema izgladnjivanja – *starenje (aging)*: prioritet nekom procesu se postepeno povećava kako on duže čeka na izvršavanje, pa konačno dolazi na red

# Algoritmi raspoređivanja – RR

- ◆ *Round-robin (RR) raspoređivanje: preemptive FCFS, specijalno osmišljen za time sharing sisteme:*
  - svakom procesu se dodeljuje vremenski kvantum (*time slice*) za izvršavanje; tipično 10 do 100 ms
  - red spremnih procesa se tretira kao cirkularni red (FIFO)
  - raspoređivač ciklično dodeljuje procesor procesima u redu spremnih na izvršavanje do isteka vremenskog kvantuma (implicitno preuzimanje), ili dok se proces sam ne odrekne procesora ili se blokira (eksplicitno preuzimanje)
- ◆ Primer: vremenski kvantum od 4 jedinice vremena

Proces:	Vreme izvršavanja:
P1	24
P2	3
P3	3

Redosled izvršavanja: P1(4), P2(3), P3(3), P1(5\*4)

Srednje vreme čekanja:  $17/3 = 5,67$

# Algoritmi raspoređivanja – RR

- ◆ Uticaj veličine vremenskog kvantuma:
  - veoma veliki kvantum: RR postaje FCFS
  - veoma mali kvantum: povećava broj promena konteksta i time i režijsko vreme
- ◆ Okvirno empirijsko pravilo: 80% vremena izvršavanja (*CPU burst*) treba da bude kraće od vremenskog kvantuma

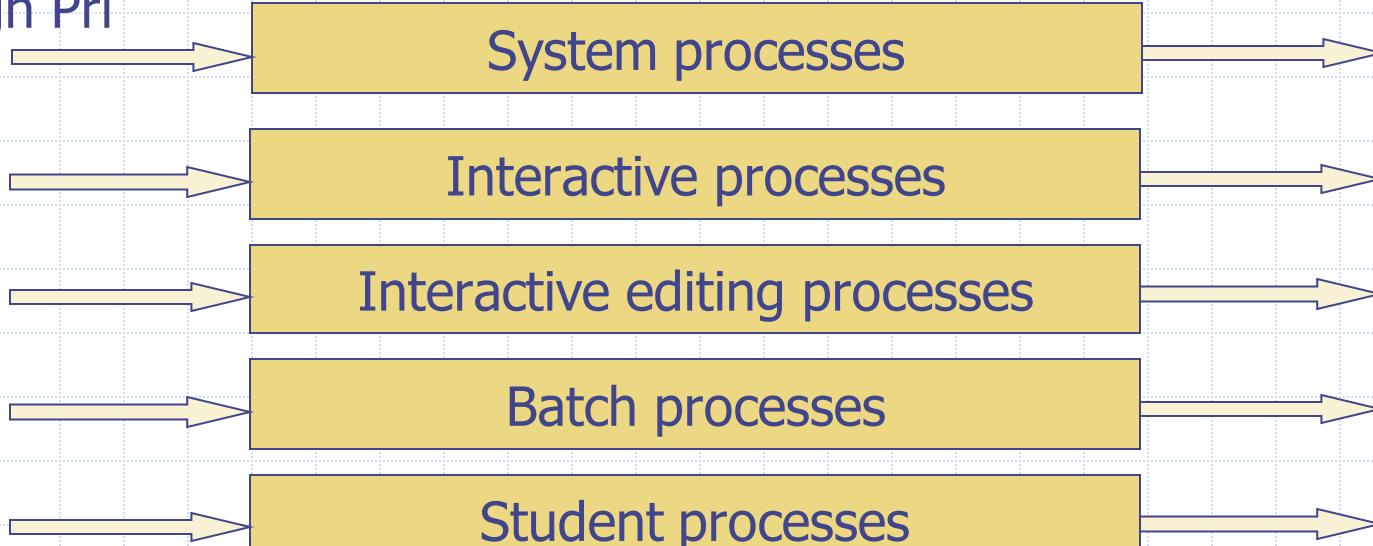
# Algoritmi raspoređivanja – MQS

- ◆ *Multilevel Queue Scheduling* (MQS): čitava klasa algoritama raspoređivanja kod kojih se procesi razvrstavaju u različite grupe različitih karakteristika, potreba i prioriteta; na primer, na interaktivne i pozadinske (paketne)
- ◆ Ideja:
  - postoji više redova spremnih procesa, prema klasifikaciji procesa
  - proces se smešta u odgovarajući red prema svojim karakteristikama (vrsta procesa ili parametri korišćenja resursa)
  - za svaki red se primenjuje poseban algoritam raspoređivanja
  - postoji i jedan globalni algoritam raspoređivanja između redova; uglavnom se sprovodi raspoređivanje po prioritetima i sa preuzimanjem (*fixed-priority preemptive scheduling*) ili se za svaki red dodeljuje odgovarajući vremenski kvantum izvršavanja

# Algoritmi rasporedivanja – MQS

## ◆ Primer:

High Pri

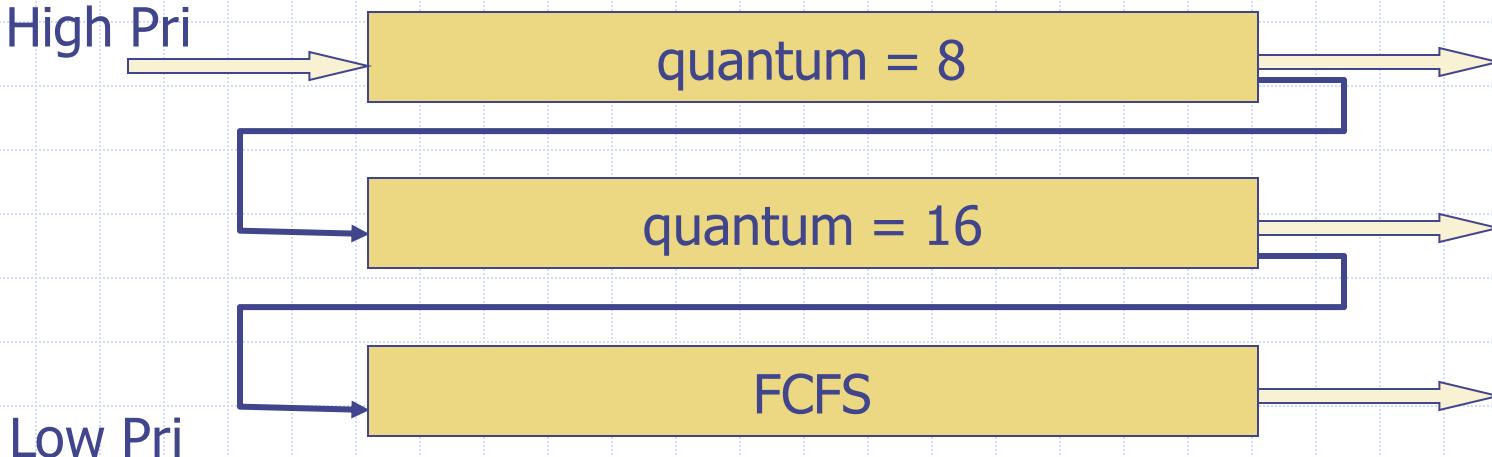


Low Pri

# Algoritmi raspoređivanja – MFQS

- ◆ *Multilevel Feedback-Queue Scheduling (MFQS)*: kao MQS, samo što procesi mogu da se premeštaju iz jednog reda spremnih u drugi:
  - ako je proces mnogo koristio CPU (istekao mu je vremenski kvantum), premešta se u red nižeg prioriteta
  - ako je proces dugo čekao u redu nižeg prioriteta, premešta se u red višeg prioriteta

◆ Primer:  
High Pri



# Algoritmi raspoređivanja – MFQS

## ◆ MFQS definiše sledeće:

- broj redova spremnih procesa
- algoritam raspoređivanja za svaki red
- metod koji se koristi za "unapređenje" procesa – premeštanje u red višeg prioriteta
- metod koji se koristi za "unazadjenje" procesa – premeštanje u red nižeg prioriteta
- u koji red se smešta novi spreman proces

# Algoritmi raspoređivanja – CFS

- ◆ *Potpuno pravedan raspoređivač* (*Completely Fair Scheduler, CFS*) ima za cilj da unapredi interaktivnost procesa, uz maksimalno iskorišćenje procesora
- ◆ Primjenjen u kernelu Linuxa počev od verzije 2.6.23 (iz 2007)
- ◆ Ideja je da se procesima dodeljuje vreme izvršavanja „najpravednije moguće“, kao da su svi ravnopravni na „idealnom“ procesoru i raspoređivaču:
  - za svaki proces prati se *ukupno vreme izvršavanja* u tom naletu (koliko vremena je imao procesor od kako je postao spremna, *execution time*); iz skupa spremnih bira se onaj koji ima minimalno ovo vreme
  - izabranom procesu dodeljuje se vremenski kvantum jednak *maksimalnom vremenu izvršavanja* (*maximum execution time*), koje je količnik vremena koje je proces proveo čekajući na procesor u redu spremnih i ukupnog broja procesa (vreme koje bi dobio pri „idealno ravnopravnom“ raspoređivanju, svima jedнако)
  - radi ubrzanja operacija, red spremnih procesa implementira se kao crveno-crno stablo (*red-black tree*), proces za izbor je prvi lev
- ◆ Nije „idealno pravedan“, postoje složeniji i „pravedniji“ algoritmi

# Algoritmi raspoređivanja – FPS i EDF

- ◆ U sistemima za rad u realnom vremenu (*real-time systems*, RT) „tvrde“ (*hard*) kategorije primenjuju se posebni algoritmi koji minimizuju neodređenost, odnosno omogućavaju proračun *rasporedivosti* (*schedulability*)
- ◆ Osnovni i najčešći algoritmi su algoritmi zasnovani na prioritetu (*priority scheduling*), ali se prioriteti procesima dodeljuju isključivo na osnovu *vremenskih karakteristika* (perioda i vremenski rok završetka)
- ◆ Najčešći algoritmi prioritiranja:
  - *Raspoređivanje po fiksnim prioritetima* (*Fixed Priority Scheduling*, FPS): prioriteti se procesima dodeljuju po vremenskim karakteristikama, tipično po učestanosti aktivacije periodičnih procesa ili po kratkoći vremenskog roka do završetka po aktivaciji
  - *Prvo onaj sa najkraćim rokom* (*Earliest Deadline First*): dinamički, pri svakom preuzimanju, procesor dobija onaj proces kome će najskorije isteći vremenski rok

# Algoritmi raspoređivanja – Grupno rasp.

- ◆ U interaktivnim sistemima, posebno serverskim sistemima sa više korisnika, za kvalitet usluge je važno da korisnici, tačnije njihovi procesi, ne „smetaju“ jedni drugima, odnosno da ne „otimaju“ procesorsko vreme; na primer, za neke RT ili interaktivne procese (reprodukcijski audio ili video snimka, proces koji interaguje preko tastature itd.) potrebno je obezbediti dobar odziv, tj. garantovano procesorsko vreme da bi napredovali bez „zapinjanja“
- ◆ To se može postići garantovanjem određene količine procesorskog vremena koje će određene *grupe* procesa dobiti kao kvant procesorskog vremena koje onda ti procesi dele između sebe
- ◆ Procesi se organizuju u grupe, koje mogu biti čak i hijerarhijski organizovane (u stablo): jedna grupa procesa može sadržati procese (listove stabla) ili podgrupe
- ◆ Na svakom nivou u hijerarhiji, vremenski kvantum dodeljen jednom čvoru deli se (npr. ravnomerno) na podčvorove, i tako rekursivno do listova kojima se dodeljuje procesor – *grupno raspoređivanje* (*group scheduling*)

# Algoritmi raspoređivanja

## ◆ Kako proceniti i izabrati odgovarajući algoritam?

- determinističkim modelovanjem: za dati skup procesa sa poznatim karakteristikama sprovesti analitičku evaluaciju i poređenje; samo za jednostavne slučajeve
- teorija i modeli redova čekanja (*queueing theory*); Litlova formula:

$$n = \lambda W$$

gde je:

$n$  – srednja dužina reda čekanja (broj poslova koji čekaju u redu)

$\lambda$  – srednja brzina pristizanja novih poslova u red

$W$  – srednje vreme čekanja u redu

- simulacijom
- implementacijom i posmatranjem realnog opterećenja

# Raspoređivanje na multiprocesorima

- ◆ Mnogo složenije nego na jednom procesoru (NP-kompletan problem)
- ◆ *Asimetričan sistem*: jedan procesor (*master*) izvršava samo kernel kod i vrši raspoređivanje, dok drugi procesori (*slaves*) izvršavaju samo korisnički kod
- ◆ *Simetričan sistem*: svi procesori su jednaki i ravnopravni i svaki vrši raspoređivanje za sebe; dve mogućnosti:
  - za svaki procesor postoji poseban red spremnih procesa: jednostavnije, ali je potrebno *balansiranje opterećenja* (*load balancing*) premeštanjem procesa između procesora
  - postoji jedan, globalni red spremnih procesa iz koga se uzimaju procesi za sve procesore: složenije, ali potencijalno bolji učinak
- ◆ *Afinititet (affinity)* procesa: da bi se smanjili penali zbog promašaja u procesorskom kešu, uvek je bolje da proces nastavi izvršavanje na istom procesoru na kom se prethodno izvršavao, pa raspoređivači i o ovome vode računa
- ◆ Jezgra (*core*) i hardverske niti (*thread*) OS u principu tretira kao fizičke procesore, osim što su penali u različitim situacijama različiti (npr. pri promašaju u kešu)

# Glava 3: Sinhronizacija i komunikacija između procesa

Uvod

Monitori

Razmena poruka

Priklučnice (*sockets*)

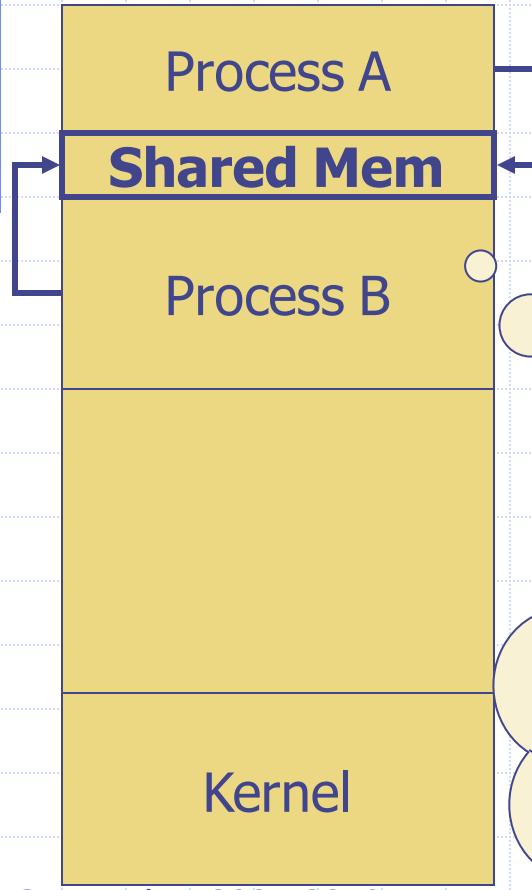
Poziv udaljene procedure (RPC)

# Uvod

- ◆ Modeli međuprocesne komunikacije (*Inter Process Communication, IPC*):
  - *deljena promenljiva (shared variable)*: objekat kome može pristupati više procesa; komunikacija se obavlja razmenom informacija preko deljene promenljive ili *deljenih podataka (shared data)*
  - *razmena poruka (message passing)*: eksplicitna razmena informacija između procesa u vidu poruka koje putuju od jednog do drugog procesa preko nekog posrednika
- ◆ Model komunikacije je stvar izbora – ne implicira način implementacije:
  - deljene promenljive je lako implementirati na multiprocesorima sa zajedničkom memorijom, ali se mogu (teže) implementirati i na distribuiranim sistemima
  - razmena poruka se može implementirati i na distribuiranim sistemima i na multiprocesorima sa deljenom memorijom
  - ista aplikacija se može isprogramirati korišćenjem oba modela, ali je po pravilu neki model pogodniji za neku vrstu aplikacije

# Uvod

Implementacija komunikacije pomoću deljene promenljive na sistemu za deljenom memorijom, ali sa procesima koji nemaju isti adresni prostor:

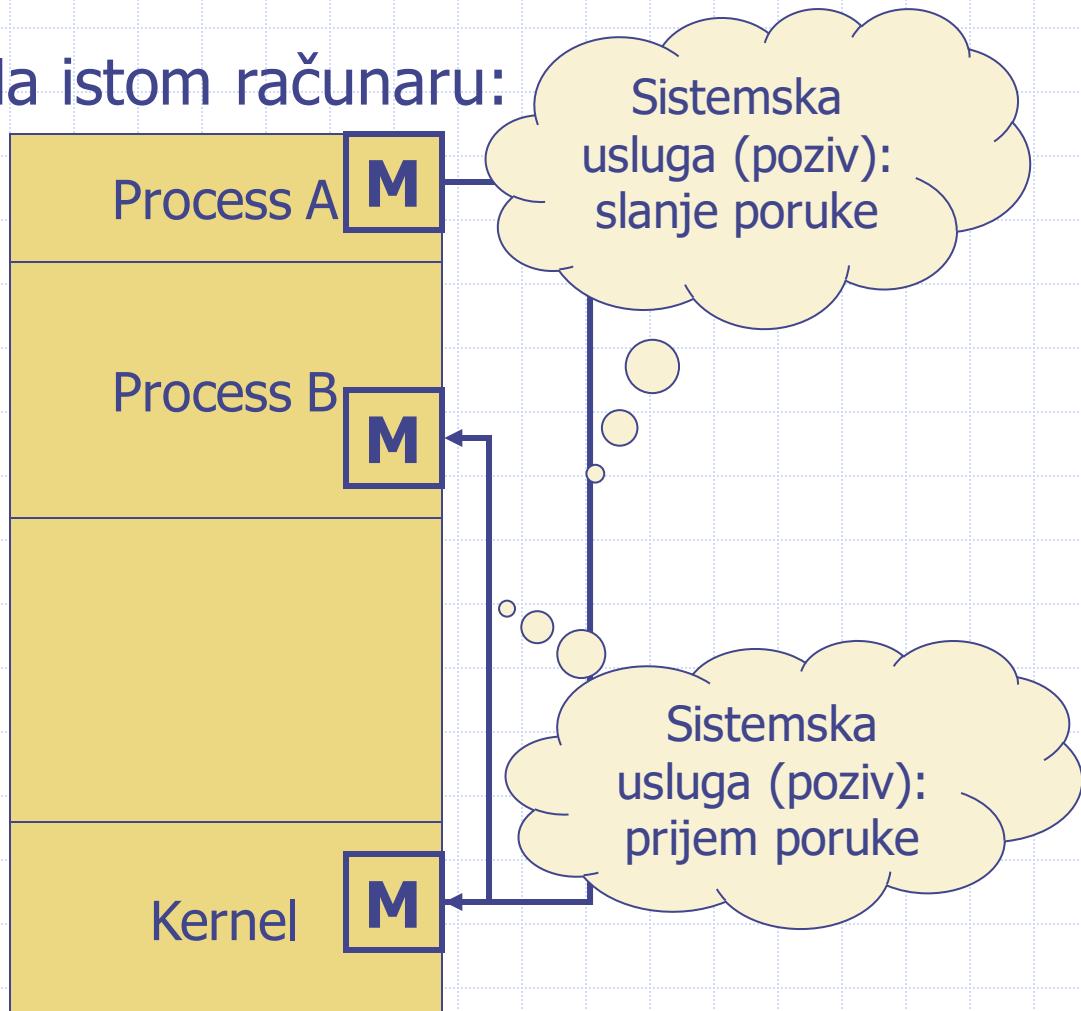


Sistemska usluga  
(poziv): obezbediti  
preslikavanje dela  
adresnog prostora  
dva procesa u istu  
deljenu memoriju

# Uvod

Implementacija komunikacije razmenom poruka:

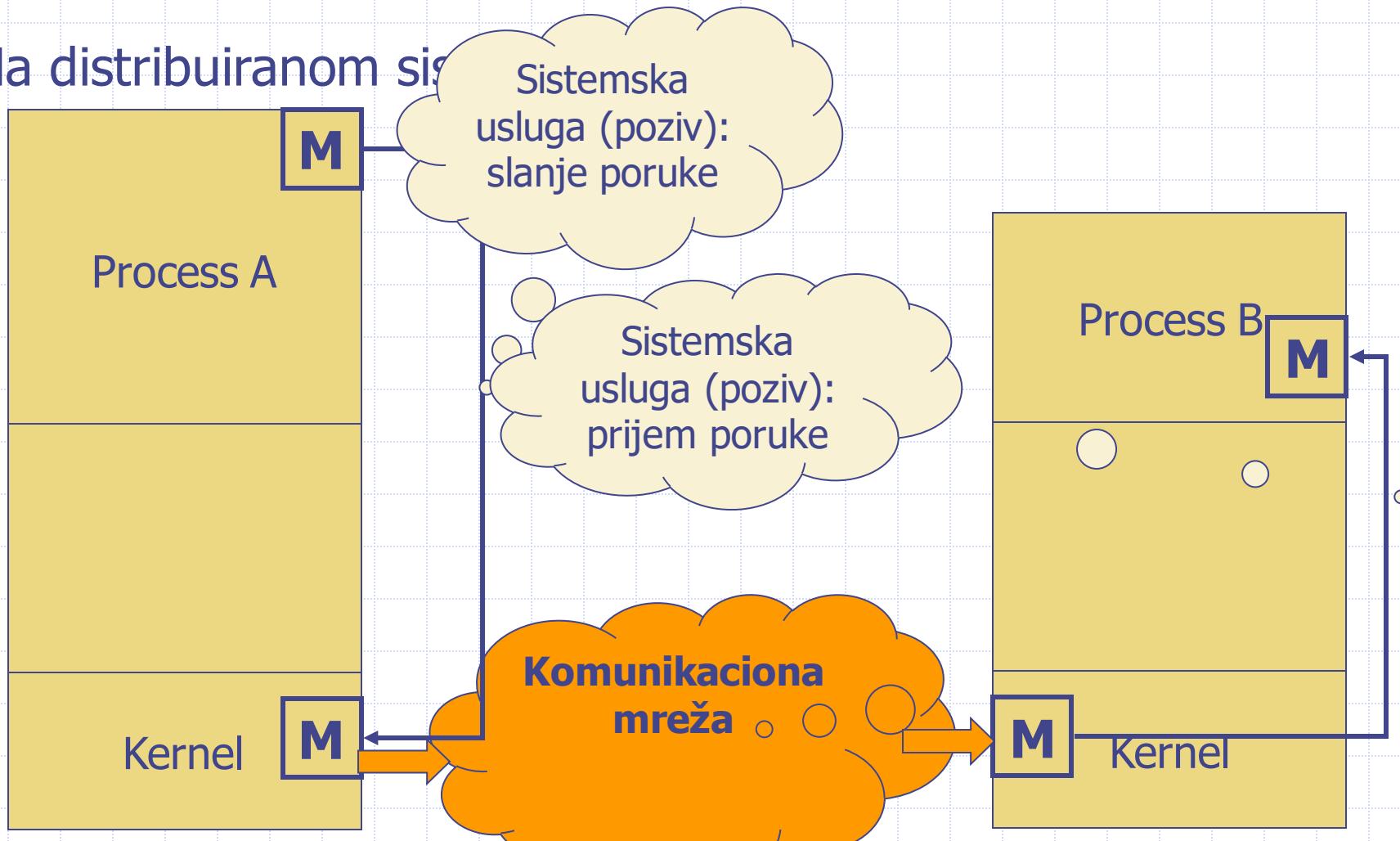
Na istom računaru:



# Uvod

Implementacija komunikacije razmenom poruka:

Na distribuiranom sistemu



# Monitori

## ◆ Loše strane semafora:

- suviše jednostavan koncept niskog nivoa – nije logički povezan sa konceptima bližim domenu problema (resurs, kritična sekcija, ...)
- kod složenijih programa lako postaje glomazan, nepregledan, težak za razumevanje, proveru i održavanje, jer su operacije nad semaforima rasute
- podložan je greškama – mora se paziti na uparenost i redosled operacija *wait* i *signal*

```
process P1 ;  
    wait(S1) ;  
    wait(S2) ;  
    ...  
    signal(S2) ;  
    signal(S1) ;  
end P1 ;
```

```
process P2 ;  
    wait(S2) ;  
    wait(S1) ;  
    ...  
    signal(S1) ;  
    signal(S2) ;  
end P2 ;
```

# Monitori

- ◆ *Monitor (monitor) je:*
  - apstraktni tip podataka koji grupiše strukturu (podatke, promenljive) i ponašanje (operacije, procedure nad tim podacima)
  - podrazumevano enkapsulira (sakriva kao privatne) svoje podatke, a otkriva (kao interfejs) svoje operacije (procedure)
  - operacije (procedure) su podrazumevano *međusobno isključive*
- ◆ Autori: Dijkstra (1968), Brinch-Hansen (1973) i Hoare (1974)
- ◆ Jezici koji podržavaju monitore: Modula 1, Concurrent Pascal, Mesa, Ada, Java, C#

# Monitori

- ◆ Primer: ograničeni bafer (*bounded buffer*)

```
monitor buffer;
  export append, take;
  var ... (* Declaration of necessary variables *)

  procedure append (i : integer);
    ...
  end;

  procedure take (var i : integer);
    ...
  end;

begin
  ... (* Initialization of monitor variables *)
end;
```

# Monitori

## ◆ Uslovna sinhronizacija u monitorima (Hoare, 1974)

- *uslovna promenljiva (condition variable)* je član (promenljiva unutar) monitora sa dve operacije:
  - *wait*: proces koji je izvršio *wait* se (bezuslovno) suspenduje (blokira) i smešta u red čekanja pridružen ovoj uslovnoj promenljivoj; proces potom oslobađa svoj ekskluzivni pristup monitoru i time dozvoljava da drugi proces uđe u monitor;
  - *signal*: kada neki proces izvrši ovu operaciju, sa reda blokiranih procesa na ovoj uslovnoj promenljivoj oslobađa se (deblokira) jedan proces, ako takvog ima; ako takvog procesa nema, onda ova operacija nema nikakvog efekta

# Monitori

- ◆ Primer: ograničeni bafer (*bounded buffer*)

```
monitor buffer;
  export append, take;

var
  buf : array[0..size-1] of integer;
  top, base : 0..size-1;
  numberInBuffer : integer;
  spaceAvailable, itemAvailable : condition;

procedure append (i : integer);
begin
  while numberInBuffer = size do
    wait(spaceAvailable);
  end while;
  buf[top] := i;
  numberInBuffer := numberInBuffer+1;
  top := (top+1) mod size;
  signal(itemAvailable);
end append;
```

# Monitori

- ◆ Primer: ograničeni bafer (*bounded buffer*), nastavak

```
procedure take (var i : integer);
begin
    while numberInBuffer = 0 do
        wait(itemAvailable);
    end while;
    i := buf[base];
    base := (base+1) mod size;
    numberInBuffer := numberInBuffer-1;
    signal(spaceAvailable);
end take;

begin (* Initialization *)
    numberInBuffer := 0;
    top := 0; base := 0
end;
```

# Monitori

- ◆ Razlike između operacija *wait* i *signal* na semaforu i na uslovnoj promenljivoj:
  - Operacija *wait* na uslovnoj promenljivoj uvek blokira proces, za razliku od operacije *wait* na semaforu
  - Operacija *signal* na uslovnoj promenljivoj nema efekta na tu promenljivu ukoliko na njoj nema blokiranih procesa, za razliku od operacije *signal* na semaforu
- ◆ Šta se dešava kada se operacijom *signal* deblokira neki proces - tada postoje dva procesa koja konkurišu za pristup monitoru (onaj koji je izvršio *signal* i onaj koji je deblokiran), pri čemu ne smeju oba nastaviti izvršavanje?

# Monitori

◆ Različite varijante definisane semantike operacije *signal* koje ovo rešavaju:

- Operacija *signal* je dozvoljena samo ako je poslednja akcija procesa pre napuštanja monitora (kao u primeru ograničenog bafera)
- Operacija *signal* ima sporedni efekat izlaska procesa iz procedure monitora (implicitni *return*) - proces nasilno napušta monitor
- Operacija *signal* koja deblokira drugi proces implicitno blokira proces koji je izvršio *signal*, tako da on može da nastavi izvršavanje tek kada monitor postane slobodan; procesi koji su blokirani na ovaj način imaju prednost u odnosu na druge procese koji tek žele da uđu u monitor spolja
- Operacija *signal* koja deblokira drugi proces ne blokira proces koji je izvršio *signal*, ali deblokirani proces može da nastavi izvršavanje tek kada proces koji je izvršio *signal* napusti monitor

# Monitori

- ◆ Jedan od osnovnih problema vezanih za monitore: kako razrešiti situaciju kada se proces koji je napravio ugnezđeni poziv operacije drugog monitora iz operacije jednog monitora suspenduje unutar tog drugog monitora?
- ◆ Zbog semantike *wait* operacije, pristup drugom monitoru biće oslobođen, ali neće biti oslobođen pristup monitoru iz koga je napravljen ugnezđeni poziv. Tako će procesi koji pokušavaju da uđu u taj monitor biti blokirani, što smanjuje konkurentnost
- ◆ Mogući pristupi:
  - Spoljašnji monitor se drži zaključanim (Java, POSIX, Mesa)
  - Potpuno se zabranjuje ugnezđivanje poziva operacija monitora (Modula-1)
  - Obezbediti konstrukte kojima bi se definisalo koji monitori se oslobađaju u slučaju blokiranja na uslovnoj promenljivoj u ugnezđenom pozivu

# Monitori

- ◆ Implementacija monitora na jeziku C++ pomoću semafora:

```
class Monitor {  
public:  
    Monitor () : sem(1) {}  
    void criticalSection ();  
private:  
    Semaphore sem;  
};  
  
void Monitor::criticalSection () {  
    sem.wait();  
    //... telo kritične sekcije  
    sem.signal();  
}
```

# Monitori

- ◆ Kako rešiti slučaj izlaska iz sredine operacije (sa *return* ili u slučaju izuzetka)?

```
int Monitor::criticalSection () {
    sem.wait();
    return f() + 2/x; // gde pozvati signal()?
}
```

- ◆ Rešenje:

```
class Mutex {
public:
    Mutex (Semaphore* s) : sem(s) { if (sem) sem->wait(); }
    ~Mutex () { if (sem) sem->signal(); }
private:
    Semaphore *sem;
};

void Monitor::criticalSection () {
    Mutex dummy (&sem);
    //... telo kritične sekcije
}
```

# Razmena poruka

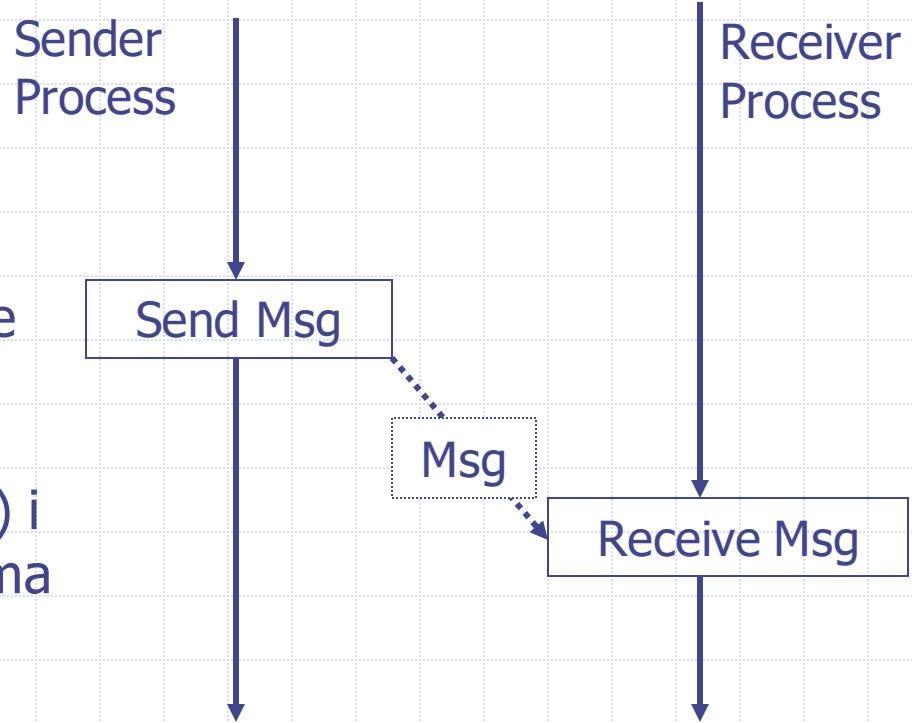
- ◆ *Razmena poruka (message passing)* podrazumeva korišćenje jedinstvenog konstrukta i za sinhronizaciju i za komunikaciju između procesa
- ◆ Osnovna ideja: jedan proces šalje poruku, a neki proces(i) prima(ju) poruku
- ◆ Varijacije u pogledu:
  - modela sinhronizacije
  - načina imenovanja
  - strukture poruke

# Razmena poruka: Sinhronizacija

◆ Implicitna sinhronizacija: primalac ne može da primi poruku pre nego što je pošiljalac poslao poruku (uporediti sa čitanjem i pisanjem deljene promenljive)

◆ Tipovi:

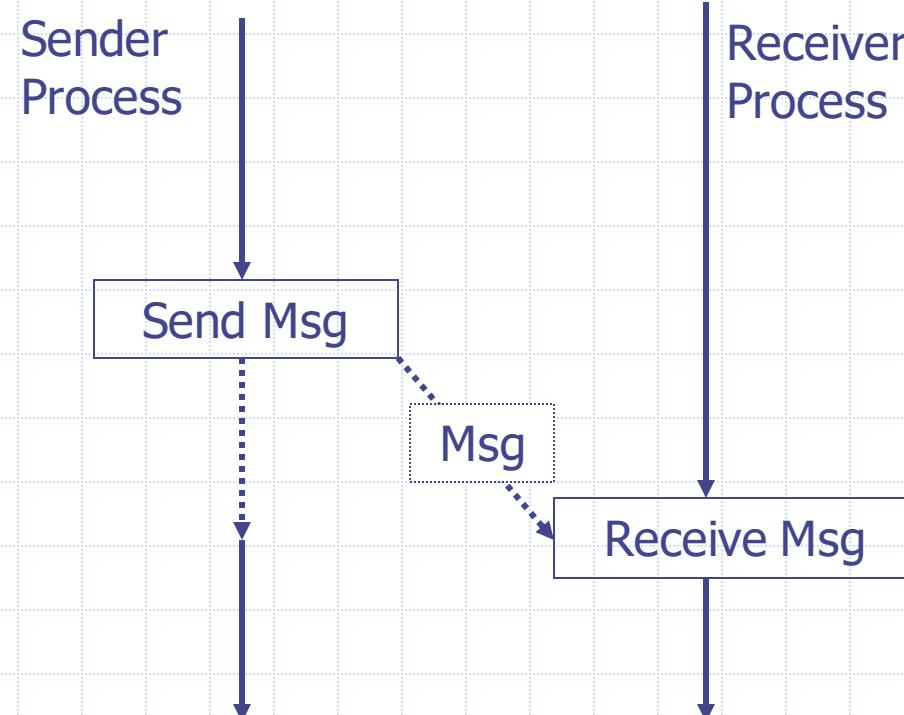
- *asinhrono ili bez čekanja/blokiranja* (*asynchronous, no-wait, non-blocking*): pošiljalac nastavlja svoje izvršavanje odmah posle slanja poruke, bez čekanja da poruka bude primljena. Neki programski jezici (npr. CONIC) i POSIX. Analogija: slanje pisama običnom ili e-poštom, SMS i druge poruke



# Razmena poruka: Sinhronizacija

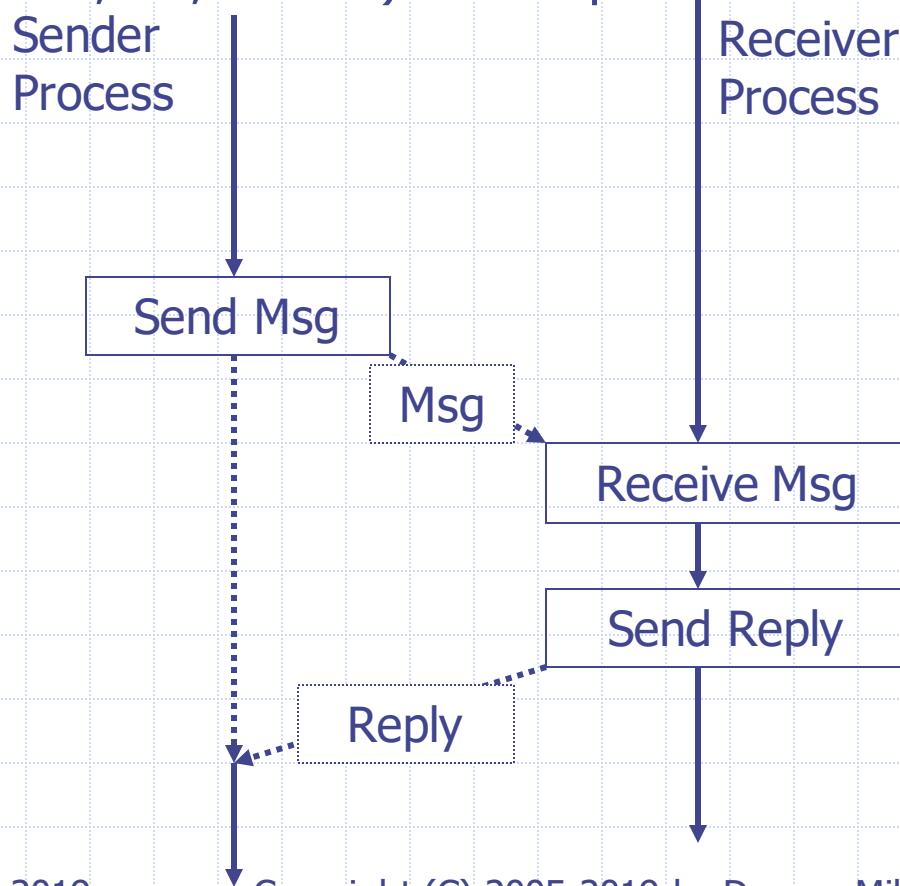
Asinhrono slanje podrazumeva postojanje bafera nenulte i teorijski neograničene veličine. Problem: baferi su uvek fizički ograničeni

- *sinhrono ili randevu (synchronous, rendez-vous)*: pošiljalac se suspenduje sve dok poruka nije primljena i tek tada nastavlja svoje izvršavanje. Neki programski jezici (npr. CSP i occam2). Ekv. asinhronom slanju sa baferima veličine 0. Analogija: telefonski poziv



# Razmena poruka: Sinhronizacija

- Poziv udaljene procedure ili proširenji randevu (*remote procedure call, RPC, extended rendez-vous*): pošiljalac nastavlja izvršavanje tek kada je primalac obradio poruku i poslao odgovor. Neki jezici (npr. Ada, SR, CONIC) i neki operativni sistemi



# Razmena poruka: Sinhronizacija

## ◆ Veze između ovih modela:

- realizacija sinhrone pomoću asinhrone komunikacije:

Sender process P1:  
`async_send(msg);  
receive(ack);`

Receiver process P2:  
`receive(msg);  
async_send(ack);`

- realizacija RPC pomoću sinhrone komunikacije:

Sender process P1:  
`sync_send(msg);  
receive(reply);`

Receiver process P2:  
`receive(msg);  
construct reply;  
sync_send(reply);`

- ◆ Sinhroni prijem: *receive* blokira primaoca dok poruka ne stigne
- ◆ Asinhroni prijem: *receive* vraća poruku ako je ona u trenutku poziva već stigla, inače vraća *null*

# Razmena poruka: Imenovanje

- ◆ Direktno imenovanje: proces direktno imenuje sagovornika  
`send(P2, msg);`      `receive(P1, msg);`
- ◆ Prednost: jednostavnost. Nedostatak: smanjena fleksibilnost - *hardcoded naming*
- ◆ Indirektno imenovanje: postoji međumedijum za prenos poruka između procesa; procesi imenuju ovaj medijum, a ne samog sagovornika
- ◆ Medijumi: *kanal (channel)*, *poštansko sanduče (mailbox)*, *veza (link)*, *cevovod (pipe)*, *vrata (port)*, *priklučnica (socket)*



- ◆ Primer: poštansko sanduče  
`send(msgBox, message);`

`receive(msgBox, message);`

# Razmena poruka: Imenovanje

- ◆ Prednost indirektnog imenovanja: veća fleksibilnost (nezavisnost učesnika od imenovanja) i bolja enkapsulacija
- ◆ Simetrično imenovanje: sagovornici na isti način imenuju jedan drugog; tipično: primalac imenuje pošiljaoca od koga želi da primi poruku:

```
send(P2, msg);
```

```
send(MBX, msg);
```

```
receive(P1, msg);
```

```
receive(MBX, msg);
```

- ◆ Asimetrično imenovanje: pošiljalac imenuje primaoca, primalac ne imenuje izvor poruke (prima poruku od bilo koga); odnos klijent/server:

```
send(P2, msg);
```

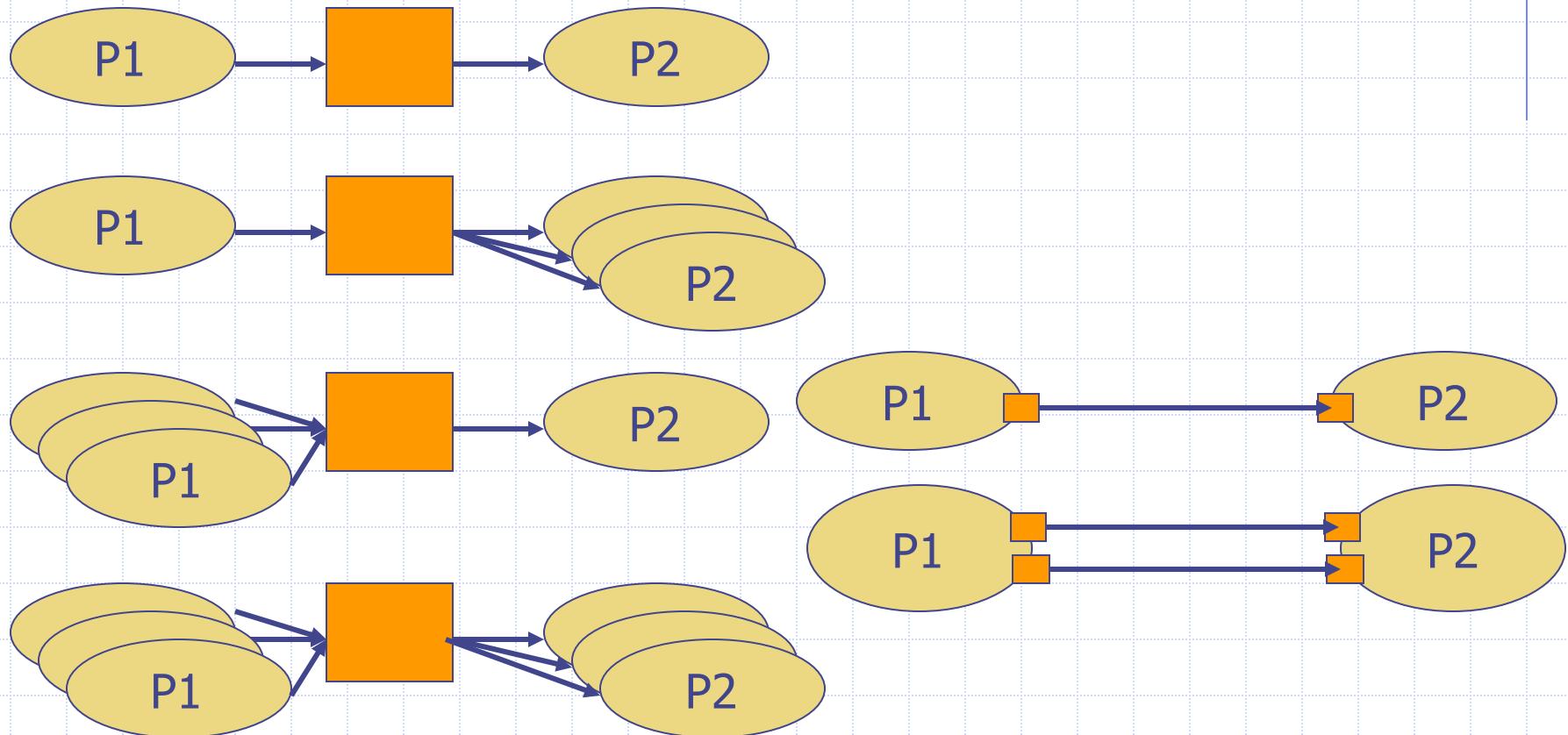
```
send(MBX, msg);
```

```
receive(msg);
```

```
receive(msg);
```

# Razmena poruka: Imenovanje

- ◆ Fleksibilnost kod indirektnog imenovanja:

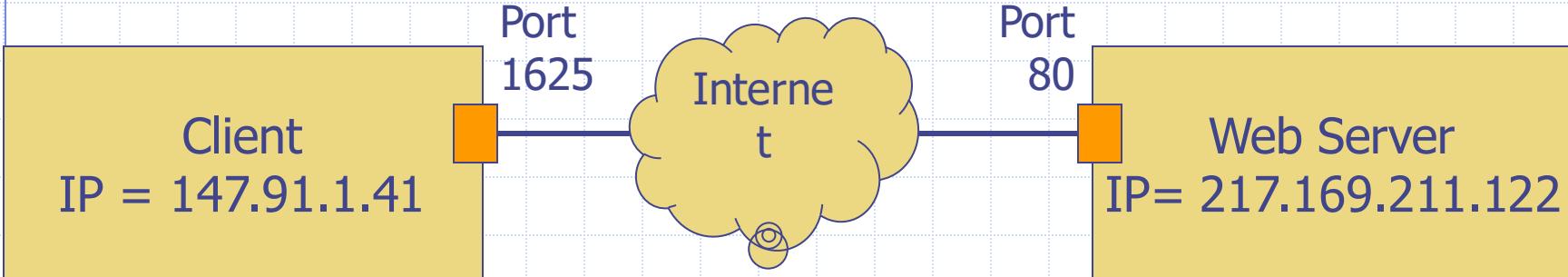


# Razmena poruka: Struktura poruke

- ◆ Operativni sistemi uglavnom podržavaju prenos samo nestrukturiranog niza bajtova određene dužine; korisnički procesi moraju da izgrade strukturu nad tim nizom bajtova
- ◆ Kada se poruka prenosi na drugi računar preko mreže, potrebno je izvršiti:
  - *serijalizaciju*: pretvaranje strukture podataka iz izvornog programskog jezika i adresnog prostora pošiljaoca u prosti niz bajtova za prenos, na strani pošiljaoca – *marshalling*
  - *deserijalizaciju*: pretvaranje prostog niza bajtova iz prenosnog bafera u strukturu podataka programskog jezika na strani primaoca – *unmarshalling*
- ◆ Problem kod prenosa strukturiranih podataka: mašinska kompatibilnost tipova podataka na različitim platformama
- ◆ Pristup rešavanju: konverzija u mašinski nezavisan, standardan format (niz znakova, npr. UTF-8)

# Priklučnice

- ◆ Koncept komunikacije preko priključnica (*socket*):
  - indirektno, simetrično imenovanje
  - asinhrono slanje, sinhroni prijem

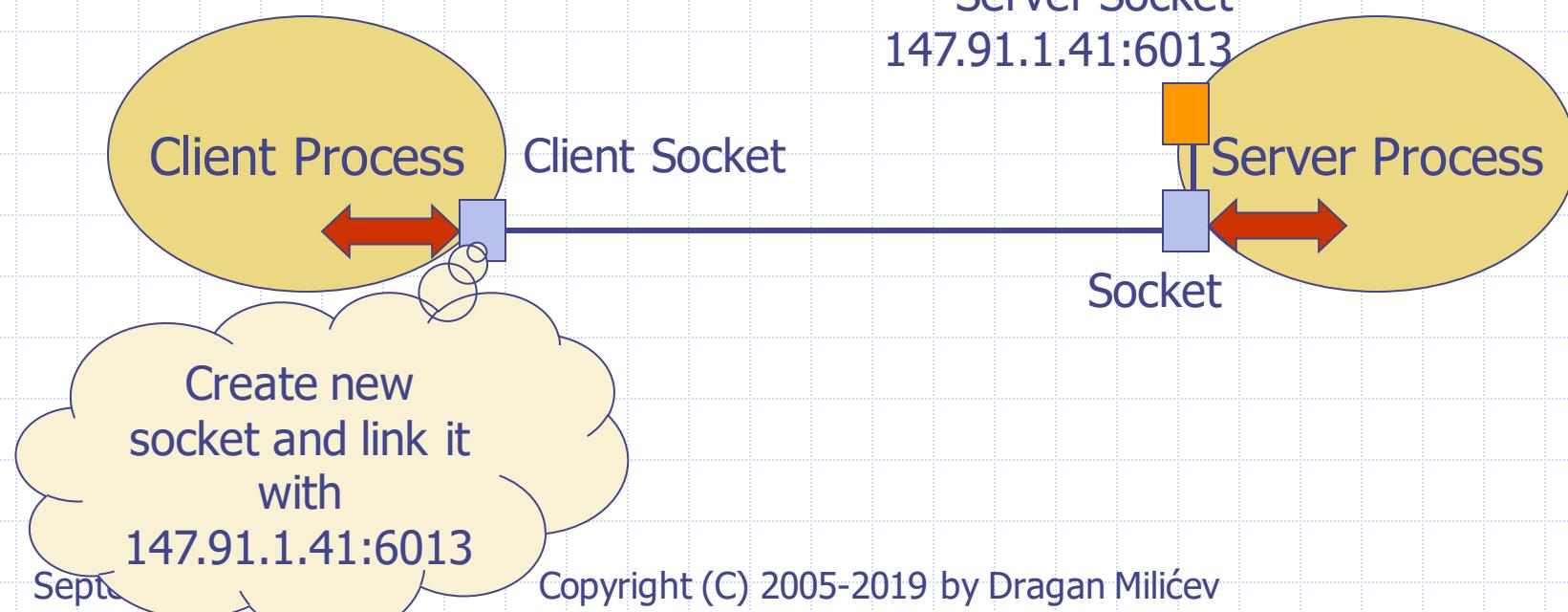


# Priklučnice

- ◆ Serverski proces koji se ne završava, već neprestano radi "u pozadini" (demonski proces, *daemon*), stalno "osluškuje" zahteve za uspostavljanjem komunikacije na svojoj priključnici ("osluškivač", *listener*)
- ◆ Portovi ispod 1024 su *poznati*, jer ih koriste standardni Internet servisi (telnet, ftp, http, ...)
- ◆ Korišćenjem API za određeni programski jezik (C, Java, ...), serverski proces može da:
  - kreira svoju priključnicu na zadatom portu, na kojoj će da prima zahteve od klijenata
  - prihvati zahtev od klijenta i identificiše njegovu priključnicu preko koje je ovaj uspostavio vezu
  - komunicira preko klijentske priključnice
  - raskine vezu sa klijentom

# Priklučnice

- ◆ Korišćenjem API za određeni programski jezik (C, Java, ...), klijentski proces može da:
  - kreira svoju priključnicu povezану на задату serversku priključnicу (IP:port); за нову priključnicу OS odvaja увек нови, јединствени port на klijentу > 1024
  - комуникаира преко своје приклjučнице са serverом
  - раскине везу са serverом



# Priklučnice

## ◆ Primer (Java): server

```
import java.net.*;
import java.io.*;

public class DateServer {
    public static void main (String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);
            while (true) {
                Socket client = sock.accept(); // Blocking
                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);
                pout.println(new java.util.Date().toString());
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

# Priklučnice

## ◆ Primer (Java): klijent

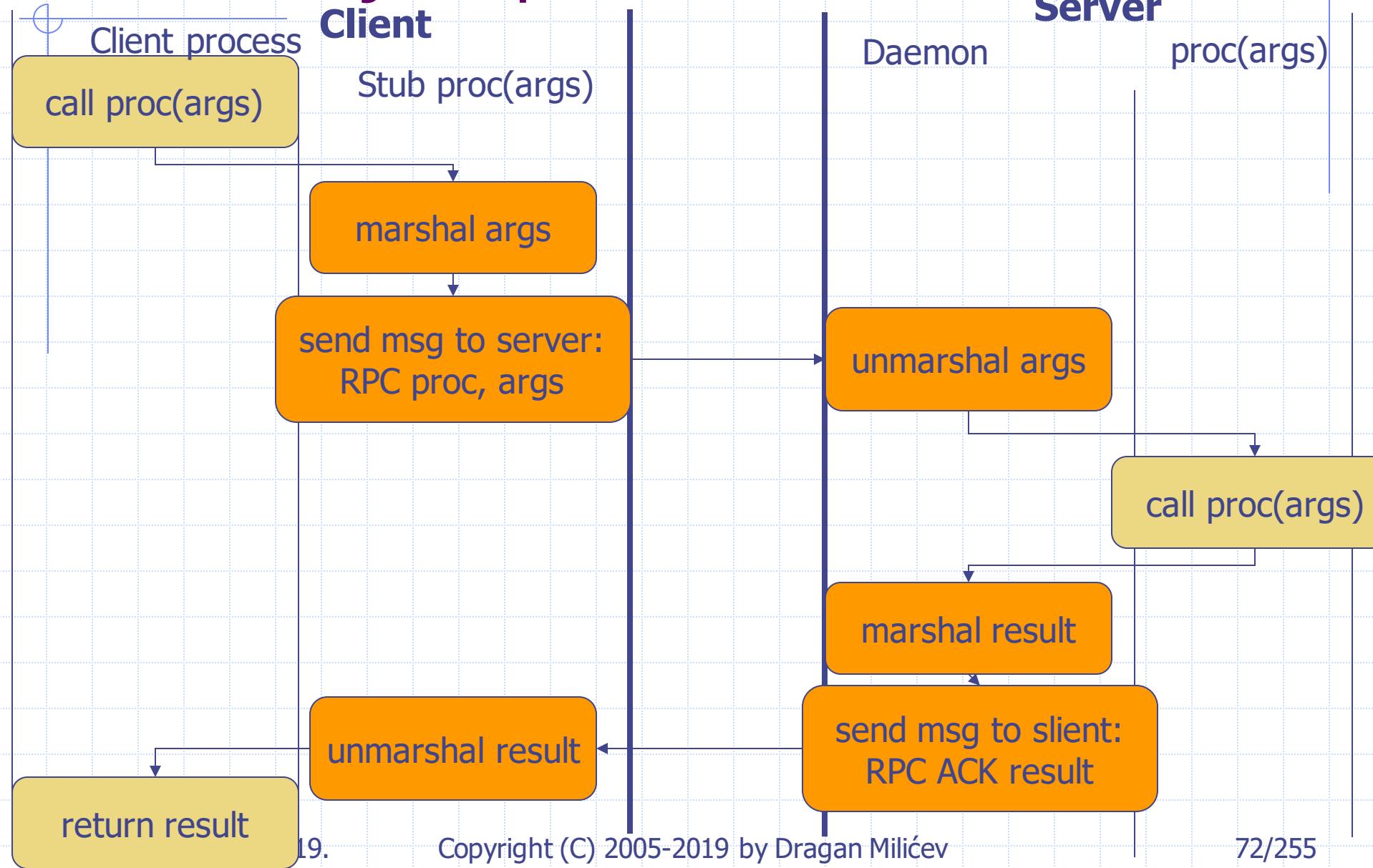
```
import java.net.*;
import java.io.*;

public class DateClient {
    public static void main (String[] args) {
        try {
            Socket sock = new Socket("147.91.1.41", 6013);
            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));
            String line;
            while ((line=bin.readLine()) != null)
                System.out.println(line);
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

# Poziv udaljene procedure

- ◆ Priključnice (*socket*) su efikasan i jednostavan koncept, ali na suviše niskom nivou apstrakcije i zahtevaju mnogo kodovanja
- ◆ Koncept višeg nivoa apstrakcije: *poziv udaljene procedure* (*remote procedure call*, RPC)
- ◆ Ideja: obezbediti mehanizam da klijentski proces poziva proceduru čiji se kod nalazi i izvršava na udaljenom računaru, transparentno, na način kao da je obična (lokalna) procedura
- ◆ RPC je mehanizam višeg nivoa koji se izgrađuje na komunikacionim mehanizmima nižeg nivoa: priključnice (*socket*), klijentski i demonski (*daemon*) serverski procesi
- ◆ RPC obezbeđuje komunikaciju sa strukturiranim porukama – argumenti poziva procedure i povratni rezultat

# Poziv udaljene procedure



return result

19.

Copyright (C) 2005-2019 by Dragan Milićev

72/255

# Poziv udaljene procedure

- ◆ Problem: obezbediti semantiku "tačno jednog" poziva, kao kod lokalne procedure:
  - otpornost na izgubljene poruke:
    - ◆ poruke za povratnu potvrdu prijema – "povratnice" (*acknowledgement, ACK*); šta raditi ako se izgubi povratnica?
    - ◆ vremenska kontrola čekanja na povratnicu; šta ako istekne vremenska kontrola?
    - ◆ ponavljanje poruke u slučaju isteka vremenske kontrole
  - otpornost na višestruke poruke: "vremenski pečati" (*time stamp*) uz poruke – server čuva istoriju poruka koje je već obradio i odbacuje ponovljene poruke
- ◆ Problem: kompatibilnost mašinskog formata tipova podataka na različitim platformama. Pristup rešavanju: konverzije u mašinski nezavisne prenosne formate, standardi (npr. XML) i alati
- ◆ Veb servisi: implementacija RPC na HTTP protokolu. Standardi SOAP i WSDL

# Poziv udaljene procedure

- ◆ Problem: kako učiniti fleksibilnim vezivanje procedura u portove za komunikaciju?
- ◆ Pristup rešavanju:
  - server prima RPC zahteve na uvek istom portu (gde osluškuje demonski proces), staticki definisanom; klijent šalje zahtev za RPC na taj port sa identifikacijom procedure
  - za svaki primljeni zahtev na ovom portu, server dinamički preslikava (vezuje) dati poziv procedure za svoj poseban port odvojen za tu proceduru i šalje klijentu odgovor sa brojem tog porta
  - klijent otvara novu konekciju sa serverom na dobijenom portu
  - klijent šalje RPC zahtev na taj port; RPC se obavlja na tom paru portova

# Glava 4: Upravljanje deljenim resursima

Modeli pristupa deljenim resursima

Čitaoci i pisci

Filozofi koji večeraju

Problemi nadmetanja za deljene resurse

Utrkivanje

Mrtvo blokiranje

Živo blokiranje

Izgladnjivanje

# Modeli pristupa deljenim resursima

- ◆ U teoriji i praksi konkurentnog programiranja koristi se nekoliko modela (test-primera) za ispitivanje i demonstraciju praktično svakog novopredloženog koncepta za sinhronizaciju i komunikaciju između procesa
- ◆ Ovi modeli odslikavaju tipične situacije nadmetanja konkurentnih procesa za pristup do deljenih resursa koje se sreću pri konstrukciji OS i konkurentnih programa
- ◆ Standardni modeli:
  - *ograničeni bafer (bounded buffer)* – već obrađen
  - *čitaoci i pisci (readers-writers)*
  - *filozofi koji večeraju (dining philosophers)*

# Čitaoci i pisci

- ◆ Koncept monitora obezbeđuje međusobno isključenje pristupa konkurentnih procesa do deljenog resursa, uz eventualnu uslovnu sinhronizaciju. Međutim, koncept potpunog međusobnog isključenja kod monitora ponekad predstavlja suviše restriktivnu politiku koja smanjuje konkurentnost programa
- ◆ Često se operacije nad deljenim resursom mogu svrstati u operacije koje:
  - samo čitaju deljene podatke, odnosno ne menjaju stanje resursa (operacije čitanja)
  - upisuju u deljene podatke, tj. menjaju stanje resursa (operacije upisa)

Koncept monitora ne dozvoljava nikakvu konkurentnost ovih operacija

# Čitaoci i pisci

Konkurentnost se može povećati ukoliko se dozvoli da:

- proizvoljno mnogo procesa izvršava operacije čitanja (*čitaoci, readers*)
- najviše jedan proces izvršava operaciju upisa (*pisac, writer*), međusobno isključivo sa drugim piscima, ali i sa čitaocima
- ◆ Na taj način, deljenom resursu u datom trenutku može pristupati ili samo jedan pisac, ili jedan ili više čitalaca, ali ne istovremeno i jedni i drugi – *više čitalaca-jedan pisac* (*multiple readers-single writer*)
- ◆ Postoje različite varijante ove šeme koje se razlikuju u pogledu prioriteta koji se daje procesima koji čekaju na pristup resursu:
  - prioritet imaju pisci koji čekaju, tj. čim postoji pisac koji čeka, svi novi čitaoci biće blokirani sve dok svi pisci ne završe
  - prioritet imaju čitaoci, tj. piscu se ne dozvoljava pristup sve dok svi čitaoci ne završe, a novi čitaoci se puštaju pre pisaca

# Čitaoci i pisci

Implementacija korišćenjem monitora: monitor poseduje četiri operacije, **startRead**, **stopRead**, **startWrite** i **stopWrite**. Čitaoci i pisci moraju da budu strukturirani na sledeći način:

**Reader:**

```
startRead();  
...// Read data structure  
stopRead();
```

**Writer:**

```
startWrite();  
...// Write data struct  
stopWrite();
```

# Čitaoci i pisci

```
class ReadersWriters {
public:
    ReadersWriters ();

    void startRead ();
    void stopRead ();
    void startWrite ();
    void stopWrite ();

private:
    Semaphore mutex, wrt;
    int readcount;
};

ReadersWriters::ReadersWriters ()
    : mutex(1), wrt(1), readcount(0) {}

void ReadersWriters::startWrite () {
    wrt.wait();
}

void ReadersWriters::stopWrite () {
    wrt.signal();
}
```

# Čitaoci i pisci

```
void ReadersWriters::startRead () {  
    mutex.wait();  
    readcount++;  
    if (readcount==1) wrt.wait();  
    mutex.signal();  
}  
  
void ReadersWriters::stopRead () {  
    mutex.wait();  
    readcount--;  
    if (readcount==0) wrt.signal();  
    mutex.signal();  
}
```

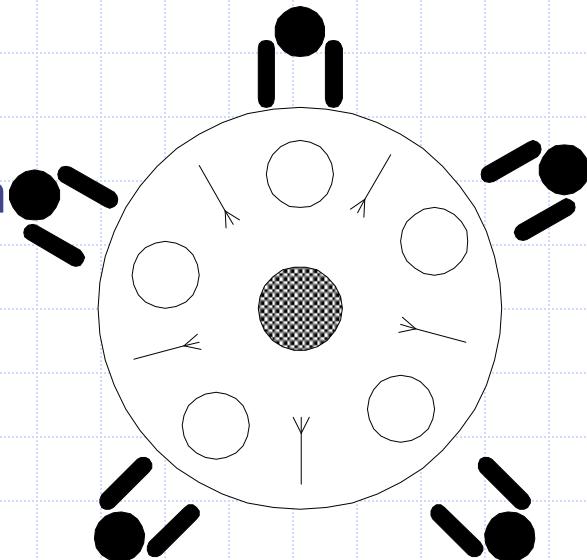
Problem?

# Filozofi koji večeraju

- ◆ Problem *filozofa koji večeraju* (*dining philosophers*, Dijkstra 1965.):

Pet filozofa sedi za okruglim stolom na kome se nalazi posuda sa špagetima, jedu i razmišljaju. Svaki filozof ima svoj tanjur, a između svaka dva susedna tanjira стоји по једна viljuška.

Prepostavlja se да су сваком filozofу, да би се послужио, потребне две viljušке, као и да може да користи само one које се налазе лево и десно од njegovog tanjira. Ако је једна од њих зазета, он мора да чека. Svaki filozof циклично jede, па размишља. Kad zavrши са јелом, filozof спушта обе viljušке на сто и наставља да размишља. Posle неког времена, filozof огладни и поново покушава да jede. Potrebno je definisati protokol (правила понашања, алгоритам) koji će obezbediti ovakvo понашање filozofa i pristup do viljušaka



# Problemi nadmetanja za deljene resurse

- ◆ Da bi konkurentni program bio korektan, mora da zadovolji sledeće uslove:
  - logičku korektnost rezultata bez obzira na redosled preplitanja izvršavanja delova konkurentnog programa
  - *živost (liveness)* ili *napredak (progress)*: sve što je programom predviđeno da se desi, treba da se desi u konačnom vremenu; procesi moraju da napreduju, ne smeju večno da čekaju
- ◆ Greške koji dovode do narušavanja ovih uslova:
  - *utrivanje (race condition)*: nije obezbeđena logička ispravnost rezultata u svim situacijama; primer je izostanak međusobnog isključenja kritične sekcije
  - *izgladnjivanje (starvation)*: nije obezbeđena živost
  - *živo blokiranje (livelock)*: nije obezbeđena živost
  - *mrtvo blokiranje (deadlock)*: nije obezbeđena živost

# Utrkivanje

## ◆ Primer (nekorektne) uslovne sinhronizacije:

- *suspend*: bezuslovno suspenduje pozivajući proces
- *resume*: bezuslovno deblokira imenovani proces, ako je blokiran

```
flag : Boolean := false;
```

Process P1 :

...

```
if not flag then suspend;  
flag := false;
```

...

Process P2 :

...

```
flag := true;  
P1.resume;
```

...

◆ Problem: P1 ispita **flag (=false)**, dode do promene konteksta, P2 postavi flag na **true** i izvrši **resume** P1 (bez efekta), potom P1 izvrši **suspend** – gubitak sinhronizacije!

◆ Opšti primer: neobezbeđeno međusobno isključenje kritične sekcije

# Utrkivanje

- ◆ Ovakav neispravan uslov naziva se *utrkivanje (race condition)*. Nastaje zbog preplitanja delova koji bi morali biti izvršeni neprekidivo
- ◆ Tipično nastaje kao posledica toga što se odluka o promeni stanja procesa (suspenziji) donosi na osnovu ispitivanja vrednosti deljene promenljive, pri čemu ta dva koraka nisu nedeljiva, pa može doći do preuzimanja, tj. "utrkivanja" od strane drugog procesa koji pristupa istoj deljenoj promenljivoj

# Mrtvo blokiranje

- ◆ Algoritam ponašanja filozofa:

```
var forks : array 0..4 of semaphore = 1;
task type Philosopher(i:int)
var left, right : 0..4;
begin
    left := i; right:=(i+1) mod 5;
loop
    think;
    forks[left].wait; // take left fork
    forks[right].wait; // take right fork
    eat;
    forks[left].signal; // release left fork;
    forks[right].signal; // release right fork;
end;
end;
```

- ◆ Scenario: svaki filozof uzme svoju levu viljušku i čeka na desnu?!

# Mrtvo blokiranje

- ◆ Ovakav neregularan uslov nastaje tako što se grupa procesa koji konkurišu za deljene resurse međusobno kružno blokiraju - *mrtvo (ili kružno) blokiranje (deadlock)*
- ◆ U opštem slučaju, mrtvo blokiranje nastaje tako što se grupa procesa nadmeće za ograničene resurse, pri čemu proces  $P_1$  drži ekskluzivan pristup do resursa  $R_1$  i pri tom čeka blokiran da se oslobodi resurs  $R_2$ , proces  $P_2$  drži ekskluzivan pristup do resursa  $R_2$  i pri tom čeka blokiran da se oslobodi resurs  $R_3$ , itd., proces  $P_n$  drži ekskluzivan pristup do resursa  $R_n$  i pri tom čeka blokiran da se oslobodi resurs  $R_1$ . Tako procesi ostaju neograničeno suspendovani u cikličnom lancu blokiranja
- ◆ Jedan od najtežih problema koji mogu da nastanu u konkurentnim programima, multiprogramskim sistemima i samim operativnim sistemima

# Živo blokiranje

- ◆ Algoritam ponašanja filozofa koji izbegava mrtvo blokiranje:

```
task type Philosopher
    loop
        think;
        loop
            take_left_fork;
            if can_take_right_fork then
                take_right_fork;
                exit loop;
            else
                release_left_fork;
            end if;
        end;
        eat;
        release_left_fork;
        release_right_fork;
    end;
end;
```

- ◆ Scenario: svi filozofi uzmu svoju levu viljušku, ne mogu da uzmu desnú, pa spuste levu, i tako ciklično!?

# Živo blokiranje

- ◆ Ovakva neregularna situacija u konkurentnom programu, kod koje se grupa procesa izvršava, ali nijedan ne može da napreduje jer u petlji čeka na neki uslov, naziva se *živo blokiranje (livelock)*
- ◆ Treba razlikovati živo od mrtvog blokiranja. Iako se u oba slučaja procesi nalaze "zaglavljeni" čekajući na ispunjenje nekog uslova, kod mrtvog blokiranja su oni suspendovani, dok se kod živog izvršavaju, tj. uposleno čekaju
- ◆ Međutim, osnovni uzroci su slični (kružna zavisnost), pa se ove dve pojave često tretiraju bez razlike
- ◆ Obe situacije su neispravna stanja jer nije obezbeđena njegova živost (*liveness*)

# Izgladnjivanje

- ◆ Algoritam ponašanja filozofa:

```
task type Philosopher
loop
    think;
    take_both_forks;
    eat;
    release_both_forks;
end;
```

- ◆ Scenario: Filozof X, njegov levi L, desni D; u jednom trenutku L može da uzme obe svoje viljuške, što sprečava filozofa X da uzme svoju levu viljušku; pre nego što L spusti svoje viljuške, D može da uzme svoje, što opet sprečava filozofa X da počne da jede; teorijski, ovaj postupak se može beskonačno ponavljati, što znači da filozof X nikako ne uspeva da zauzme svoje viljuške (resurse) i počne da jede, jer njegovi susedi naizmenično uzimaju njegovu levu, odnosno desnu viljušku!?

# Izgladnjivanje

- ◆ Ovakva neregularna situacija u konkurentnom programu, kod koje jedan proces ne može da dođe do željenog resursa jer ga drugi procesi neprekidno pretiču i zauzimaju te resurse, naziva se *izgladnjivanje (starvation)*, ili *neograničeno odlaganje (indefinite postponement)*, ili *lockout*
- ◆ Drugi primer: prikazano rešenje čitalaca i pisaca; ko izgladnjuje?

# Problemi nadmetanja za deljene resurse

- ◆ Zaključak: da bi konkurentni program ili multiprogramske sisteme obezbedio živost (progres), ne sme da poseduje probleme živog blokiranja, mrtvog blokiranja, ni izgladnjivanja
- ◆ Jedno korektno rešenje problema filozofa pomoću semafora:

```
var forks : array 0..4 of semaphore = 1;
deadlockPrevention : semaphore = 4;

task type Philosopher(i:int) begin
    left := i; right:=(i+1) mod 5;
    loop
        think;
        deadlockPrevention.wait;
        forks[left].wait;
        forks[right].wait;
        eat;
        forks[left].signal;
        forks[right].signal;
        deadlockPrevention.signal;
    end;
end;
```

# Mrtvo blokiranje - Rešavanje

## ◆ Model sistema:

- deljeni resursi su grupisani u *tipove*; postoji  $n_t$  ( $\geq 1$ ) *identičnih instanci* resursa tipa  $t$ ; resursi mogu biti fizički (I/O uređaj, CPU, memorija itd.) ili logički (fajl, semafor, monitor itd.)
- kada neki proces traži resurs tipa  $t$ , *bilo koja* od  $n_t$  instanci resursa tog tipa može da mu bude dodeljena, ukoliko je slobodna
- svaki proces mora da *traži* (*request*) resurs(e) tipa  $t$  pre nego što ga (ih) upotrebi i da ga (ih) *oslobodi* (*release*) posle upotrebe; ukoliko ne postoji dovoljno slobodnih resursa datog tipa kada ih proces traži, proces se blokira; zahtev i oslobađanje resursa su sistemski pozivi
- sistem vodi evidenciju o zauzetim i slobodnim resursima, kao i redovima procesa koji čekaju na resurse (blokirani)

# Mrtvo blokiranje - Rešavanje

## ◆ Neophodni uslovi za nastanak mrtvog blokiranja:

- *Međusobno isključenje (mutual exclusion)*: bar jedan resurs mora biti nedeljiv – samo ga jedan proces može koristiti u jednom trenutku
- *Držanje i čekanje (hold and wait)*: mora postojati proces koji drži bar jedan resurs i istovremeno čeka na neki drugi
- *Nema preotimanja (no preemption)*: resursi se ne mogu preotimati; resurs može dobrovoljno osloboditi samo proces koji ga je zauzeo
- *Kružno čekanje (circular wait)*: mora postojati cikličan lanac procesa tako da svaki proces u lancu drži resurs koga traži naredni proces u lancu

Mrtvo blokiranje može nastati samo ako su sva četiri uslova ispunjena!

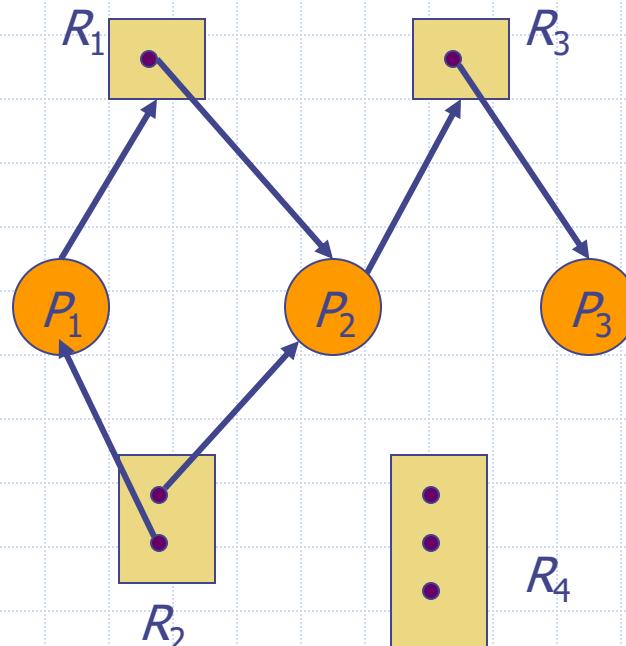
# Mrtvo blokiranje - Rešavanje

◆ Graf zauzetosti resursa (*resource-allocation graph*): usmereni graf čiji su čvorovi aktivni procesi i tipovi resursa, a grane označavaju sledeće:

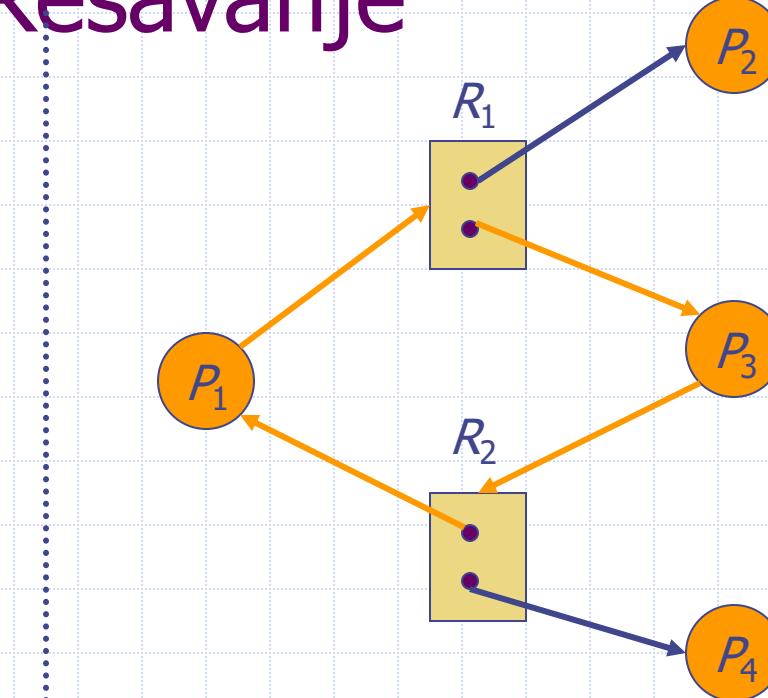
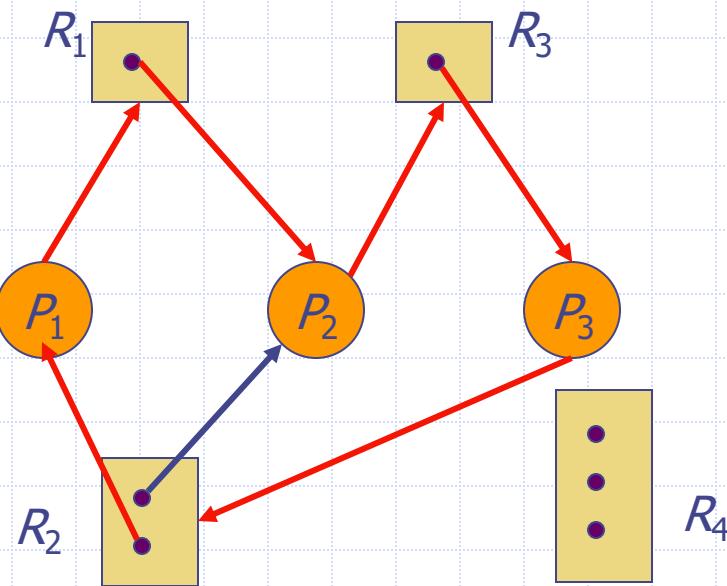
- $P_i \rightarrow R_j$ : proces  $P_i$  traži instancu tipa resursa  $R_j$  i čeka na slobodnu
- $R_j \rightarrow P_i$ : proces  $P_i$  drži zauzetu instancu tipa resursa  $R_j$  (grana zapravo izlazi iz instance tipa resursa  $R_j$ )

◆ Održavanje grafa:

- kada  $P_i$  traži resurs tipa  $R_j$ , uvodi se grana  $P_i \rightarrow R_j$
- kada  $P_i$  zauzme resurs tipa  $R_j$ , briše se grana  $P_i \rightarrow R_j$  i uvodi grana  $R_j \rightarrow P_i$
- kada  $P_i$  oslobodi resurs tipa  $R_j$ , briše se grana  $R_j \rightarrow P_i$



# Mrtvo blokiranje - Rešavanje



- ◆ Ako graf ne sadrži petlju, onda *sigurno nema* mrtve blokade
- ◆ Ako graf sadrži petlju, onda *možda postoji* mrtva blokada:
  - Ako petlja uključuje samo resurse sa po jednom instancom, onda *sigurno postoji* mrtva blokada
  - Ako petlja uključuje i resurse sa više instanci, mrtva blokada može, ali ne mora da postoji

# Mrtvo blokiranje - Rešavanje

## ◆ Načini za rešavanje problema mrtvog blokiranja:

- *sprečavanje (deadlock prevention)*: unapred obezbediti da bar jedan od 4 neophodna uslova za nastanak ne važi
- *izbegavanje (deadlock avoidance)*: tokom izvršavanja izbegavati situacije koje mogu da dovedu do nastanka mrtvog blokiranja
- *detekcija i oporavak (deadlock detection and recovery)*: tokom izvršavanja ne sprečavati niti izbegavati mrtvo blokiranje, već detektovati slučaj kada ono nastane i oporaviti sistem iz te situacije
- *ignorisanje (ignoration)*: jednostavno ignorisati nastanak mrtvog blokiranja: većina današnjih sistema primenjuje ovo – mrtve blokade se retko dešavaju, pa je jednostavnije i jeftinije manuelno rešiti problem (gašenjem procesa ili celog sistema) nego ugrađivati složene mehanizme za rešavanje

# Mrtvo blokiranje - Sprečavanje

- ◆ Sprečavanje mrtvog blokiranja – unapred ukinuti jedan od neophodnih uslova:
  - međusobno isključenje: ako je resurs deljiv (ne zahteva međusobno isključenje), sigurno ne može nastati mrtvo blokiranje; primer: fajl ili drugi resurs koji se samo čita (*read-only*) dozvoljava konkurentan pristup; za nedeljive resurse ovo nije izvodljivo
  - držanje i čekanje: obezbediti da kad proces zahteva resurs, nijedan drugi već ne drži; protokoli:
    - ◆ proces traži *sve* resurse koje koristi odjednom, npr. na početku svog izvršavanja
    - ◆ kada traži resurs, proces ili ne drži druge resurse, ili ih pritom oslobađa
    - ◆ držanje i čekanje sa ograničenim vremenom (*timeout*)  
problemi: slabo iskorišćenje (dugo zadržavanje resursa) i izgladnjivanje

# Mrtvo blokiranje - Sprečavanje

- preotimanje; protokoli:
  - ◆ kada proces traži resurs koji je zauzet, svi resursi koje već drži se oslobođaju i dodaju na spisak onih koje traži
  - ◆ ako proces traži resurs koji je zauzet, i ako taj resurs drži proces koji čeka neki drugi resurs, resurs se preotima i dodeljuje procesu koji ga traži

primenjivo samo na resurse čije se stanje može lako sačuvati i povratiti (CPU, OM)
- kružno čekanje: uspostaviti totalno uređenje između (tipova) resursa i nametnuti da svaki proces zahteva resurse samo u rastućem poretku:
  - ◆ svakom tipu resursa dodeliti funkciju  $F(R)$ , npr. prirodan broj
  - ◆ ako proces već drži zauzet resurs  $R_i$ , proces može da zahteva resurs  $R_j$  samo ako je  $F(R_j) > F(R_i)$ , za svaku  $R_i$  koje drži; inače, mora da oslobodi druge resurse  $R_i$  za koje je  $F(R_j) \leq F(R_i)$

Dokazati da tada ne može da nastane kružno čekanje! (Uputstvo: dokaz kontradikcijom)

# Mrtvo blokiranje - Izbegavanje

- ◆ Tehnike izbegavanja zahtevaju da sistem unapred zna na koji način će proces zahtevati resurse tokom izvršavanja
- ◆ Kada proces traži resurs, sistem analizira informacije o zauzeću resursa i potrebama za resursima i zaključuje da li postoji mogućnost da u budućnosti uđe u mrtvu blokadu; ako zaključi da postoji, ne dozvoljava procesu zauzeće resursa čak i ako je resurs slobodan, *pre nego što je blokada nastala, izbegavajući mrtvu blokadu*
- ◆ Proces unapred deklariše *maksimalni* broj instanci resursa svakog tipa koje će potencijalno koristiti tokom izvršavanja
- ◆ Sistem prati *stanje* zauzeća resursa koje je definisano brojem zauzetih i slobodnih resursa svakog tipa

# Mrtvo blokiranje - Izbegavanje

- ◆ Stanje je *sigurno (safe state)* ako sistem može da dodeli resurse svakom procesu (do maksimuma njegove potražnje) u nekom poretku, a da ipak izbegne mrtvu blokadu
- ◆ Stanje je sigurno akko postoji *sigurna sekvenca*. Sekvenca procesa  $P_1, P_2, \dots, P_n$  je sigurna za dato stanje alokacije resursa, ako za svaki  $P_i$  sistem može da zadovolji zahteve  $P_i$  za resursima koje  $P_i$  još može da postavi, uzimajući u obzir trenutno slobodne resurse, kao i sve resurse koje su zauzeli procesi  $P_j$ ,  $j < i$ ; ako trenutno nema dovoljno slobodnih resursa, onda  $P_i$  može da sačeka da se završe svi  $P_j$ ,  $j < i$ , (koji takođe mogu da se završe), da bi dobio sve resurse, obavio svoj zadatak, oslobođio resurse i završio, posle čega  $P_{i+1}$  može da završi itd.
- ◆ Ako sigurna sekvenca ne postoji, stanje je *nesigurno (unsafe)*

# Mrtvo blokiranje - Izbegavanje

- ◆ Primer: ukupno postoji 12 identičnih resursa

	Maksimum potražnje	Trenutno zauzima
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2



Sigurna sekvenca:  $P_1, P_0, P_2 \Rightarrow$  prikazano stanje je sigurno

Ako  $P_2$  sada zauzme još jedan resurs, sistem prelazi u nesigurno stanje i moguća je mrtva blokada

- ◆ Ako je stanje sigurno, onda ne postoji mrtva blokada
- ◆ Stanje mrtve blokade je nesigurno stanje
- ◆ Ako je sistem u nesigurnom stanju, može (ali ne mora) nastati mrtva blokada
- ◆ Ideja: ne dozvoliti alokaciju resursa ako ona vodi sistem u nesigurno stanje, čak i ako ima slobodnih resursa – stalno držati sistem u sigurnom stanju

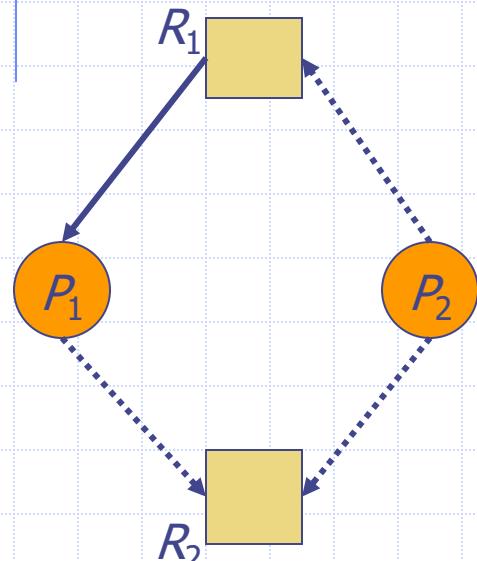
# Mrtvo blokiranje - Izbegavanje

## ◆ Algoritam zasnovan na grafu alokacije:

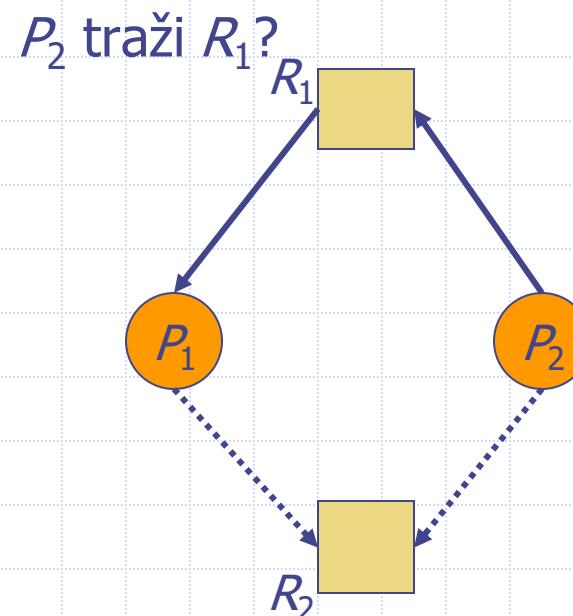
- primenjiv samo za slučajeve sa po jednom instancom svakog tipa resursa
- pored grana zauzetosti i potražnje resursa, uvođe se i grane *najave* tražnje:  $P_i \rightarrow R_j$  znači da proces  $P_i$  može tražiti instancu tipa resursa  $R_j$  u nekom trenutku u budućnosti
- kada proces  $P_i$  zatraži instancu tipa resursa  $R_j$ , granu najave pretvara se u granu potražnje
- kada proces  $P_i$  oslobodi instancu tipa resursa  $R_j$ , granu zauzeća pretvara se u granu najave
- sve grane najave moraju biti inicijalno unesene u graf prilikom pokretanja procesa; relaksiranje: kada proces traži prvi resurs, mora prethodno definisati sve svoje grane najave

# Mrtvo blokiranje - Izbegavanje

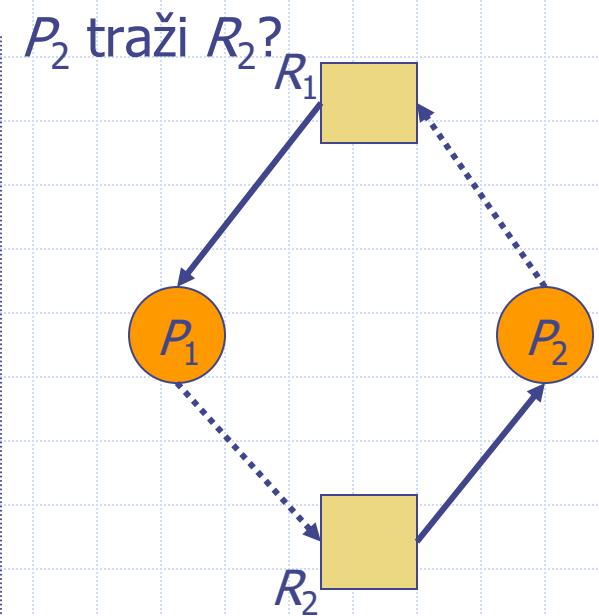
- kada proces  $P_i$  zatraži instancu tipa resursa  $R_j$ , resurs mu se dodeljuje samo ako novonastala grana zauzeća  $R_j$   
→  $P_i$  ne bi formirala petlju u grafu – zadržava sistem u sigurnom stanju; inače, proces mora da čeka na resurs



Sigurno stanje



Sigurno stanje



Nesigurno stanje

# Mrtvo blokiranje - Izbegavanje

## ◆ Bankarev algoritam:

- primenjiv na slučajeve tipova resursa sa više instanci
- manje efikasan nego algoritam zasnovan na grafu
- naziv dobio po tome što pokazuje kako banka da alocira svoj raspoloživi keš tako da uvek može da ispunи potrebe svih svojih klijenata
- osnovna ideja: kada proces traži nove resurse, algoritam proverava da li bi njihova alokacija, čak i ako su slobodni, zadržala sistem u sigurnom stanju, ispitujući da li postoji sigurna sekvenca; ako ne bi, resursi se ne alociraju, već proces mora da čeka

# Mrtvo blokiranje - Izbegavanje

- ◆ Bankarev algoritam - primer:

	<i>Allocation</i>			<i>Max</i>			<i>Available</i>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Da li je ovo stanje sigurno? Pronaći sigurnu sekvencu!

Postupak: pronaći proces koji sa raspoloživim slobodnim resursima može da zadovolji svoj maksimum tražnje, završi svoj posao i oslobodi sve resurse koje drži; zatim pronaći sledeći takav itd.

$Need_i := Max_i - Allocation_i$

Uslov:  $Need_i \leq Available$

$Available := Available + Allocation_i$

# Mrtvo blokiranje - Izbegavanje

- ◆ Bankarev algoritam – primer (nastavak):

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Sigurna sekvenca:  $P_1,$

# Mrtvo blokiranje - Izbegavanje

- ◆ Bankarev algoritam – primer (nastavak):

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	5	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Sigurna sekvenca:  $P_1, P_3,$

# Mrtvo blokiranje - Izbegavanje

- ◆ Bankarev algoritam – primer (nastavak):

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	7	4	3
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Sigurna sekvenca:  $P_1, P_3, P_4,$

# Mrtvo blokiranje - Izbegavanje

- ◆ Bankarev algoritam – primer (nastavak):

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	7	4	5
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Sigurna sekvenca:  $P_1, P_3, P_4, P_2,$

# Mrtvo blokiranje - Izbegavanje

- ◆ Bankarev algoritam – primer (nastavak):

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	10	4	7
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Sigurna sekvenca:  $P_1, P_3, P_4, P_2, P_0$

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	10	5	7
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

# Mrtvo blokiranje - Izbegavanje

- ◆ Bankarev algoritam – primer (nastavak):

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Šta ako u ovom (sigurnom) stanju  $P_1$  traži resurse  $Request_1 = (1, 0, 2)$ ?

Postupak: Ispitati uslov  $Request_1 \leq Available$

Zatim prepostaviti da mu se ovi resursi dodele:

$$Available := Available - Request_1$$

$$Allocation_1 := Allocation_1 + Request_1$$

Ispitati da li je dobijeno stanje sigurno; ako jeste, dozvoliti alokaciju; ako nije, blokirati proces

# Mrtvo blokiranje - Izbegavanje

- ◆ Bankarev algoritam – primer (nastavak):

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	2	3	0
$P_1$	3	0	2	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Sigurna sekvenca:  $P_1, P_3, P_4, P_0, P_2$ , stanje je sigurno, dozvoliti alokaciju

Šta ako sada  $P_0$  traži  $(0, 2, 0)$ ?

Dobijeno stanje bi bilo nesigurno, ne dozvoliti alokaciju!

Zadatak: Precizno formulisati bankarev algoritam i implementirati ga!

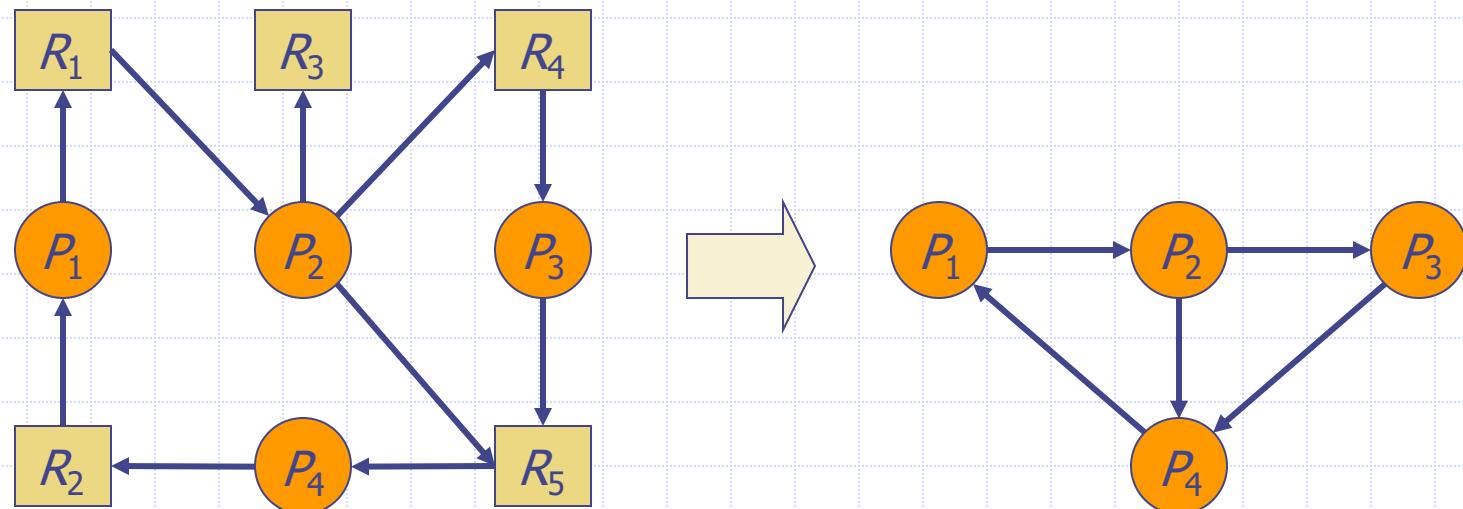
# Mrtvo blokiranje – Detekcija i oporavak

- ◆ Ideja: ako se mrtvo blokiranje ne sprečava i ne izbegava, onda se može dogoditi; upotrebiti algoritme koji:
  - detektuju da je došlo do mrtve blokade
  - vrše oporavak iz ovog stanja
- ◆ Ovakav pristup ima svoju cenu:
  - potrebno je tokom izvršavanja trošiti vreme na režije: održavanje potrebnih struktura podataka i izvršavanje algoritma detekcije
  - oporavak po pravilu dovodi do gubitka
- ◆ Algoritmi detekcije mrtve blokade:
  - zasnovan na grafu alokacije (samo za tipove resursa sa po jednom instancom)
  - zasnovan na varijaciji bankarevog algoritma (za tipove resursa sa više instanci)

# Mrtvo blokiranje – Detekcija i oporavak

## ◆ Algoritam detekcije zasnovan na grafu alokacije:

- upotrebljiv samo za tipove resursa sa po jednom instancom
- graf alokacije se može pojednostaviti pretvaranjem u graf čiji su čvorovi samo procesi, a grane predstavljaju relaciju "čeka na": grane  $P_i \rightarrow R_q$  i  $R_q \rightarrow P_j$  stapaju se u jednu granu  $P_i \rightarrow P_j$
- mrtva blokada postoji ako i samo ako u ovom grafu postoji petlja
- kompleksnost algoritma detekcije petlje je  $\mathcal{O}(n^2)$ , gde je  $n$  broj grana



# Mrtvo blokiranje – Detekcija i oporavak

- ◆ Algoritam za više instanci istog tipa resursa (varijacija bankarevog algoritma):
  - na isti način kao i bankarev algoritam, traži sigurnu sekvencu, tako što fiktivno “završava” jedan po jedan proces koji može da dobije resurse koje traži, a koji su slobodni
  - varijacija onog dela bankarevog algoritma koji proverava da li je stanje sigurno (tj. traži sigurnu sekvencu): umesto da se proverava uslov  $Need_i \leq Available$ , već uslov  $Request_i \leq Available$
  - ako takvu sekvencu nađe, mrtva blokada ne postoji
  - ako takvu sekvencu ne nađe, mrtva blokada postoji, i ona uključuje proces  $P_i$  za koji ovaj uslov nije bio ispunjen
  - kompleksnost  $\mathcal{O}(m \times n^2)$ ,  $m$  – broj tipova resursa,  $n$  – broj procesa

# Mrtvo blokiranje – Detekcija i oporavak

- ◆ Kada pozivati algoritam detekcije mrtve blokade? Odgovor zavisi od toga:
  - koliko često nastaje mrtva blokada: ako nastaje često, i detekciju treba vršiti često
  - koliko procesa je ugroženo mrtvom blokadom: što duže traje mrtva blokada, sve više procesa može biti ugroženo
- ◆ Jedna krajnost: vršiti detekciju prilikom svakog zahteva za resursom koji ne može biti dodeljen – veoma veliki režijski troškovi
- ◆ Mnogo jeftinije rešenje: detekciju vršiti ređe, npr. kada iskorišćenje procesora padne ispod neke granice (mrtva blokada utiče na to da je sve manje procesa aktivno, pa iskorišćenje procesora vremenom pada)

# Mrtvo blokiranje – Detekcija i oporavak

## ◆ Oporavak od mrtve blokade:

- obavestiti operatera koji treba manuelno da reši problem (gašenjem spornih procesa)
- automatski izvršiti oporavak

## ◆ Automatski oporavak od mrtve blokade:

- ugasiti jedan ili više procesa:
  - ◆ ugasiti sve procese koji učestvuju u blokadi
  - ◆ gasiti jedan po jedan proces, sve dok se blokada ne raskine
- izvršiti preotimanje resursa dok se ne raskine blokada

## ◆ Nasilno gašenje procesa nije jednostavno: šta ako je proces u sred pisanja u fajl ili slično?

## ◆ Preotimanje resursa nije jednostavno: šta raditi sa procesom kome je resurs preotet, kako izbeći izgladnjivanje, ...

## ◆ Izbor "žrtve" - procesa za gašenje ili resursa za preotimanje – je prvenstveno ekonomsko pitanje (postići minimalnu cenu/štetu od oporavka)

# Glava 5: Virtuelna memorija

Zamena stranica

Alokacija okvira

*Thrashing*

Memorijski preslikani fajlovi

Alokacija memorije za jezgro OS

Dohvatanje stranica unapred

Zaključavanje stranica

Uticaj programa na performanse

# Zamena stranica

## ◆ Deo OS (*pager*) koji obrađuje *page fault*:

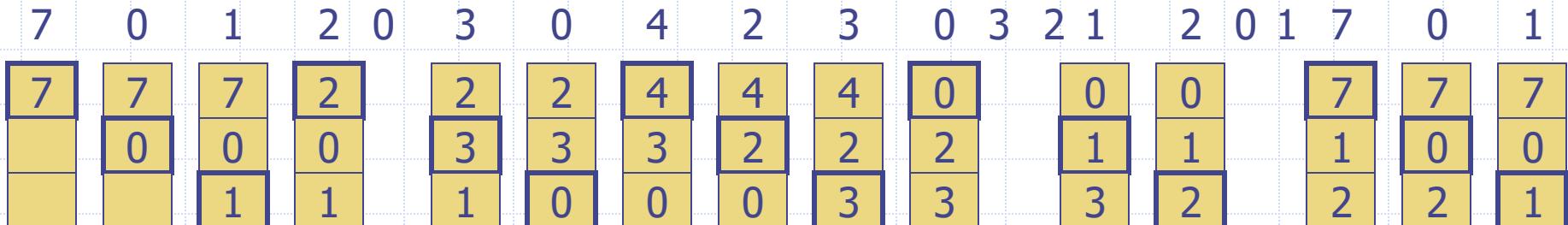
- proverava da li je pristup traženoj stranici uopšte dozvoljen procesu; ako nije, proces se može ugasiti
- pronalazi slobodan okvir u memoriji
- ako takvog nema:
  - ◆ izbacuje taj proces i učitava drugi (*swapping*) ili
  - ◆ bira stranicu-”žrtvu” (*victim*) istog ili drugog procesa koju će izbaciti iz OM po *algoritmu zamene* (*page replacement algorithm*); ako je potrebno, pokreće operaciju snimanja stranice koju izbacuje na disk
- pokreće se operacija sa diskom za učitavanje tražene stranice u odabrani okvir
- kada se operacija završi, ažurira se PMT procesa

# Zamena stranica

- Prilikom izbacivanja stranice, nije je potrebno snimati na disk ako u nju nije bilo upisa:
  - bit upisa ("zaprljanosti", *modify, dirty*) u deskriptoru stranice
  - prilikom učitavanja/alokacije ove stranice, OS briše ovaj bit
  - prilikom operacije upisa u neku lokaciju stranice, HW postavlja ovaj bit
  - prilikom zamene, OS snima stranicu na disk samo ako je ovaj bit postavljen
- Kako evaluirati i porediti algoritme zamene:
  - generisati sekvencu referenciranja memorijskih lokacija (sintetički ili praćenjem realnog sistema)
  - svesti sekvencu referenciranja lokacija na sekvencu referenciranja stranica
  - ispitati ponašanje datog algoritma na datu sekvencu
- Krajnosti: ako je broj okvira 1, broj *page faults* jednak je dužini sekvence; ako je broj blokova dovoljno veliki, broj *page faults* jednak je broju referenciranih stranica
- Intuitivno očekivanje: broj *page faults* opada sa porastom broja blokova fizičke memorije za datu sekvencu referenciranja

# Zamena stranica - FIFO

- ◆ FIFO algoritam: izbaciti onu stranicu koja je najdavnije *učitana* u memoriju
- ◆ Najjednostavniji za implementaciju: implementirati FIFO red stranica; prilikom učitavanja, stranica se stavlja na kraj reda, prilikom izbacivanja, izbacuje se stranica sa početka reda
- ◆ Primer: tri bloka, sekvenca referenciranja stranica je:  
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1  
Broj *page faults*: 15

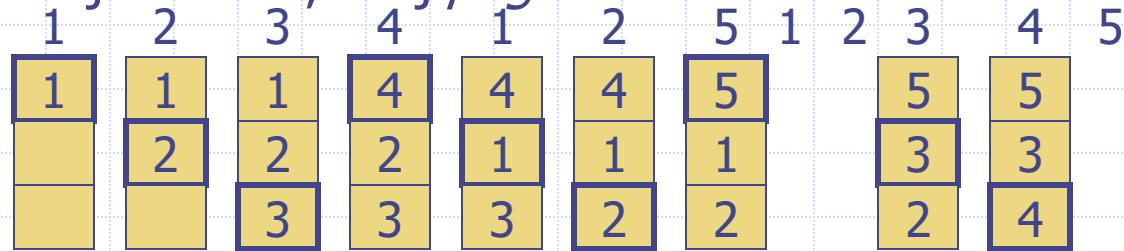


# Zamena stranica - FIFO

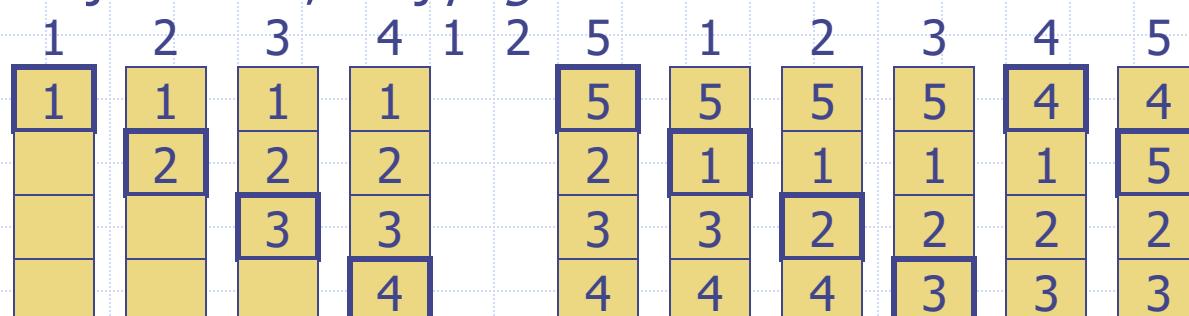
- ◆ Jednostavan algoritam, ali ima problem sa učinkom: šta ako je neki jako korišćeni okvir odavno učitan?
- ◆ Drugi problem - *FIFO anomalija (FIFO anomaly, Belady's anomaly)*: za neke sekvence i neki broj okvira, broj *page faults* može da poraste sa porastom broja okvira; primer:

sekvenca: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

broj okvira 3, broj *page faults*: 9



broj okvira 4, broj *page faults*: 10



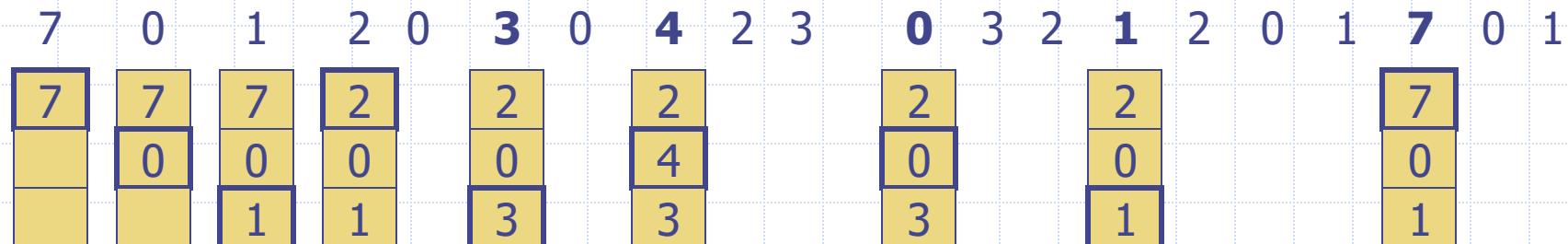
# Zamena stranica - OPT

- ◆ *Optimalni algoritam* (OPT, MIN): zameniti onu stranicu koja *neće biti korišćena* najduže vremena
- ◆ Dokazivo optimalan algoritam u smislu:
  - garantovano namanji broj *page faults* za svaki dati uslov od svih algoritama (za dati broj okvira i sekvencu referenciranja)
  - nikada ne pati od Beladijeve anomalije
- ◆ Jedini problem: nemoguće ga je primeniti (implementirati), jer zahteva poznavanje budućnosti!

- ◆ Primer: tri bloka, ista sekvenca referenciranja:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Broj *page faults*: 9

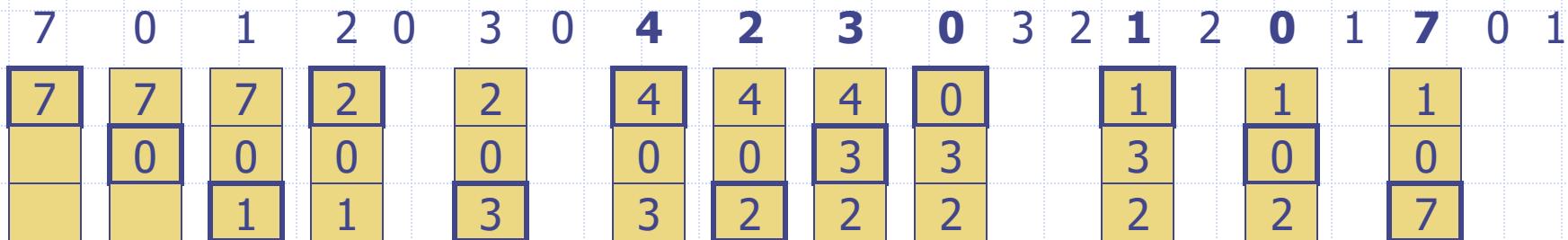


# Zamena stranica - LRU

- ◆ Ako OPT algoritam nije izvodljiv, da li je moguća njegova aproksimacija? Ideja: smatrati blisku prošlost aproksimacijom bliske budućnosti
- ◆ *Least-Recently-Used* (LRU, najdavnije korišćen): izbaciti onu stranicu koja je najdavnije *korišćena*
- ◆ Smatra se dobrim algoritmom i često se koristi. Ne pati od Beladijeve anomalije
- ◆ Primer: tri bloka, ista sekvenca referenciranja:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Broj *page faults*: 12



# Zamena stranica - LRU

- ◆ Kao i OPT algoritam, LRU nikada ne pati od Beladijeve anomalije, pošto spada u grupu *stek algoritama*
- ◆ *Stek algoritam (stack algorithm)* je algoritam zamene stranice za koji se može pokazati da je skup stranica u fizičkoj memoriji sa  $n$  okvira uvek podskup skupa stranica u fizičkoj memoriji sa  $n+1$  okvira, za svaku sekvencu referenciranja
- ◆ Dokaz za LRU je trivijalan: skup stranica u fizičkoj memoriji sa  $n$  okvira u svakom trenutku je skup  $n$  poslednje korišćenih (različitih) stranica, što je uvek podskup od  $n+1$  poslednje korišćenih stranica, a što je skup stranica u fizičkoj memoriji sa  $n+1$  okvira u istom trenutku

# Zamena stranica - LRU

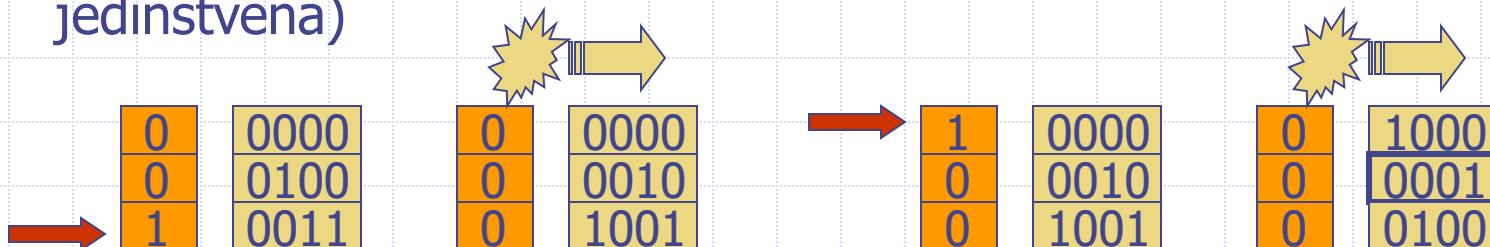
- ◆ Osnovni problem: kako ga efikasno implementirati?  
Neophodna HW podrška zbog ažuriranja struktura podataka  
pri svakom *pristupu* stranici (ne samo učitavanju)
- ◆ Moguća rešenja (*tačna* implementacija LRU algoritma):
  - Brojači:
    - ◆ svakoj stranici pridruži se brojač (čuva se u deskriptoru stranice u PMT)
    - ◆ postoji jedan globalni brojač u CPU koji se inkrementira pri svakom  
pristupu memoriji
    - ◆ pri svakom pristupu memoriji, vrednost CPU brojača prepisuje se u brojač  
pridružen stranici
    - ◆ zamenjuje se stranica sa najmanjom vrednošću brojača
    - ◆ zahteva pretragu svih brojača za traženje minimalnog prilikom zamene
    - ◆ problem prekoračenja opsega brojača

# Zamena stranica - LRU

- Stek stranica:
  - ◆ implementirati LIFO stek stranica
  - ◆ pri svakom pristupu memoriji, stranica kojoj se pristupa premešta se na vrh steka
  - ◆ zamenjuje se stranica sa dna steka
  - ◆ zbog toga je najbolje implementirati pomoću dvostruko ulančane liste
  - ◆ zahteva promenu bar šest pokazivača prilikom premeštanja stranice na vrh steka – pri svakom pristupu drugoj stranici
  - ◆ izbor stranice za zamenu je efikasan
- ◆ Zaključak: *tačna* implementacija LRU algoritma nije nimalo efikasna i zahteva značajnu podršku hardvera
- ◆ Zbog toga se u praksi pribegava *aproksimacijama* LRU algoritma

# Zamena stranica – LRU aproksimacija

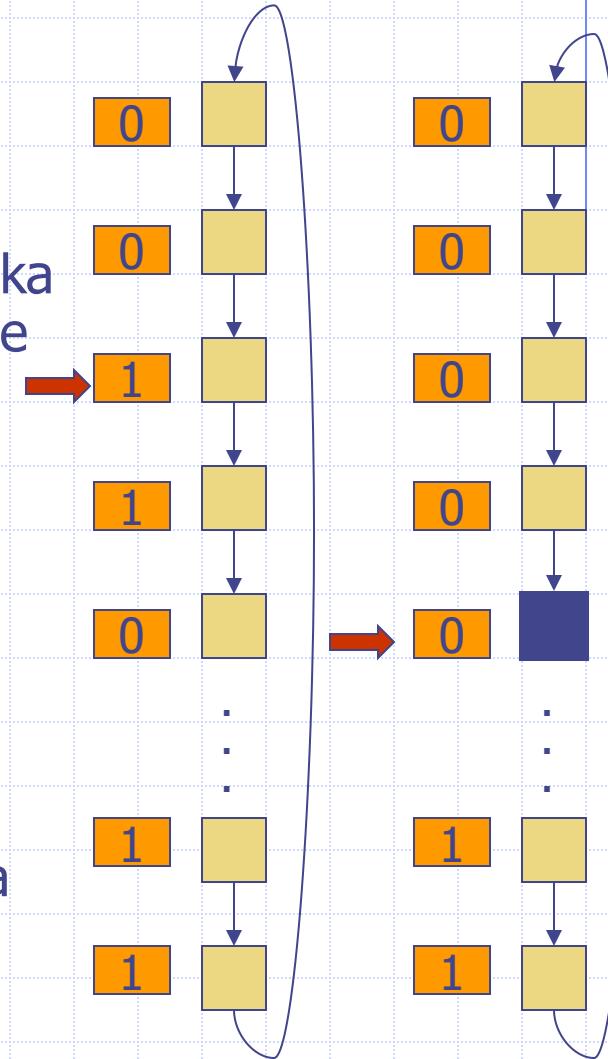
- ◆ Jednostavna HW podrška za grubu aproksimaciju LRU: *bit pristupa* pridružen svakoj stranici; prilikom učitavanja postavlja se na 0, prilikom pristupa postavlja se na 1
- ◆ Algoritam sa *dodatnim bitima referenciranja (additional-reference-bits algorithm)*:
  - HW manipuliše samo jednim bitom referenciranja
  - OS u deskriptoru stranice vodi evidenciju o istoriji ovih bita u registru
  - na regularne vremenske intervale (tajmerski prekid), OS pomera (*shift*) registar istorije udesno, odbacujući desni bit, a dodajući sleva bit referenciranja
  - ako se ovi registri istorije smatraju neoznačenim celim brojevima, izbacuje se stranica sa najmanjom vrednošću регистра (ne mora biti jedinstvena)



# Zamena stranica – LRU aproksimacije

◆ Algoritam *davanja nove šanse ili časovnika* (*second-chance algorithm, clock algorithm*):

- osnova je FIFO: postoji pokazivač (kao kazaljka sata) koji ide po kružnoj listi stranica i ukazuje na sledećeg kandidata za izbacivanje
- kada treba izbaciti stranicu, proverava se bit referenciranja; ako je on 0 (strаница nije коришћена од учитавања), та страница се избације; ако је он 1, FIFO pokazиваč се помера на sledeћу страницу (овој страници се дaje нова шанса, јер је била коришћена), а бит referenciranja се briše
- на тај начин се тражи даље, док се не нађе на страницу са битом referenciranja jednakим 0
- своди се на обичан FIFO ако су сви бити referenciranja били 1



# Zamena stranica – LRU aproksimacije

- ◆ *Prošireni algoritam davanja nove šanse (enhanced second-chance algorithm)*: umesto tretiranja samo bita referenciranja, posmatra se i bit zaprljanosti; uređeni par (*reference bit, modify bit*) može biti:
  - (0,0): najbolji kandidat za izbacivanje, jer stranica nije ni korišćena, niti modifikovana, pa ne mora ni da se snima na disk
  - (0,1): sledeći kandidat za izbacivanje, jer stranica nije korišćena (od brisanja bita ref.), ali je modifikovana, pa mora da se snima na disk
  - (1,0): sledeći kandidat za izbacivanje, jer stranica jeste korišćena, ali nije modifikovana, pa ne mora da se snima na disk
  - (1,1): najlošiji kandidat za izbacivanje, jer je stranica i korišćena i modifikovana

Umesto samo ispitivanja bita referenciranja, algoritam časovnika ispituje kojoj od ovih klasa stranica pripada i bira se ona stranica koja pripada najboljoj klasi pre nego što se traži stranica iz naredne klase (zahteva više obilazaka liste)

# Zamena stranica – Brojački algoritmi

- ◆ Postoje mnogi drugi algoritmi zamene. Jedan pristup je pridružiti svakoj stranici *brojač pristupa* koji se resetuje pri učitavanju stranice, a inkrementira pri svakom pristupu njoj
- ◆ Mogući algoritmi zamene:
  - *najređe korišćen* (*least frequently used*, LFU): izbacuje se stranica sa najmanjim brojačem, jer je najređe korišćena; problem je ako je stranica bila davno mnogo korišćena, a u skorije vreme nije; jedno rešenje: periodično pomerati udesno ove brojače
  - *najčešće korišćen* (*most frequently used*, MFU): izbacuje se stranica sa najvećim brojačem, računajući na to da je stranica sa najmanjim brojačem najskorije učitana
- ◆ Ovi algoritmi nisu mnogo korišćeni, jer im je implementacija složena, a ne aproksimiraju dobro OPT

# Zamena stranica – Ostale tehnike

- ◆ Mnoge druge tehnike se koriste pored samih algoritama zamene, kako bi se povećala njihova efikasnost:
  - sistemi obično održavaju *rezervoar (pool) slobodnih okvira*: kada se traži slobodan okvir, odmah je raspoloživ u bazenu, tako da se nova stranica učitava pre nego što se pokrene izbacivanje neke druge radi dopune rezervoar (moguće uraditi kasnije)
  - vodi se spisak modifikovanih stranica; kada je uređaj na kome se čuvaju stranice slobodan, pokreće se operacija snimanja modifikovanih stranica na disk, kako bi se u trenutku zamene povećala verovatnoća da je stranica koja se izbacuje “čista”
  - u rezervoaru slobodnih okvira sadržaj se ne menja; zbog toga, ako se traži ista stranica koja je bila izbačena u rezervoar slobodnih, može se ponovo iskoristiti ukoliko taj okvir nije bio upotrebljavan; ovaj pristup uspešno ispravlja nedostatke algoritama zamene (VAX VMS – FIFO, UNIX – SC)

# Alokacija okvira

- ◆ Maksimalni broj okvira koji se može dodeliti aktivnim procesima je broj trenutno slobodnih okvira, eventualno umanjen za veličinu rezervoara slobodnih okvira
- ◆ Koliki je najmanji broj okvira koje treba dodeliti procesu?
- ◆ U svakom slučaju, potrebno je da broj blokova dodeljenih procesu bude najmanje takav da može da se izvrši tekuća instrukcija (instrukcija se izvršava ispočetka prilikom svakog *page fault*)
- ◆ Za RISC procesore sa Load/Store arhitekturom, instrukcija referiše najviše instrukciju i operand, pa zahteva najviše jednu do četiri stranice (jednu do dve za instrukciju, još jednu za direktno adresiranje, ili još dve za indirektno adresiranje podatka)
- ◆ Za CISC procesora sa složenim instrukcijama, broj stranica koje referiše instrukcija može da bude veliki!

# Alokacija okvira

- ◆ Kako raspodeliti slobodne okvire procesima?
  - najjednostavnije: svima približno podjednako
  - proporcionalno njihovoj veličini (veličini virtuelnog adresnog prostora koji koriste)
  - prioritetnijim procesima dodeliti više okvira (da bi se brže izvršavali)
- ◆ Naravno, broj okvira dodeljenih procesima raste dinamički sa porastom stepena multiprogramiranja i obratno
- ◆ Zamena stranica može da bude:
  - lokalna: samo stranice procesa koji je generisao *page fault* učestvuju u izboru žrtve za izbacivanje; na izvršavanje nekog procesa (broj njegovih *page faults*) utiče samo ponašanje tog procesa
  - globalna: stranice svih procesa učestvuju u izboru žrtve za izbacivanje; na izvršavanje nekog procesa (broj njegovih *page faults*) utiču drugi procesi; efikasnije jer daje veću propusnost sistema i više je u upotrebi

# Thrashing

- ◆ Šta ako proces nema dovoljno okvira da završi tekuću instrukciju? Jedino rešenje – izbaciti ga celog (*swap out*)
- ◆ Inače, može da se desi sledeće:
  - tekuća instrukcija procesa izazove *page fault*, jer adresirana stranica nije u memoriji
  - nema slobodnog okvira, jer je cela memorija zauzeta
  - OS izbaci jednu stranicu tog procesa da bi učitao potrebnu
  - međutim, izbačena stranica je ubrzo ponovo potrebna, možda čak u istoj instrukciji
  - čim proces nastavi, ubrzo ponovo generiše *page fault* zahtevajući nedavno izbačenu stranicu
  - itd.
- ◆ Situacija kada proces više vremena provodi u zameni stranica nego u korisnom izvršavanju naziva se *batrganje* (*thrashing*)

# Thrashing

## ◆ Ponašanje koje su manifestovali rani sistemi:

- OS prati iskorišćenje procesora; kada ono padne, OS povećava stepen multiprogramiranja učitavanjem novih procesa
- OS upotrebljava globalnu politiku zamene stranica
- neki proces uđe u fazu kada zahteva više stranica; on generiše *page faults* i preotima okvire od drugih procesa
- i drugi procesi nastavljaju izvršavanje, tako da i oni izazivaju *page faults* i preoptimaju okvire
- pošto se povećava broj zamena stranica, I/O uređaj je sve zauzetiji, a procesor sve slobodniji
- OS zaključuje da je iskorišćenje procesora palo i povećava stepen multiprogramiranja!
- Posledica: propusnost sistema drastično pada, sistem praktično ne radi nikakav koristan posao

# *Thrashing*

CPU utilization

Degree of multiprogramming

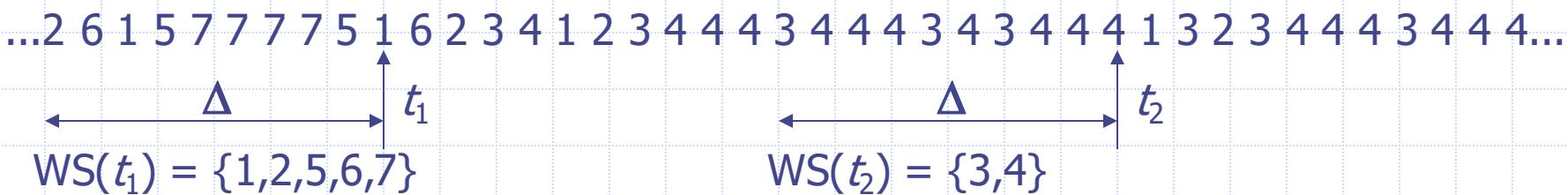
thrashing

# *Thrashing*

- ◆ *Thrashing* se može ograničiti upotrebom lokalne politike zamene stranica. Tada jedan proces ne može da preotima okvire od drugoga. Međutim, ako jedan proces uđe u *thrashing*, to utiče na efikasnost drugih procesa jer on opterećuje I/O uređaj za zamenu stranica
- ◆ Da bi se sprečio *thrashing*, potrebno je obezbediti da proces ima onoliko okvira koliko mu je trenutno potrebno
- ◆ Ali kako odrediti koliko je to potrebno?
- ◆ *Lokalitet (locality)* je skup stranica koje proces koristi zajedno
- ◆ *Model lokalnosti (locality model)* izvršavanja: tokom svog izvršavanja, proces prelazi sa jednog lokaliteta na drugi
- ◆ Primer: prilikom poziva potprograma, lokalitet čini deo sa kodom potprograma, sa lokalnim podacima na vrhu steka i sa delom globalnih podataka koje potprogram koristi; kada se izade iz potprograma, proces menja lokalitet
- ◆ Ideja: obezbediti procesu onoliko okvira koliko pokriva tekući lokalitet

# Thrashing

- ◆ Model radnog skupa (*working set model*) zasniva se na modelu lokalnosti:
  - $\Delta$  - parametar koji definiše veličinu *prozora radnog skupa* (*working-set window*)
  - skup stranica unutar sekvence od  $\Delta$  najskorije referenciranih stranica čini *radni skup*
  - radni skup je aproksimacija lokaliteta procesa, ograničena na  $\Delta$  poslednjih referenci:
    - ◆ ako je  $\Delta$  suviše mali, radni skup neće pokrivati ceo lokalitet
    - ◆ ako je  $\Delta$  suviše veliki, pokriće i neki neaktivni lokalitet
    - ◆ u ekstremnom slučaju, ako je  $\Delta$  beskonačno, radni skup je skup svih stranica koje je proces koristio tokom svog izvršavanja



# Thrashing

- ◆ Ukupna potreba svih aktivnih procesa i za okvirima ( $WSS_i$  je veličina radnog skupa procesa  $i$ ):

$$D = \sum WSS_i$$

- ◆ *Thrashing* nastaje kada je  $D$  veće od ukupnog broja raspoloživih okvira
- ◆ Ako je odabran  $\Delta$ , postupak tokom izvršavanja je sledeći:
  - OS prati radni skup svakog procesa i procesu dodeljuje onoliko okvira koliko je potrebno za njegov radni skup
  - ako preostaje slobodnih okvira, može se pokrenuti ili učitati (*swap in*) neki proces
  - ako zbir veličina radnih skupova preraste broj raspoloživih okvira, OS bira proces za suspendovanje i izbacivanje (*swap out*), a njegove okvire dodeljuje drugim procesima, da bi izbegao *thrashing*
- ◆ Ovakva tehnika izbegava *thrashing*, a optimizuje iskorišćenje procesora

# *Thrashing*

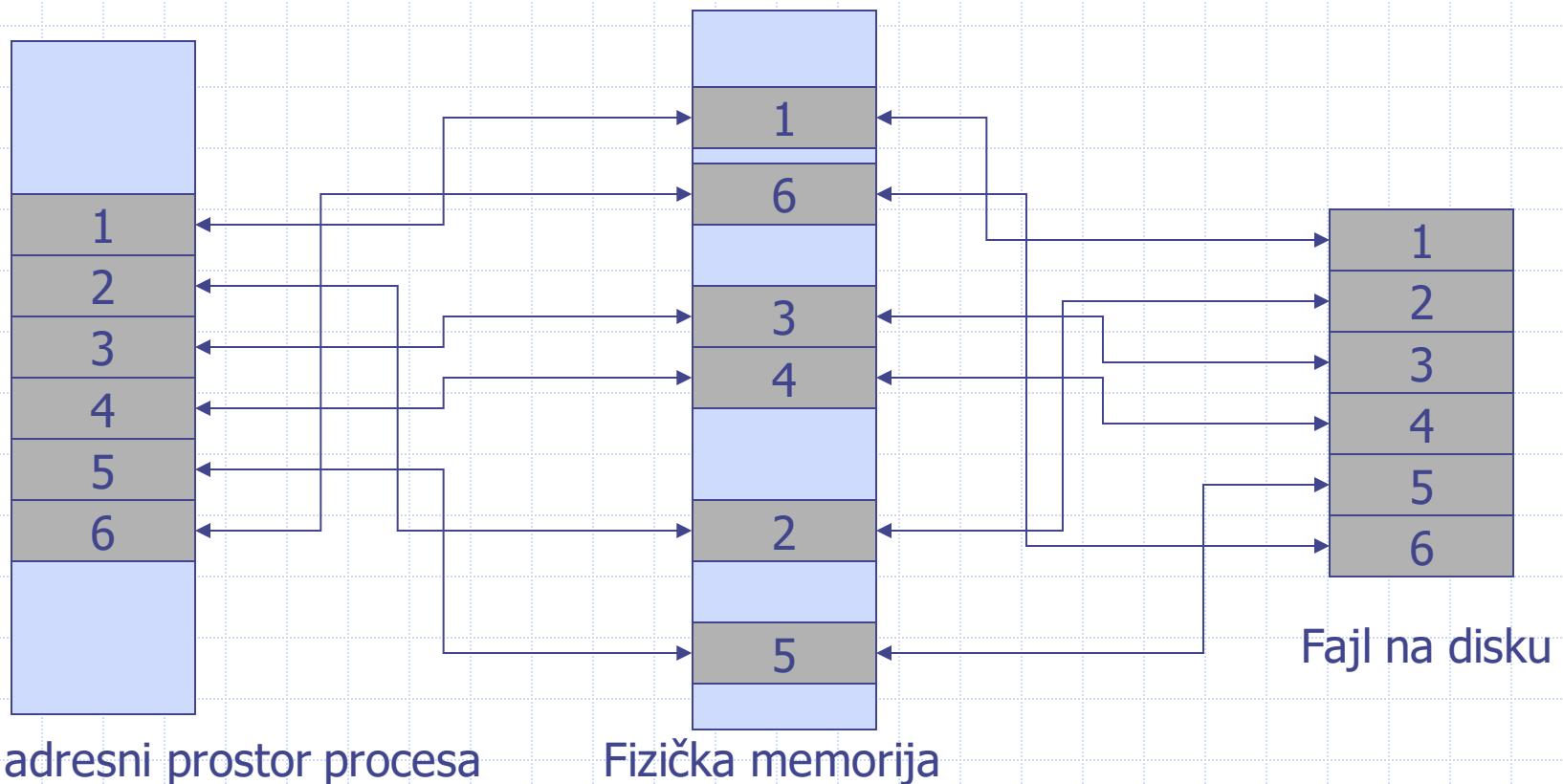
- ◆ Problem je implementacija: kako pratiti radni skup (skup stranica u sekvenci poslednje referisanih stranica dužine  $\Delta$ )? Potrebno ga je ažurirati prilikom svakog obraćanja stranici!
- ◆ Aproksimacija pomoću periodičnog prekida i bita referenciranja:
  - periodično se generiše prekid, tako da se  $\Delta$  odnosi na vreme, a ne na dužinu sekvence pristupa
  - OS periodično kopira u svoju evidenciju i potom briše bite referenciranja, detektujući tako radni skup u poslednjoj periodi
- ◆ Izbegavanje *thrashing*-a pomoću tehnike radnog skupa je efikasan, ali komplikovan metod

# Thrashing

- ◆ Tehnika izbegavanja praćenjem *frekvencije page fault-ova* (*page-fault frequency*, PFF):
  - OS prati učestanost pojave *page fault* za procese
  - kada ona poraste preko određene mere, proces zahteva više okvira i OS mu to obezbeđuje
  - kada ona padne ispod neke granice, proces ima previše okvira i OS mu oduzima okvire
  - ako ona poraste preko određene granice, a nema slobodnih okvira, OS izbacuje neki proces (*swap out*) da bi sprečio *thrashing*

# Memorijski preslikani fajlovi

- Ideja: koristiti mehanizam virtuelne memorije da bi se pristup fajlu vršio kao običan pristup memoriji; logički pridružiti deo virtuelnog adresnog prostora fajlu – *memorijski preslikani fajl (memory-mapped file, MMF)*



# Memorijski preslikani fajlovi

## ◆ Princip:

- blok na disku preslikava se u jednu ili više stranica u memoriji
- inicijalni pristup fajlu, kao pristup stranici, rezultuje *page fault*-om
- OS dovlači deo fajla kao jednu ili više stranica
- program dalje pristupa svom adresnom prostoru direktno, ne preko sistemskih poziva za čitanje ili pisanje fajla
- OS preslikava stranice u delove fajla
- upis u stranice nije obavezno sinhroni upis u fajl, već odloženi upis prilikom zamene stranice ili prilikom periodičnog snimanja "zaprljanih" stranica

## ◆ Neki OS obezbeđuju memorijski preslikane fajlove posebnim sistemskim pozivima

## ◆ Neki OS sve pristupe fajlovima obavljaju ovom tehnikom (npr. preslikavanjem u adresni prostor kernela)

## ◆ Mnogi OS omogućavaju deljenje MMF između procesa

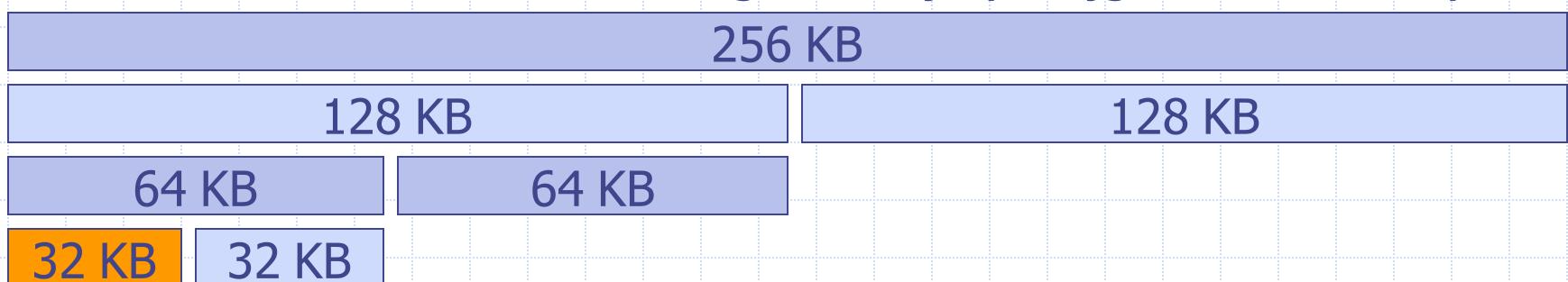
# Alokacija memorije za jezgro OS

- ◆ Memorija za jezgro (*kernel*) OS alocira se iz posebnog skupa okvira, ne iz onog namenjenog za korisničke procese jer:
  - jezgro mora pažljivo, kompaktno i štedljivo da koristi prostor, da bi izbegao internu fragmentaciju
  - neki I/O uređaji koriste kontinualne delove fizičke memorije, pa susedne stranice moraju ponekad biti susedne u fizičkoj memoriji

# Alokacija memorije za jezgro OS

## ◆ Sistem *parnjaka* (*buddy system*):

- memorija se alocira iz fizički kontinualnog segmenta memorije
- memorija se zauzima u komadima veličine stepena dvojke (npr. 4KB, 8KB, 16KB, ...); manji komadi se zaokružuju na prvu veću jedinicu
- kontinualni raspoloživi segment se deli na parnjake jednake veličine
- zahtev se zadovoljava u najmanjem slobodnom parnjaku u koji može da se smesti; ako je on veći od zahtevanog, deli se rekursivno na dva dela, sve dok se ne dobije slobodan blok zahtevane veličine
- pogodnost je što se dva susedna slobodna parnjaka brzo i lako spajaju u dva puta veći slobodan segment
- nedostatak: velika interna fragmentacija (u najgorem i do 50%)



# Alokacija memorije za jezgro OS

## ◆ Sistem ploča (*slab allocation*):

- ploča (*slab*) se sastoji od jedne ili više fizički susednih stranica
- keš (*cache*) se sastoji od jedne ili više ploča
- svaki keš služi za smeštanje svih instanci neke od struktura podataka koje koristi jezgro; npr. jedan keš za PCBove, jedan za semafore, jedan za FCBove itd.
- broj odeljaka za objekte unutar ploče zavisi od veličine objekta
- inicijalno su odeljci za objekte unutar jedne ploče svi označeni kao slobodni; kada se u odeljak smesti objekat, označava se zauzetim
- kada se traži prostor za novi objekat, on se traži sledećim redom:
  - ◆ najpre unutar delimično popunjene ploče
  - ◆ zatim unutar prazne ploče
  - ◆ inače, ako takvih nema, zauzima se nova ploča i dodeljuje se kešu
- pogodnosti: nema fragmentacije (objekti iste veličine se smeštaju u tačno dimenzionisane odeljke), efikasan postupak

# Dohvatanje stranica unapred

- ◆ Kada se novi proces aktivira ili se reaktivira posle izbacivanja (*swap in*), inicijalno će generisati mnogo *page fault*-ova
- ◆ Ideja za eliminaciju ovog problema: dohvati stranice koje će verovatno biti potrebne unapred (*prepaging*), prilikom aktivacije procesa, sve odjednom
- ◆ Jedan pristup: kada se proces suspenduje (*swap out*), sačuva se informacija o njegovom radnom skupu; kada se ponovo učita (*swap in*), učitavaju se sve stranice iz njegovog radnog skupa
- ◆ Pitanje je da li je ovo isplativije od dohvatanja stranica na zahtev, ukoliko veći broj unapred učitanih stranica neće biti korišćen

# Zaključavanje stranica

- ◆ I/O operacija koju vrši poseban uređaj (DMA, koprocesor) koristi bafer za prenos. Potencijalni problem:
  - korisnički proces započne prenos i suspenduje se čekajući na završetak
  - OS pokrene drugi proces koji zatraži stranicu
  - okvir u kome se nalazi dodeljeni I/O bafer bude dodeljen drugom procesu
- ◆ Rešenja:
  - baferi za I/O operacije se smeštaju u prostor jezgra, ne korisničkih procesa; dodatne režije za prepisivanje bafera iz korisničkog prostora u prostor jezgra
  - *zaključavanje stranice (interlock)*: posebnim bitom stranica se označi zaključanom – zabranjuje se njena zamena

# Uticaj programa na performanse

- ◆ Posmatrajmo sledeći (veštački, ali ilustrativan) primer:

```
const int N = ...;
int data[N][N];
for (int j=0; j<N; j++)
    for (int i=0; i<N; i++)
        data[i][j]=0;
```

- ◆ Šta se dešava ako je  $N$  jednako veličini stranice (broju reči u stranici) i ako OS odvoji manje od  $N$  okvira za ovaj proces? – Može da rezultuje sa  $N^2$  *page fault*-ova!
- ◆ Nasuprot tome, sledeća transformacija programa daje svega  $N$  *page fault*-ova:

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        data[i][j]=0;
```

# Uticaj programa na performanse

- ◆ Zaključak: ponašanje i struktura programa može tako da utiče na performanse izvršavanja
- ◆ Lokalnost ponašanja programa poboljšava performanse. Na lokalnost utiču korištene strukture. Na primer:
  - stek ima dobru lokalnost (stalno radi oko vrha)
  - *hash* tabela ima slabu lokalnost (ima zadatak da rasipa ključeve)
- ◆ Prevodilac ima značajnu ulogu u organizaciji programa:
  - razdvajanjem koda od podataka u različite stranice čini da su stranice sa kodom uvek nepromenjene i ne moraju se snimati na disk prilikom zamene
  - potprogrami koji se intenzivno međusobno pozivaju daju bolje performanse ako su u istoj stranici

# Glava 6: Upravljanje diskovima

Struktura diska

Konfiguracije priključivanja diskova

Raspoređivanje zahteva

Rukovanje prostorom na disku

Rukovanje prostorom za zamenu

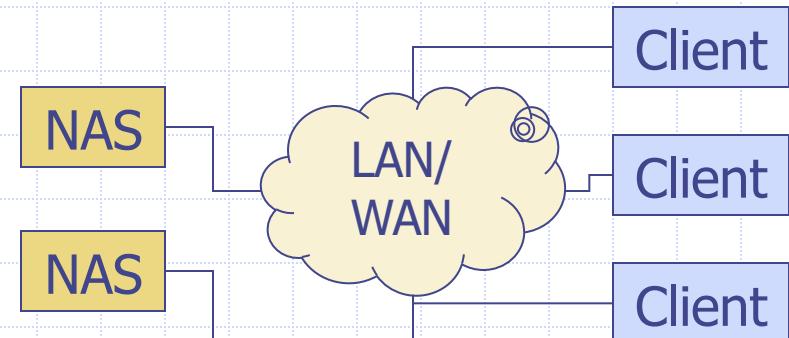
RAID strukture

# Struktura diska

- ◆ Disk se logički posmatra kao linearan, sekvenčijalan niz blokova-sektora:
  - blokovi su tipično veličine 512B
  - blokovi su redom numerisani, tako da su blokovi sa susednim brojem fizički susedni: blok 0 je na sektoru 0 spoljnog cilindra, zatim prema susednom sektoru na istom cilindru itd., zatim prema narednom cilindru ka unutrašnjosti itd.
- ◆ Iako deluje trivijalno, preslikavanje logičke adrese bloka u njegovu fizičku poziciju na disku nije jednostavno:
  - diskovi obično sadrže loše sektore (*bad sector*) koji nisu upotrebljivi
  - broj sektora po cilindru za neke diskove nije isti za sve cilindre: gustina zapisa je jednaka, pa je broj sektora na unutrašnjim cilindrima manji (tipično i do 40%); zbog toga uređaj povećava brzinu obrtanja kada glava ide ka unutrašnjim cilindrima

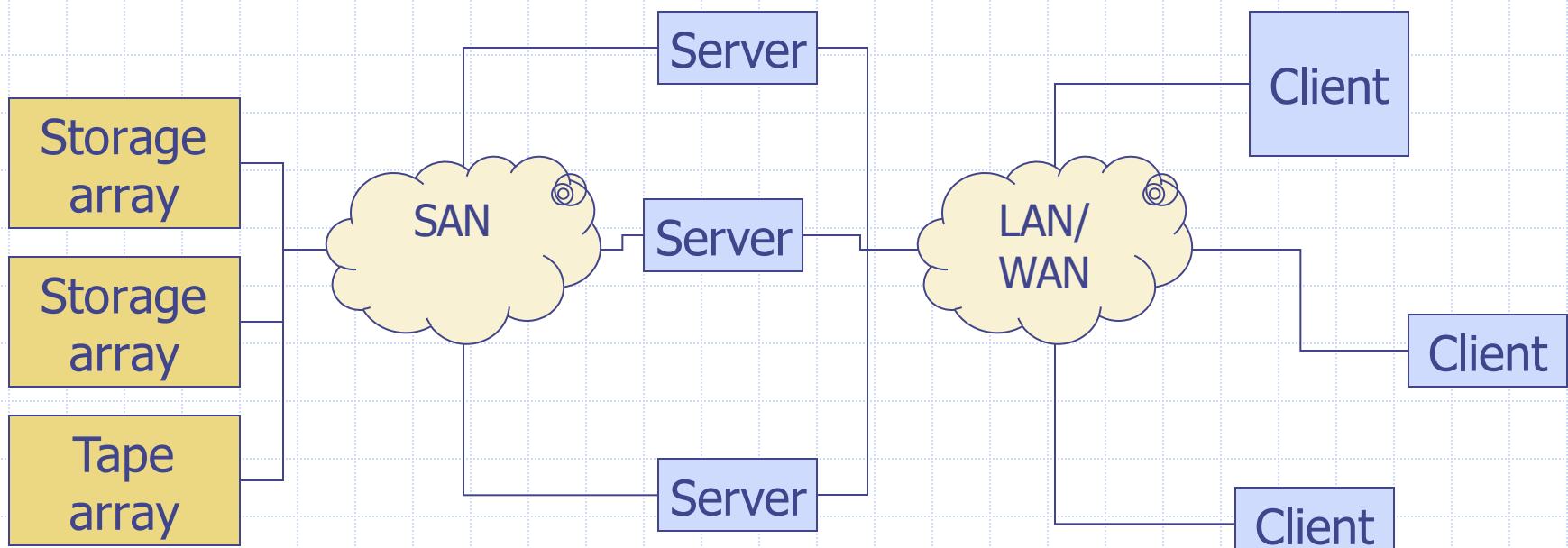
# Konfiguracije priključivanja diskova

- ◆ Direktno priključenje diska na računar-domaćin (*host-attached storage*):
  - diskovi se priključuju na sam računar-domaćin, tako da im računar pristupa preko I/O portova
  - današnji standardi: IDE (do dva uređaja po I/O magistrali), SCSI (do  $16 = 1 + 15$  uređaja po magistrali), FC (serijska magistrala preko optičkog kabla), SATA
- ◆ Priključivanje preko mreže (*network-attached storage, NAS*):
  - računari pristupaju priključenim diskovima preko lokalne IP mreže, koristeći mrežne protokole (RPC)
  - NAS implementira fajl sistem i klijenti koriste udaljeni fajl sistem



# Konfiguracije priključivanja diskova

- ◆ Priključivanje na mrežu namenjenu samo za pristup diskovima (*storage-area network, SAN*):
  - diskovi se priključuju na posebnu lokalnu mrežu namenjenu samo za komunikaciju sa diskovima
  - SAN nudi klijentima samo interfejs blokovskog pristupa, ne implementira fajl sistem (to je odgovornost klijenta)



# Raspoređivanje zahteva – FCFS

- ◆ Zadatak: rasporediti zahteve za operacijama sa diskom koji čekaju u redu za disk tako da operacije sa diskom za procese budu najefikasnije; iz reda zahteva izabratи onaj koji će sledeći biti opslužen
- ◆ *First-Come, First Served* (FCFS): zahteve opsluživati po redosledu kojim su pristigli
- ◆ Jednostavan, ali neefikasan
- ◆ Primer:
  - sekvenca zahteva (brojevi cilindara): 98, 183, 37, 122, 14, 124, 65, 67; glava je na početku na cilindru 53  
 $53 \rightarrow^{45} 98 \rightarrow^{85} 183 \rightarrow^{146} 37 \rightarrow^{85} \underline{122} \rightarrow^{108} \underline{14} \rightarrow^{110} \underline{124} \rightarrow^{59} 65 \rightarrow^2 67$
  - ukupno pređeni put glava je 640 cilindara
- ◆ FCFS rezultuje velikim šetanjem glava diska po cilindrima, pa time povećava ukupno vreme opsluživanja

# Raspoređivanje zahteva – SSTF

- ◆ *Shortest-Seek-Time-First* (SSTF): opsluživati najpre zahteve koji se odnose na cilindre najbliže tekućoj poziciji glava – opslužiti zahtev koji ima najkraće vreme pretrage za tekuću poziciju (najbliži je)
- ◆ Primer:
  - sekvenca zahteva (brojevi cilindara): 98, 183, 37, 122, 14, 124, 65, 67; glava je na početku na cilindru 53  
 $53 \rightarrow^{12} 65 \rightarrow^2 67 \rightarrow^{30} 37 \rightarrow^{23} 14 \rightarrow^{84} 98 \rightarrow^{24} 122 \rightarrow^2 124 \rightarrow^{59} 183$
  - ukupno pređeni put glava je 236 cilindara
- ◆ SSTF je analogan SJF za procese i zato može da uzrokuje izgladnjivanje nekih zahteva (kako?)
- ◆ SSTF je značajno bolji nego FCFS, ali nije optimalan; za dati primer, bolje bi bilo sa 53 ići na 37 pa na 14, pa onda naviše – ukupno pređeni put bio bi 208

# Raspoređivanje zahteva – SCAN

- ◆ *SCAN*: glava se kreće u jednom smeru, opslužujući zahteve koji se odnose na cilindre koje prolazi, stiže do kraja, onda kreće u drugom smeru
- ◆ Glava se ponaša kao lift u zgradi (*elevator algorithm*): ide naniže i usput opslužuje sve pozive u tom smeru, a potom ide naviše
- ◆ Primer:
  - sekvenca zahteva (brojevi cilindara): 98, 183, 37, 122, 14, 124, 65, 67; glava je na početku na cilindru 53 i ide naniže (ka 0)  
 $53 \rightarrow^{16} 37 \rightarrow^{23} 14 \rightarrow^{14} 0 \rightarrow^{65} 65 \rightarrow^2 67 \rightarrow^{31} 98 \rightarrow^{24} 122 \rightarrow^2 124 \rightarrow^{59} 183$
  - ukupno pređeni put glava je 236 cilindara

# Raspoređivanje zahteva – C-SCAN

- ◆ Prepostavljući uniformnu raspodelu zahteva po cilindrima i SCAN algoritam, kada glava stigne do jednog kraja i promeni smer, veoma malo ima zahteva koji su tu odmah ispred glave, jer je ona nedavno tuda prošla. Najviše je zahteva na drugom kraju koji je još daleko, pa će oni dugo čekati. Zašto ne ići odmah na taj kraj?
- ◆ *C-SCAN (circular scan)*: modifikacija SCAN algoritma tako što se glava kreće cirkularno, a ne gore-dole; kada stigne do kraja diska, prelazi odmah na drugi kraj (bez opsluživanja zahteva, brzo) i uvek se kreće u istom smeru
- ◆ Ima uniformnije vreme čekanja zahteva nego SCAN algoritam
- ◆ Primer:

sekvenca zahteva (brojevi cilindara): 98, 183, 37, 122, 14, 124, 65, 67; glava je na početku na cilindru 53

53 → 65 → 67 → 98 → 122 → 124 → 183 → Max → 0 → 14 → 37

# Raspoređivanje zahteva – LOOK

- ◆ Kod SCAN ili C-SCAN algoritama, zašto ići do kraja diska ako više do kraja nema zahteva – može se odmah promeniti smer, odnosno preći na prvi zahtev na drugom kraju
- ◆ *LOOK* i *C-LOOK*: modifikacija SCAN i C-SCAN algoritama tako što glava ne ide do kraja diska, već do poslednjeg/prvog zahteva na tom kraju
- ◆ Primer:

sekvenca zahteva (brojevi cilindara): 98, 183, 37, 122, 14, 124, 65, 67;  
glava je na početku na cilindru 53

C-LOOK:

53 → 65 → 67 → 98 → 122 → 124 → 183 → 14 → 37

# Raspoređivanje zahteva

## ◆ Kako izabrati najbolji algoritam?

- za svaku sekvencu zahteva može se izračunati optimalan raspored, ali je cena tog izračunavanja neisplativa u odnosu na dobit koju donosi optimalan raspored u odnosu na neki podrazumevani dobar raspored (npr. SSTF ili SCAN)
- za svaki izabrani algoritam, učinak jako zavisi od prirode zahteva
- na prirodu zahteva jako utiče i organizacija direktorijuma, metod alokacije fajlova, način pristupa do fajlova i način keširanja direktorijuma i fajlova
- prikazani algoritmi uzimaju u obzir samo vreme pomeranja glave (*seek time*); kod modernih diskova vreme rotacije do traženog sektora je srazmerno ovom vremenu; veoma je teško da OS optimizuje i ovo vreme, pošto diskovi sakrivaju način rasporeda logičkih blokova
- moderni disk-kontroleri implementiraju sami svoj algoritam raspoređivanja, prilagođen organizaciji samog diska
- OS može da tretira druge prioritete: zahteve za stranicama u odnosu na pristup fajlovima, snimanje u odnosu na čitanje itd.

# Rukovanje prostorom na disku

- ◆ *Fizička formatizacija*, formatizacija niskog nivoa (*physical formatting, low-level formatting*): organizovanje sektora i upisivanje struktura podataka na svaki sektor diska:
  - zaglavlje (*header*) i rep (*trailer*): informacije koje koristi disk-kontroler, npr. broj sektora i kod za korekciju greške (*error-correction code, ECC*)
  - deo za podatke, tipično 512B
- ◆ ECC:
  - kontroler diska ažurira ECC prilikom svakog upisa podataka u sektor
  - prilikom svakog čitanja podataka sa sektora, kontroler diska izračunava ECC i poredi ga sa zapisanim; ako postoji razlika, sektor je možda loš (*bad sector*)
  - ECC sadrži informacije koje omogućuju (ograničenu) korekciju greške, tako da se u određenim slučajevima delimično uništeni podaci mogu restaurirati i eventualno prenesti na drugi sektor
- ◆ Većina diskova je fizički formatirano u samoj proizvodnji

# Rukovanje prostorom na disku

- ◆ Neki kontroleri diskova dozvoljavaju da im se komandom za fizičku formatizaciju zada i veličina prostora za podatke na sektoru (iz skupa dozvoljenih veličina)
- ◆ Da bi pripremio disk, OS vrši:
  - *particioniranje (partitioning)*: podelu fizičkog diska na grupe cilindara – particije; svaka particija se logički tretira kao zaseban disk od strane OS
  - *logičko formatiranje (logical formatting)*: kreiranje fajl-sistema, tj. organizovanje inicijalnih struktura podataka na particiji za fajl sistem i grupisanje sektora u klastere
- ◆ *Boot blok na disku*:
  - u ROMu se nalazi samo mali, jednostavan *bootstrap* program koji samo učitava i pokreće veći *bootstrap* program sa *boot* bloka
  - disk na kome postoji *boot* blok naziva se *sistemski disk (system disk, boot disk)*
  - veliki *bootstrap* program učitava ceo OS sa proizvoljnog mesta na disku i pokreće ga

# Rukovanje prostorom na disku

- ◆ Ponekad se magnetni sloj na nekom sektoru diska nepovratno ošteti. Ponekad je sektor neispravan u samoj proizvodnji
- ◆ Takav sektor je neupotrebljiv i označava se kao *loš sektor* (*bad sector*)
  - jednostavniji diskovi zahtevaju da se pokrene poseban program (`chkdsk` u MS DOS) i pronađe loše sektore i označi ih u FAT kao neupotrebljive; podaci na ovakvim sektorima su obično izgubljeni
  - kod boljih diskova (SCSI) sam disk-kontroler vodi računa o lošim sektorima; listu loših sektora inicijalizuje tokom fizičke formatizacije i održava je tokom korišćenja diska; loš sektor se može logički zameniti ispravnim (*sector sparing, forwarding*): kontroler dodeljuje bivšu adresu lošeg sektora ispravnom slobodnom sektoru, po mogućству na istom cilindru – šta je sa algoritmima raspoređivanja ako to nije slučaj?!  
Alternativa – pomeriti zauzete blokove za po jedno mesto (*sector slipping*), tako da se loš sektor preskoči

# Rukovanje prostorom za zamenu

- ◆ Virtuelna memorija koristi prostor na disku kao proširenje operativne memorije, jer na njega izbacuje zamenjene stranice (*swapping*)
- ◆ Pošto je zamena stranica kritična operacija, a pristup disku spor, organizacija i rukovanje prostorom za zamenu (*swap space*) stranica na disku je važno
- ◆ Važno je da prostor za zamenu bude dovoljno veliki, jer inače sistem mora da gasi procese ili će potpuno krahirati; preporuke:
  - Solaris: prostor za zamenu treba da bude onoliki koliko virtuelni prostor prevaziđa fizički OM prostor raspoloživ za stranice
  - Linux: istorijski – prostor za zamenu treba da bude dva puta veći nego fizički OM prostor, mada noviji Linux sistemi smanjuju potrebe; Linux dozvoljava višestruke prostore za zamenu (na više diskova)

# Rukovanje prostorom za zamenu

- ◆ Prostor za zamenu može biti lociran:
  - u okviru standardnog fajl sistema OS-a, kao običan veliki fajl; OS kreira, imenuje i alocira taj veliki fajl korišćenjem standarnih fajl operacija, a onda ga koristi za smeštanje zamenjenih stranica; lako za implementaciju, ali neefikasno
  - na presnoj (*raw*) particiji: poseban deo OS-a vodi računa o alokaciji i dealokaciji prostora za zamenjene stranice direktno na sektorima ove particije; algoritmi su optimizovani na brzinu, ne na efikasnu upotrebu prostora na disku
- ◆ Neki OS (Linux) dozvoljavaju obe varijante – izbor je na administratoru
- ◆ Neki OS (Solaris, Linux) smeštaju samo zamenjene stranice (koje se upisuju iz OM, podaci) u prostor za zamenu, dok stranice sa kodom prosto izbacuju i ponovo učitavaju iz fajla programa

# RAID strukture

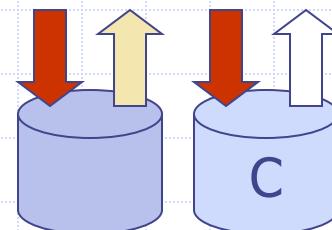
- ◆ Vremenom su diskovi postali sve manji i jeftiniji, pa je sasvim priuštivo imati više diskova na istom računaru
- ◆ Postojanje više diskova na istom računaru dozvoljava da se oni iskoriste, umesto samo zbog povećanja prostora, za:
  - povećanje pouzdanosti, zbog postojanja redundantnih diskova
  - poboljšanje performansi, zbog mogućeg paralelizma u radu
- ◆ Čitav niz tehnika organizacije više diskova koje imaju ovo za cilj, upotrebom više (jeftinih) diskova umesto jednog skupog, nazivaju se *RAID* (*redundant arrays of inexpensive disks*)
- ◆ Danas poenta više nije u ceni, jer su diskovi svakako jeftini, već u poboljšanju pouzdanosti i performansi, pa zato "I" više ne znači *inexpensive* nego *independent* (nezavisni)

# RAID strukture

- ◆ RAID diskovi mogu da budu priključeni:
  - na standadnu I/O magistralu na uobičajeni način; tada OS mora da implementira RAID funkcionalnost
  - preko posebnog hardverskog kontrolera koji implementira RAID funkcionalnost, tako da je RAID transparentan za OS i softver i može da se koristi na bilo kom OS bez RAID funkcionalnosti

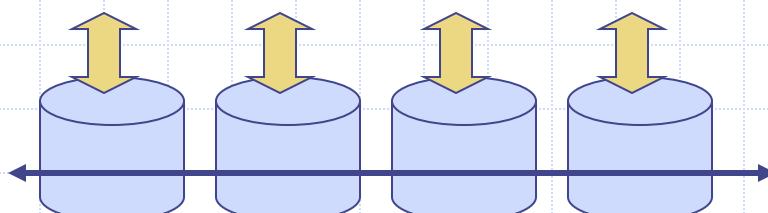
# RAID strukture – Povećanje pouzdanosti

- ◆ Ako bi se  $N$  diskova koristilo samo za povećanje prostora  $N$  puta, bez redundantne podatka, onda se srednje vreme između otkaza sistema skratilo  $N$  puta u odnosu na srednje vreme između otkaza jednog diska; npr. ako je MTBF (*mean time between failure*, srednje vreme između otkaza) jednog diska 100.000 sati, onda je MTBF za 100 diskova 1000 sati, odnosno oko 42 dana, što je neprihvatljivo!
- ◆ Rešenje: višestruke diskove koristiti za smeštanje redundantnih podataka, tako da u slučaju otkaza nekog diska ne dolazi do gubitka informacija
- ◆ Jedan pristup – *ogledanje (mirroring)*: jedan logički disk sastoji se od dva fizička diska sa identičnim kopijama; svaki upis se vrši identično na oba diska; ako jedan otkaže, drugi je raspoloživ do popravke/zamene
- ◆ Upis je dupliran, čitanje može iz bilo kog – može se paralelizovati



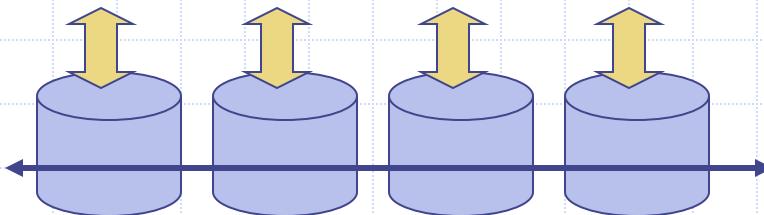
# RAID strukture – Poboljšanje performansi

- ◆ Poboljšanje performansi tehnikom *pruga (data striping)* - rasporediti podatke sukcesivno na susedne diskove:
  - *bit-level striping*: po jedan bit svakog bajta rasporediti na 8 diskova
  - *block-level striping*: susedni blokovi fajla su raspoređeni na susedne diskove; najčešće primenjivano
- ◆ Sa  $N$  diskova, povećava prostor  $N$  puta, ali i poboljšava performanse:
  - za više pristupa malim komadima, povećava propusnu moć, jer se više malih zahteva može paralelizovati na više diskova
  - za jedan pristup većem komadu, smanjuje vreme odziva jer deli pristup na više diskova koji rade paralelno

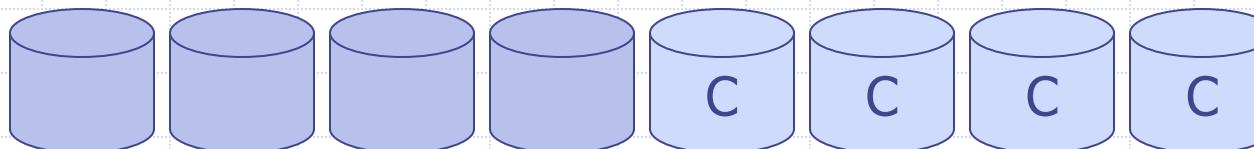


# RAID strukture – Nivoi

- ◆ RAID nivoi: različite RAID konfiguracije koje poboljšavaju pouzdanost i/ili performanse
- ◆ RAID 0: *block striping* bez ikakve redundanse; pogodno za zahtevne aplikacije kod kojih pouzdanost nije bitna

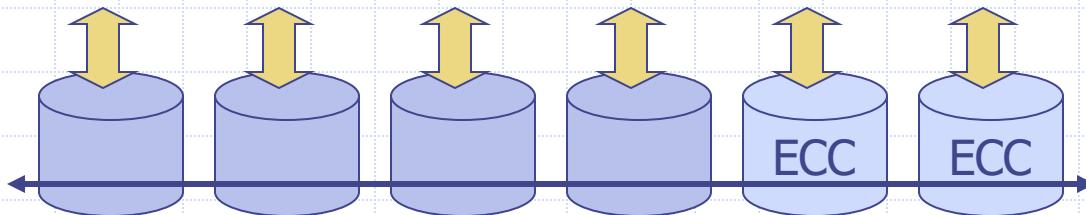


- ◆ RAID 1: *mirroring* bez ikakve paralelizacije; jeftina i jednostavna konfiguracija za servere koji zahtevaju pouzdanost (C – kopija)

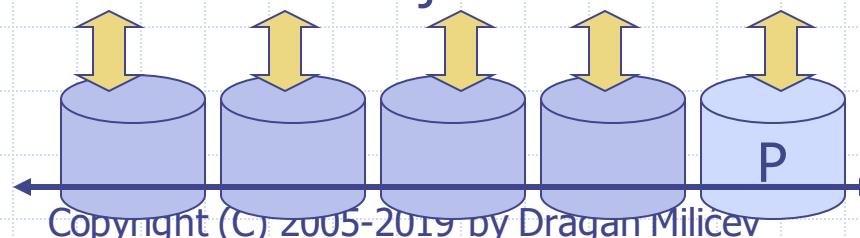


# RAID strukture – Nivoi

- ◆ RAID 2 (*memory-style error-correcting-code organization*): koristi *bit-striping* pri čemu se dodatni diskovi koriste za smeštanje ECC; ne koristi se (prevaziđen od strane RAID 3)



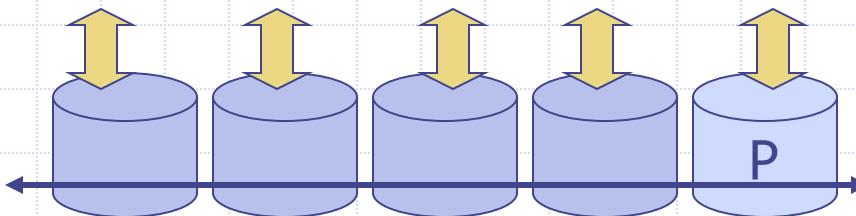
- ◆ RAID 3 (*bit-interleaved parity organization*):
  - ◆ svaki pojedinačni disk-kontroler zna da li je njegov pročitani sektor korektan ili ne
  - ◆ zbog toga je *bit parnosti* (*parity bit*, P) dovoljan i za detekciju i za korekciju greške, pošto se tačno zna koji bit je pogrešan
  - ◆ kao RAID 2, samo što koristi samo jedan dodatni disk kao redundans za bit parnosti



# RAID strukture – Nivoi

- ◆ RAID 4 (*block-interleaved parity organization*): kao RAID 3, samo koristi *block striping*; jedan dodatni disk čuva blok parnosti

- čitanje jednog bloka obrađuje jedan disk, ostali mogu da obrađuju druge zahteve u paraleli
- čitanje i upis velikih segmenata obavlja se paralelizovano
- upis jednog bloka ili nezavisni upisi po jednog bloka jesu problem: ne mogu se obavljati u paraleli, a svaki zahteva čitanje bloka, izmenu i upis, i to i za podatke i za blok parnosti



# RAID strukture – Nivoi

- ◆ RAID 5 (*block-interleaved distributed parity*): kao RAID 4, samo što podatke i bite parnosti rasipa po svim diskovima – za svaki paket od  $N$  blokova, jedan disk (bilo koji) čuva parnost, ostali podatke, ciklično
- ◆ RAID 6 (*P+Q redundancy scheme*): kao RAID 5, ali umesto parnosti, čuva dodatne redundantne ECC kako bi se zaštitio od otkaza više od jednog diska
- ◆ RAID 0+1: kombinacija RAID 0 i RAID 1, da bi dobio i na pouzdanosti i na performansama; generalno, ponaša se bolje nego RAID 5, ali udvostručuje broj diskova
  - RAID 0+1:  $N$  diskova u pruzi (*stripe*), i još  $N$  takvih u drugoj pruzi koja je ogledalo (*mirror*)
  - RAID 1+0:  $N$  parova diskova u ogledalu (*mirrored*) povezani u prugu (*stripe*)

# RAID strukture – Implementacija

## ◆ Implementacija RAID u softveru (OS):

- ne zahteva nikakve dodatne usluge hardvera, pa se mogu koristiti obični diskovi
- RAID nivoi koji koriste parnost nisu efikasni za implementaciju u softveru zbog stalnog računanja parnosti, pa se primenjuju tipično RAID 0, 1 ili 0+1

## ◆ Implementacija RAID u hardveru:

- unutar adaptera računara-domaćina: dozvoljava da se obični diskovi vežu na adapter; jeftino ali nefleksibilno
- unutar posebnog uređaja koji sadrži niz RAID diskova

## ◆ Najčešće korišćeni RAID nivoi:

- RAID 0 za zahtevne aplikacije visokih performansi bez potrebe za pouzdanostu
- RAID 1 za nezahtevne sisteme povišene pouzdanosti
- RAID 0+1 ili 1+0 za sisteme koji zahtevaju i performanse i pouzdanost (npr. baze podataka)
- RAID 5 za veoma velika skladišta podataka

# Glava 7: Arhitektura operativnih sistema

Usluge OS

Korisnički interfejs prema OS

Sistemski pozivi

Sistemski programi

Struktura OS

Virtuelne mašine

# Usluge OS

- ◆ Korisnički interfejs (*user interface, UI*):
  - interpreter komandne linije (*command line interpreter, CLI*)
  - *batch*: komande i direktive se zapisuju u fajl koji se izvršava
  - grafički korisnički interfejs (*graphical UI, GUI*)
- ◆ Izvršavanje programa
- ◆ I/O operacije
- ◆ Manipulacije fajlovima
- ◆ Komunikacija između procesa na istom i različitim računarima
- ◆ Detekcija grešaka
- ◆ Alokacija resursa
- ◆ Zaštita i sigurnost
- ◆ Računovodstvo: praćenje upotrebe resursa od strane korisnika

# Korisnički interfejs prema OS

## ◆ Komandna linija i interpreter (CLI):

- neki OS sadrže interpreter komandne linije unutar kernela
- kod drugih OS interpreter je samo sistemski program (Windows XP, UNIX, Linux)
- takvi OS mogu ponuditi različite interpretere – školjke (*shells*)
- Linux shells: *Bourne shell*, *C shell*, *Bourne-Again shell*, *Korn shell* itd.
- neki interpreteri u sebi sadrže kod za izvršavanje komandi; komanda se interpretira i skače se na taj kod koji uključuje sistemski poziv
- kod nekih interpretera (UNIX) većina komandi se izvršava kao sistemski program; interpreter samo interpretira komandu, zatim traži i pokreće program imenovan u komandi, prosleđujući mu parametre iz komandne linije; npr.: `rm file.txt`; fleksibilno, jer se interpreter ne menja dodavanjem komandi i veoma je jednostavan

# Korisnički interfejs prema OS

## ◆ GUI:

- *dekstop*, ikonice koje predstavljaju objekte u sistemu (uredaji, fajlovi, programi), intuitivne operacije (miš, tastatura)
- prvi put napravljeni ranih 1970ih u Xerox PARC – Xerox Alto, 1973.
- značajnu popularnost dobijaju 80ih, sa Apple Macintosh računarima (Mac OS)
- Microsoft Windows 1.0 uvodi GUI iznad DOS-a krajem 1980ih, a kasnije verzije Windows unapređuju GUI
- UNIX je tradicionalno orijentisan na CLI, iako postoji mnogo GUI školjki za UNIX: Common Desktop Environment (CDE), X-Windows
- UNIX/Linux *open-source* GUI školjke: *K Desktop Environment* (KDE), *GNOME*

## ◆ Izbor između CLI i GUI je stvar ličnog preferencijala:

- UNIX/Linux korisnici su tradicionalno orijentisani ka CLI
- Windows korisnici su orijentisani ka GUI
- Mac OS inicijalno nije imao CLI, već samo GUI, a sada ima oba!

# Sistemski pozivi

- ◆ *Sistemski poziv (system call)* predstavlja interfejs kojim OS programima nudi pristup do neke svoje usluge
- ◆ Primer: jednostavan program koji kopira jedan fajl u drugi koristi mnogo sistemskih poziva:
  - interaktivno učitavanje naziva fajlova sa tastature
  - otvaranje fajla za čitanje i kreiranje fajla za upis
  - čitanje i upis u fajl
  - ukoliko je došlo do greške u nekim od ovih poziva (npr. pristup fajlu), ispisuje poruku o grešci (sistemska poziv) i gasi se (sistemska poziv)
  - zatvara fajlove i gasi se

# Sistemski pozivi

- ◆ Prilikom programiranja na nekom programskom jeziku, programer vidi samo API (*application programming interface*) prema sistemskim pozivima – skup funkcija koje korisnički program poziva, prosleđujući parametre
- ◆ Najpopularniji API: Win32 API (Windows) i POSIX (UNIX, Linux, Mac OS)
- ◆ API funkcije realizovane su unutar standardnih biblioteka koje idu uz kompjajlere
- ◆ API funkcije izvršavaju sistemske pozive na nižem nivou detalja – *interfejs sistemskih poziva (system call interface)*; način poziva i način prenosa parametara zavisi od računara i OS-a

# Sistemski pozivi – Realizacija

- ◆ Na nivou interfejsa sistemskih poziva, pozivi imaju svoj broj i svoje parametre
- ◆ Jedan način realizacije sistemskih poziva jeste preko softverskih prekida (zamki, *trap*); broj prekida ili neki drugi parametar nose broj sistemskog poziva (usluge)
- ◆ Parametri se mogu prenositi preko:
  - registara procesora
  - steka
  - bloka podataka u memoriji na koji ukazuje pokazivač (adresa u registru)

# Sistemski pozivi – Realizacija

- ◆ Primer (izmišljen, principijelni i pojednostavljen):

- API funkcija:

```
FHANDLE fopen (char* filename);
```

- Implementacija API funkcije:

```
FHANDLE fopen (char* filename) {  
    asm {  
        mov ax,3Dh // broj sistemskog poziva FileOpen  
        mov bx,#filename[sp] // parametar  
        int 21h // sw prekid za sistemske pozive  
        // u ax je vraćeni rezultat  
    }  
}
```

# Sistemski pozivi – Tipovi

## ◆ Upravljanje procesima:

- *end, abort*
- *load, execute*
- *create process, terminate process*
- *get process attributes, set process attributes*
- *wait for time*
- *wait event, signal event*
- *allocate memory, free memory*

## ◆ Upravljanje fajlovima:

- *create file, delete file*
- *open file, close file*
- *read, write, reposition*
- *get file attributes, set file attributes*

# Sistemski pozivi – Tipovi

## ◆ Upravljanje uređajima:

- *request device, release device*
- *read, write, reposition*
- *get device attributes, set device attributes*
- *logically attach, detach devices*

## ◆ Održavanje informacija:

- *get time or date, set time or date*
- *get system data, set system data*
- *get process, file, or device attributes*
- *set process, file, or device attributes*

## ◆ Komunikacija:

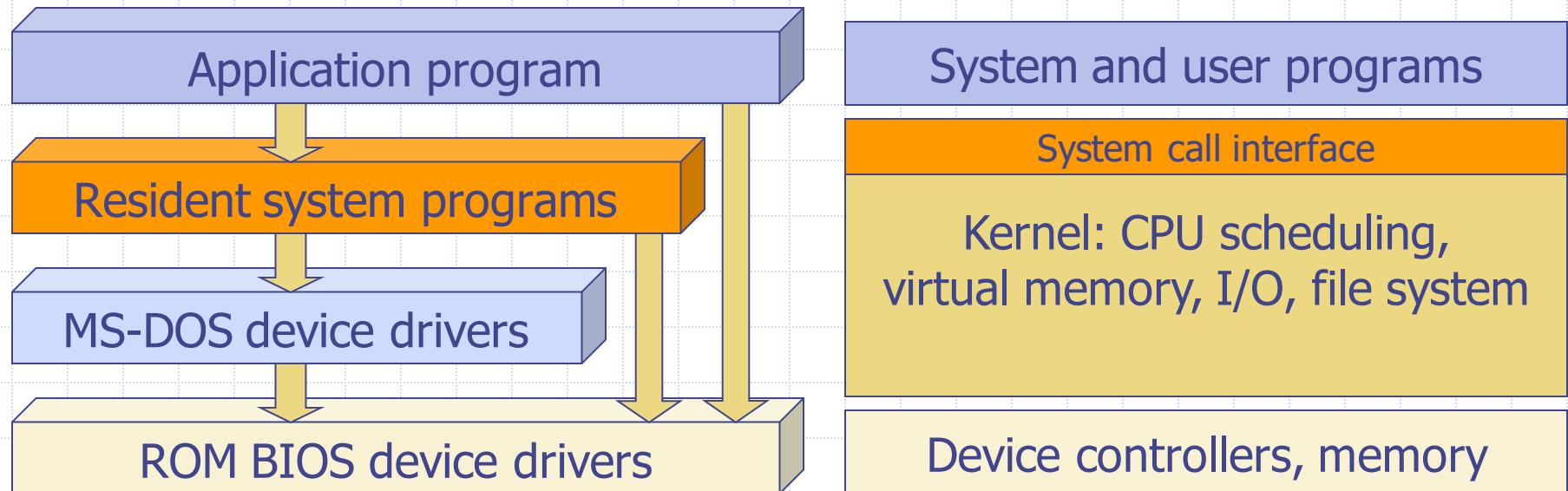
- *create, delete communication connection*
- *send, receive messages*
- *transfer status information*
- *attach, detach remote devices*

# Sistemski programi

- ◆ Sistemski programi su programi koji stižu u paketu sa OS; neki prosto "umotavaju" sistemske pozive, neki su složeniji
- ◆ Kategorije sistemskih programa:
  - rukovanje fajlovima: kreiraju, brišu, kopiraju, premeštaju, štampaju, listaju, prikazuju i uopšte manipulišu fajlovima
  - informativni: ispituju i prikazuju sistemske informacije, npr. vreme i datum, raspoloživu memoriju, prostor na disku, broj korisnika itd.
  - modifikacija fajlova: editori teksta i drugih standardnih formata
  - podrška za programiranje: prevodioci, linker, asembleri, debageri
  - učitavanje i izvršavanje programa
  - komunikacioni programi: uspostavljanje komunikacionih kanala, korišćenje Internet servisa (email, Web, ftp, telnet itd.)
- ◆ Mnogi OS dolaze u paketu i sa drugim, složenijim uslužnim aplikativnim programima (*utilities*)

# Struktura OS

- ◆ OS je veliki i složen softverski sistem, tako da se mora dobro strukturirati, organizovati i dekomponovati na celine sa jasno definisanim interfejsima i enkapsuliranim implementacijama
- ◆ Različiti OS primenjuju različite pristupe u svojoj strukturi
- ◆ Monolitna, jednostavna struktura: MS DOS, originalni UNIX



# Struktura OS

## ◆ Slojevita (*layered*) struktura:

- podeliti OS na manje delove organizovane u slojeve po nivoima apstrakcije
- svaki sloj sadrži strukture podataka i funkcionalnosti koje implementiraju interfejse koje dati sloj nudi višem sloju, uz oslanjanje na interfejse nižeg sloja
- osnovna prednost: jednostavnost konstrukcije i testiranja
- osnovni problemi:
  - ◆ kako jasno razdvojiti odgovornosti po slojevima
    - razrešiti međusobne zavisnosti i učiniti ih lineranim
  - ◆ veće režije zbog toka informacija po slojevima, uz usputne transformacije



# Struktura OS

## ◆ Mikrokernel (*microkernel*) struktura:

- ideja: izbaciti sve nepotrebne delove iz kernela i pretvoriti ih u sistemske programe i procese koji rade u korisničkom modu
- kernel učiniti što manjim, sa minimumom potrebnih funkcionalnosti koje se izvršavaju u kernel modu
- primer:
  - ◆ u kernelu ostaviti samo upravljanje procesima (promenu konteksta, raspoređivanje, sinhronizaciju i komunikaciju) i memorijom
  - ◆ pristup do fajlova ili uređaja implementirati procesima sa kojim drugi procesi komuniciraju da bi pristupali svom fajlu ili uređaju
- prednost: fleksibilnost (laka proširivost, promenljivost i prenosivost)
- nedostatak: lošije performanse
- primeri: Mach, QNX
- Windows NT je najpre imao slojevitu mikrokernel strukturu, ali sa lošijim performansama nego Windows 95; Win NT 4.0 je premestio mnoge funkcije u kernel

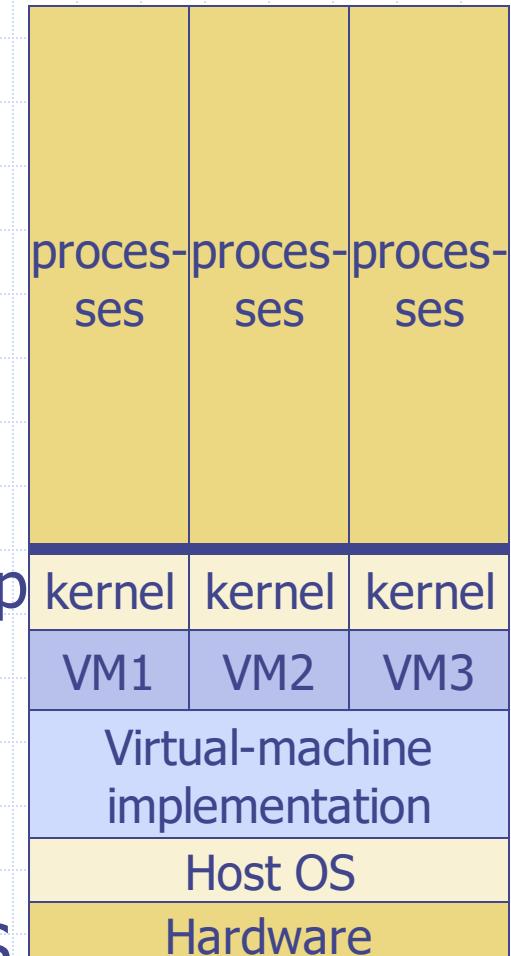
# Struktura OS

## ◆ Modularna struktura:

- korišćenje OO dekompozicije za modularizaciju kernela
- intenzivno korišćenje dinamičkog učitavanja i vezivanja (polimorfizma)
- moderne implementacije upotrebljavaju ovaj pristup:  
Solaris, Linux, Mac OS X
- enkapsulacija dobra kao kod slojevite strukture, ali fleksibilnija i lakša za projektovanje
- liči na mikrokernel strukturu, ali moduli mogu da komuniciraju direktno i efikasno, a ne međuprocesnom komunikacijom

# Virtuelne mašine

- ◆ Ideja: apstrahovati platformu (hardver i OS) jednog računara u nekoliko različitih izvršnih okruženja, stvarajući iluziju da svako posebno izvršno okruženje predstavlja zasebnu platformu
- ◆ Virtuelna mašina nudi procesu kompletno okruženje, sve što on uopšte može da koristi (instrukcije koje izvršava, podatke nad kojima radi, sistemske usluge i pristup hardveru), tako da proces radi potpuno zatvoreno unutar virtuelne mašine, ne pristupajući uopšte stvarnoj domaćinskoj platformi
- ◆ Primer: IBM VM – korisnici izvršavaju CMS jednokorisnički interaktivni OS



# Virtuelne mašine

- ◆ Iako koristan, koncept virtuelne mašine nije jednostavno implementirati – nije lako ostvariti potpunu iluziju – virtuelno okruženje identično ciljnom
- ◆ Na primer:
  - VM softver može da radi i u sistemskom i u korisničkom režimu, jer obuhvata ciljni OS
  - implementacija VM radi u korisničkom režimu sistema-domaćina
  - prema tome, implementacija VM mora da obezbedi virtuelni korisnički i sistemski režim, pri čemu oba rade u korisničkom režimu sistema-domaćina; svi sistemski pozivi koji menjaju režim u ciljnom sistemu moraju da menjaju virtuelni režim u VM
- ◆ Osnovna razlika je u vremenu, jer virtuelne I/O operacije uzimaju mnogo više vremena nego realne, a i mnoge instrukcije, kao i sistemski pozivi, se interpretiraju ili presreću

# Virtuelne mašine

## ◆ Pogodnosti koncepta virtuelne mašine:

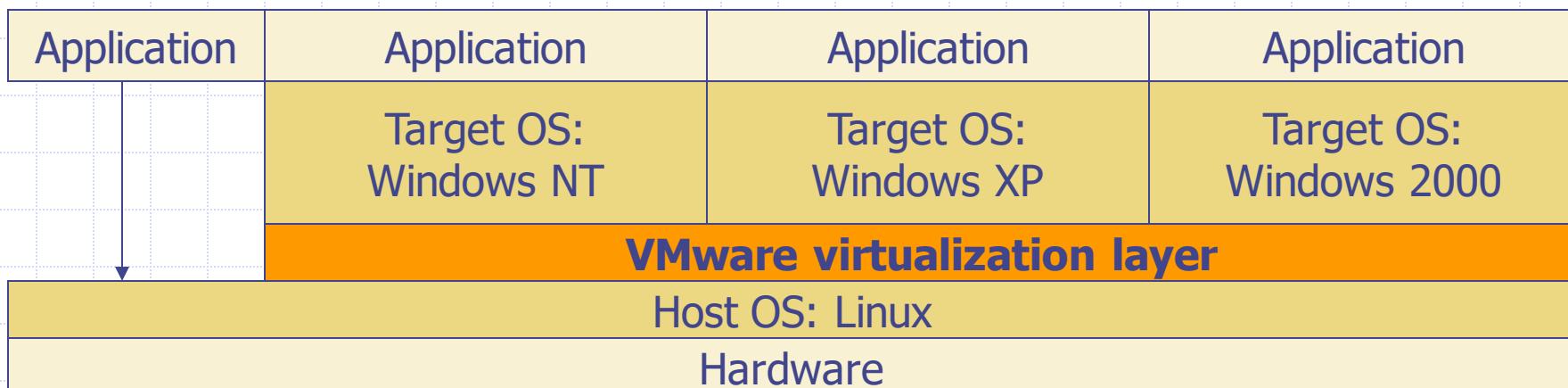
- višestruki, izolovani serverski sistemi na istom fizičkom računaru
- potpuna izolacija i zaštita sistema-domaćina od korisničkog programa, kao i jedne virtuelne mašine od drugih
- potpuna simulacija okruženja od strane softvera, potpuno apstrahovanje hardvera, uključujući i mreže
- odličan način za ispitivanje softvera na raznim ciljnim platformama
- odličan način za razvoj i ispitivanje samih OS; primer: realizacija projekta na OS1

## ◆ Koncept virtuelnih mašina je dugo bio zapostavljen, iako je odavno osmišljen. Danas ponovo dobija na popularnosti:

- VMware
- Java Virtual Machine
- Microsoft .Net Common Language Runtime (CLR)

# Virtuelne mašine - VMware

- ◆ VMware je popularna komercijalna aplikacija koja se izvršava na PC platformi i OS domaćinu (Linux ili Windows)
- ◆ Omogućava da OS domaćin izvršava jednu ili više virtuelnih mašina koje stvaraju okruženje ciljnog OS – bilo koji Windows ili Linux
- ◆ Odličan za jednostavno i jeftino testiranje razvijene aplikacije na raznim ciljnim platformama, na samo jednom računaru, umesto na više računara sa raznim platformama



# Virtuelne mašine - JVM

- ◆ Java (Sun Microsystems, 1995.) – popularni OO jezik čija je jedna od osnovnih ideja konstrukcije bila prenosivost
- ◆ Izvorni kod (.java fajlovi) se prevodi u *Java bytecode* (.class fajlovi) – platformski-nezavisan međukod za izmišljeni, apstraktni računar (stek-mašina)
- ◆ *Java Virtual Machine* (JVM) je specifikacija tog apstraktnog računara koji izvršava (interpretira) Java bytecode
- ◆ JVM se implementira na svakoj posebnoj platformi i na taj način čini Java bytecode potpuno prenosivim
- ◆ Implementacija JVM je softverski interpreter Java bytecode-a
- ◆ *Just-In-Time* (JIT) kompilacija: kada se neka metoda prvi put pozove, njen bytecode se prevede u mašinski jezik računara-domaćina i sledeći put direktno izvršava
- ◆ JVM se može implementirati i u hardveru – na čipu

# Virtuelne mašine - .Net CLR

- ◆ Microsoft .Net Framework je skup tehnologija, biblioteka, alata i izvršnih okruženja za razvoj i izvršavanje softvera
- ◆ Jedan od činilaca .Net Framework je *Common Language Runtime (CLR)* – Microsoftov odgovor na JVM
- ◆ Kao i JVM, CLR predstavlja virtuelnu mašinu koja interpretira međukod - MS *Intermediate Language*, MS-IL
- ◆ MS-IL (tzv. *assembly* fajlovi) može biti dobijen prevođenjem (pomoću .Net prevodilaca!) izvornog koda na različitim jezicima (C#, VB, C++, ...); svi prevedeni delovi su međusobno kompatibilni, bez obzira na izvorni jezik
- ◆ Izvorni kod se piše za .Net platformu, ne za ciljnu arhitekturu i OS, pa je prenosiv (!)
- ◆ CLR upotrebljava JIT

# Glava 8: Primer operativnog sistema – Linux

Istorijat

Principi dizajna

Moduli jezgra

Upravljanje procesima

Raspoređivanje

Upravljanje memorijom

Fajl sistemi

Ulaz i izlaz

Međuprocesna komunikacija

# Istorijat

- ◆ Linux spolja izgleda skoro isto i kompatibilan je sa drugim UNIX sistemima
- ◆ Linus Torvalds, finski student, 1991. počinje razvoj malog, novog *open-source* kernela za 80386 i objavljuje ga na Internetu; naziva ga Linux
- ◆ Linux je proizvod kolaboracije mnogih programera širom sveta i pravi je začetnik *open-source* filozofije
- ◆ U svom ranom stadijumu, razvoj Linuxa se uglavnom fokusirao na *kernel* – jezgro sa privilegovanim izvršavanjem koje direktno upravlja svim sistemskim resursima i hardverom
- ◆ Linux kernel: potpuno originalan deo softvera razvijen potpuno od početka od strane Linux zajednice (*community*)
- ◆ Linux sistem: kernel plus skup mnogih drugih komponenata, nekih razvijenih ispočetka, nekih pozajmljenih od drugih sistema i grupa
- ◆ Linux distribucija: sve standardne komponente Linux sistema, plus skup administrativnih alata koje olakšavaju instalaciju i održavanje i drugih sistemskih, kao i aplikativnih programa

# Istorijat – Linux kernel

## ◆ V0.01, 14. maj 1991:

- izvršava se samo na Intel 80386 i kompatibilnim procesorima
- UNIX procesi sa zaštićenim adresnim prostorima
- nema mrežnog softvera
- veoma ograničen skup drajvera uređaja
- podsistem za virtuelnu memoriju jednostavan, podržava straničnu organizaciju, deljenje stranica i *copy-on-write*, ali ne podržava memorijski preslikane fajlove
- Minix fajl sistem

## ◆ V1.0, 14. mart 1994:

- standardni UNIX TCP/IP, BSD kompatibilan *socket* interfejs
- podrška Ethernetu i modemima, kao i širem spektru raznih I/O uređaja (flopi, CD-ROM, miševi, zvučne kartice, internac. tastature)
- proširen fajl sistem i podrška lepezi SCSI kontrolera
- zamena stranica u *swap* fajlove i memorijski preslikani fajlovi

# Istorijat – Linux kernel

## ◆ Oznake verzija kernela:

- sa neparnim brojem podverzije (1.1, 1.3, 2.1,...) – *razvojne (development kernels)*, sadrže nove i možda netestirane stvari
- sa parnim brojem podverzije (1.2, 1.4, 2.2,...) – stabilne *proizvodne (production kernels)* verzije sa ispravkama, bez novih stvari

## ◆ V1.2, mart 1995:

- podrška širem spektru hardvera: PCI bus, podrška 8086 virtuelnog moda na 80386 (emulacija DOSa)
- poslednji kernel koji je bio samo za PC – započinje podrška za Sun SPARC, DEC Alpha i MIPS procesore
- kompletirana implementacija IP i *firewalls*

# Istorijat – Linux kernel

## ◆ V2.0, jun 1996:

- puna podrška za više arhitektura (sada i Motorola 68000 serija), uključujući i 64-bitni Alpha
- podrška za multiprocesorske arhitekture
- unificirano keširanje fajlova nezavisno od keširanja blokovskih uređaja, poboljšane performanse fajl sistema i VM
- poboljšanje performansi TCP/IP, veći broj novih protokola (ISDN, AX.25)
- Interne kernel niti, automatsko učitavanje modula na zahtev, dinamička konfiguracija kernela

## ◆ V2.2, januar 1999:

- podrška za UltraSPARC, poboljšan mrežni softver, proširena podrška I/O uređajima

## ◆ V2.6, kraj 2003:

- efikasno raspoređivanje procesa kompleksnosti  $\mathcal{O}(1)$ , potpuno *preemptive* u kernel modu i druga proširenja

# Istorijat – Linux sistem i distribucije

- ◆ Linux sistem sadrži mnogo komponenata koje su razvijene za UNIX van Linux pokreta : Berkley BSD, MIT X Windows, Free Software Foundation GNU (npr. *GNU C compiler*, gcc)
- ◆ Ovaj uticaj je išao u oba smera: Linux je doprineo poboljšanjima ili proširenjima ovih drugih projekata
- ◆ Teorijski, svako može da instalira Linux skidajući poslednje verzije izvornog koda potrebnih komponenata sa ftp sajtova i prevodeći ih; nekada je to tako i bilo, ali je nepraktično
- ◆ Distribucije obezbeđuju gotove pakete komponenata spremnih za instalaciju i održavanje instalacije. Neke važnije:
  - SLS: najranije doba, prva distribucija
  - Slackware, Red Hat, SuSe, Debian, Caldera, Fedora, Ubuntu, Craftworks, Unifix...

# Istorijat – Licenciranje

- ◆ Linux nije *public domain*, jer to znači da su se autori odrekli svog *copyright*, *copyright* na Linux kod drže njegovi autori
- ◆ Linux kernel se distribuira pod GNU *General Public License* (GPL) koju definiše Free Software Foundation; osnovna pravila:
  - softver je besplatan
  - svako može da ga kopira, koristi, menja ili distribuira bez restrikcija
  - niko ko koristi Linux ili ga je koristio da bi napravio svoj proizvod, ne sme taj proizvod da učini svojim vlasništvom
  - softver se ne može distribuirati isključivo u binarnoj formi; svakome kome se da binarna forma (besplatno ili komercijalno), mora biti dostupan i izvorni kod (besplatno ili za razumnu cenu koja ne uključuje cenu proizvoda, već samo isporuke/usluge)

# Principi dizajna

- ◆ U svom sveukupnom dizajnu, Linux liči na bilo koju drugu tradicionalnu, ne-mikrokernelsku implementaciju UNIX-a
- ◆ Linux je multikorisnički, multitasking sistem sa skupom UNIX-kompatibilnih alatki, fajl- i mrežnim sistemom koji imaju istu semantiku kao i UNIX sistemi
- ◆ Zbog toga što je u početku nastao samo na PC platformi, kao rezultat rada entuzijasta bez finansijske podrške, Linux je dizajniran tako da izvlači maksimum iz ograničenih resursa. Zato Linux može uspešno da radi i na multiprocesorskom sistemu sa stotinama MB RAM-a, i na samo 4MB RAM-a
- ◆ U novije vreme, mnogo više napora se ulaže u standardizaciju kao cilj, pored efikasnosti i brzine (kompatibilnost između raznih implementacija UNIX-a nije 100%). Linux je dizajniran da bude kompatibilan sa POSIX i Pthread standardom
- ◆ Linux API podržavaju semantiku SVR4 UNIX API, ne BSD semantiku

# Principi dizajna

- ◆ Linux sistem se sastoji od tri glavne celine programskega koda:
  - kernel obezbeđuje sve osnovne apstrakcije OS-a
  - sistemske biblioteke: definišu standardni skup funkcija kojima korisnički program interaguje sa kernelom
  - sistemske uslužne programe (*utilities*): sistemske programe koji obavljaju određene zadatke; neki se mogu pozivati samo jednom da bi nešto inicijalizovali ili konfigurisali, drugi (*demoni*, *daemons*) se permanentno izvršavaju, obrađujući pristigne zahteve (npr. dolazeći saobraćaj sa mreže, *logon* korisnika sa terminala, upis u *log* fajlove)
- ◆ Ceo kernel se izvršava u procesorskom privilegovanim režimu (*kernel mode*) sa punim pristupom do svih fizičkih resursa
- ◆ Sav sistemske kod koji ne treba da se izvršava u kernel modu smešten je u sistemske biblioteke ili uslužne programe

# Principi dizajna

- ◆ Linux kernel je monolitan, u cilju boljih performansi: sav kod i podaci (za sve funkcije, uključujući i drajvere, fajl sistem, mrežnu komunikaciju) su u istom adresnom prostoru, nema promene konteksta kada proces vrši sistemski poziv
- ◆ Kernel može dinamički da učita (i izbaci) deo koda/modul po potrebi tokom izvršavanja
- ◆ Linux kernel implementira sve funkcije jednog OS-a, ali ono što on nudi nije ni približno nalik potpunom UNIX-u – mnoge funkcije fale ili su drugačije. Umesto toga, sistemske biblioteke nude API koji odgovara UNIX-u; one vrše sistemske pozive kernela
- ◆ Sistemske biblioteke nude i kompleksnije sistemske usluge, kao i usluge koje nisu deo kernela (npr. sortiranje, stringovi)

# Moduli jezgra

- ◆ Kernel može da učitava proizvoljni modul dinamički, po potrebi
- ◆ Svi moduli kernela izvršavaju se u kernel modu i imaju neograničen pristup do svog hardveda
- ◆ Kernel modul npr. može da implementira drajver uređaja, fajl sistem, ili mrežni protokol
- ◆ Zbog toga je kernel proširiv: svako može da doda novi modul (npr. drajver), prevede ga i dinamički poveže (bez prevodenja, povezivanja i restartovanja celog kernela)
- ◆ Posledica: tako dodati moduli ne moraju biti po GPL licenci jer ne zahtevaju da se ceo kernel isporučuje sa dodatim komponentama, već nezavisno

# Moduli jezgra – Rukovanje modulima

- ◆ Dinamičko učitavanje modula ne zahteva samo učitavanje binarne forme u memoriju kernela, već i dinamičko povezivanje referenci modula na simbole iz kernela
- ◆ Zato kernel čuva internu tabelu simbola – ne svih, već samo onih koji su eksplicitno “izveženi” iz kernela da bi ih moduli koristili (interfejs kernela prema modulima); ovo zahteva posebnu intervenciju programera kernela
- ◆ Da bi modul “uvezao” simbol iz kernela, nije potreban nikakav poseban postupak – koristi se standardno spoljno povezivanje jezika C (svi simboli koji se koriste, a nisu definisani)
- ◆ Kada se modul učita, poseban deo kernela vrši povezivanje, kao klasičan linker, korišćenjem interne tabele simbola
- ◆ Učitani modul može da dopuni ovu tabelu simbola svojim simbolima koje izvozi
- ◆ Kad god korisnički proces zahteva pristup do modula koji nije učitan, pokreće se postupak dinamičkog učitavanja i povezivanja
- ◆ Poseban proces povremeno proziva kernel da vidi da li je modul u aktivnoj upotrebi i izbacuje ga ako nije

# Moduli jezgra – Registracija drajvera

- ◆ Kernel održava dinamičke tabele svih raspoloživih drajvera i nudi rutine za dodavanje ili izbacivanje drajvera iz tabele
- ◆ Kernel poziva inicijalizacionu i završnu rutinu modula po učitavanju i izbacivanju. One su odgovorne za registraciju i deregistraciju funkcionalnosti modula
- ◆ Jedan modul može da registruje više drajvera raznih tipova
- ◆ Tipovi drajvera:
  - drajveri uređaja: znakovnih, blokovskih i mrežnih
  - fajl sistem: bilo šta što implementira Linux virtualni fajl sistem (fajlovi na disku u određenom formatu, mrežni fajl sistem itd.)
  - mrežni protokol (uključujući i pravila propuštanja paketa – *firewall*)
  - binarni format: specifičuje način prepoznavanja i učitavanja novog tipa izvršnog fajla

# Moduli jezgra – Rešavanje konflikta

- ◆ Linux može da se izvršava na najrazličitijim hardverskim konfiguracijama (PC). Kako izbeći da dinamički učitani i nezavisni drajveri ne pristupaju konfliktno istim uređajima?
- ◆ Kernel održava listu hardverskih resursa (I/O portovi, linije zahteva za prekid i DMA kanali) i njihove alokacije od strane drajvera
- ◆ Pre nego što pristupa određenom HW resursu, svaki drajver je dužan da ga najpre *rezerviše* pozivom određene funkcije kernela
- ◆ Ako kernel zaključi da je resurs već zauzet od strane drugog drajvera ili prosto taj resurs nije instaliran, odbija zahtev; na modulu je da taj problem dalje razreši (može da obustavi svoju inicijalizaciju ili da proba da rezerviše alternativne resurse)

# Upravljanje procesima

- ◆ UNIX stil: kreiranje novog procesa (nad istim kodom) i pokretanje novog programa (unutar istog procesa) su dve nezavisne operacije – `fork()` i `exec()`
- ◆ *Identitet (identity)* procesa:
  - *Process ID (PID)*: jedinstveni identifikator procesa koji se koristi u sistemskim pozivima kao argument
  - *Credentials*: ID korisnika i ID grupe korisnika koje određuju prava procesa da pristupa sistemskim resursima i fajlovima
  - *Personality*: identifikator koji može neznatno da izmeni semantiku sistemskih poziva; prvenstveno se koristi kod emulacionih biblioteka da bi se zahtevalo da sistemski poziv bude kompatibilan sa određenim varijantama UNIXa
- ◆ *Okruženje (environment)* procesa – nasleđuju se od roditelja:
  - vektor argumenata iz komandne linije
  - vektor varijabli iz okruženja: lista parova "name=value"; definiše ga `exec()`

# Upravljanje procesima

## ◆ Kontekst (*context*) procesa:

- *scheduling context*: informacije potrebne da se proces suspenduje i ponovo restartuje (kontekst izvršavanja)
  - ◆ svi registri procesora; *floating-point* registri se čuvaju i restauriraju samo po potrebi
  - ◆ *kernel stek procesa (process' kernel stack)*: deo memorije kernela rezervisan za kod koji se izvršava u kernel modu; svi sistemski pozivi i prekidne rutine koriste ovaj stek
- *accounting*: informacije o resursima koje proces koristi i korišćenju resursa tokom dosadašnjeg života procesa
- *file table*: niz pokazivača na deskriptore otvorenih fajlova unutar kernela; kada identificiše fajlove u sistemskim pozivima, proces koristi indeks ulaza u ovoj tabeli

# Upravljanje procesima

## ◆ Kontekst (*context*) procesa (nastavak):

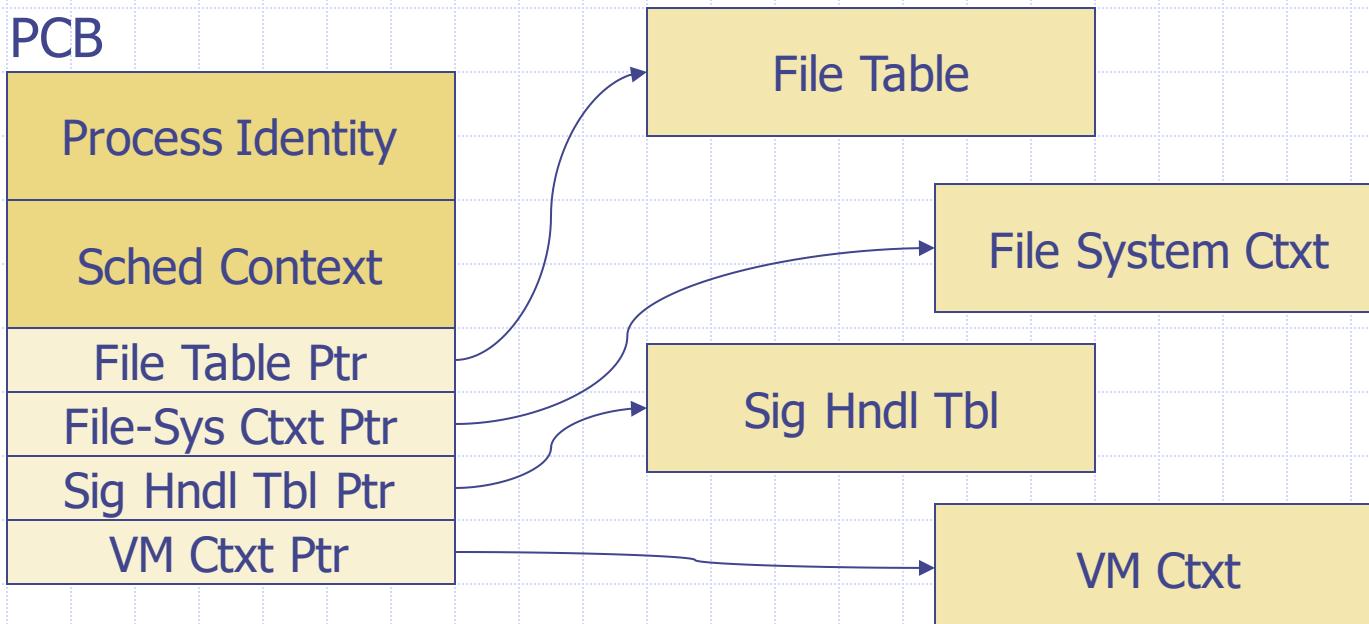
- *file-system context*: podaci koji se odnose na zahteve za otvaranjem novih fajlova (tekući i koreni direktorijum za traženje fajlova)
- *signal-handler table*: UNIX može da prosleđuje asinhronne signale procesu; ova tabela definiše rutine u adresnom prostoru procesa koje se pozivaju za određene signale
- *virtual memory context*: podaci o adresnom prostoru procesa

# Upravljanje procesima

- ◆ Kreiranje niti (*thread*): sistemski poziv `clone()`
- ◆ Linux zapravo ne razlikuje procese i niti, već ih generalizuje u pojam *zadatka* (*task*) – tok kontrole unutar programa
- ◆ Ideja je da se pri kreiranju zadatka-deteta samo definiše šta će tačno on deliti sa roditeljem. Moguće je deliti:
  - Deliti:
    - Podatke o fajl sistemu
    - Adresni prostor
    - Rutine za signale
    - Otvorene fajlove
  - Fleg (argument kreiranja zadatka-deteta):
    - `CLONE_FS`
    - `CLONE_VM`
    - `CLONE_SIGHAND`
    - `CLONE_FILES`
- ◆ `clone()` sa postavljenim svim ovim flegovima – kreira nit
- ◆ `clone()` sa obrisanim svim ovim flegovima – kreira proces (kao `fork()`)

# Upravljanje procesima

- ◆ Ovo je implementirano tako što PCB zadatka zapravo čuva pokazivače na strukture koje čuvaju kontekst fajl-sistema, tabelu deksriptora stranica, tabelu rutina za obradu signala i kontekst virtuelne memorije, pa više zadataka može da deli ove strukture (potkontekste)
- ◆ Argumenti poziva `clone()` zapravo govore koje od ovih podkonteksta treba kopirati u zadatak-dete (duboko kopiranje), a koje deliti (plitko kopiranje, samo kopirati pokazivače na potkontekste unutar PCB-a)



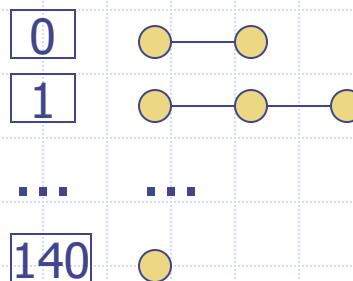
# Raspoređivanje

- ◆ Linux poseduje dva algoritma raspoređivanja:
  - *time-sharing* za pravično raspoređivanje sa preuzimanjem
  - *real-time* raspoređivanje po prioritetima
- ◆ Pre V2.5 kernela, Linux je koristio varijantu tradicionalnog UNIX *time-sharing* raspoređivanja; ovo nije imalo adekvatnu podršku za multiprocesorske sisteme i nije bilo skalabilno za više procesa
- ◆ Od V2.5 do V2.6.23, algoritam raspoređivanja izvršava se u konstantnom vremenu, nezavisnom od broja procesa ( $O(1)$ )
- ◆ Od V2.6.23 koristi se CFS kao osnovni, ali kernel može imati različite raspoređivače za različite kategorije procesa
- ◆ Raspoređivanje  $O(1)$  je sa preuzimanjem (*preemptive*), bazirano na prioritetima (*priority-based*)
- ◆ Dva opsega prioriteta (niži broj-viši prioritet):
  - *real-time*: 0-99
  - *nice*: 100-140

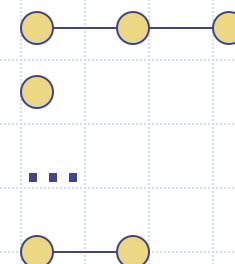
# Raspoređivanje

- ◆ Procesima višeg prioriteta dodeljuje se duži vremenski interval i obratno (suprotno od ustaljenog principa) – od 10 ms (prioritet 140) do 200 ms (prioritet 0)
- ◆ Proces se smatra kandidatom za izvršavanje sve dok mu ne istekne dodeljeni interval; kada mu istekne interval, on se smešta u red *expired* procesa i nije kandidat za izvršavanje sve dok svim ostalim procesima ne istekne interval
- ◆ Za svaki prioritet vode se dva reda: *active* i *expired*

Priority Active task list



Expired task list



# Raspoređivanje

- ◆ Bira se proces najvišeg prioriteta iz *active* liste
- ◆ Kada se cela *active* struktura isprazni, strukture *active* i *expired* zamenjuju uloge
- ◆ Kod multiprocesora, svaki procesor ima svoju ovakvu strukturu i vrši svoje raspoređivanje po istom principu
- ◆ Procesu se dodeljuje dinamički prioritet baziran na *nice* vrednosti +/-5 u zavisnosti od interaktivnosti procesa. Interaktivnost je srazmerna vremenu koju je proces proveo čekajući na završetak I/O operacija. Veća interaktivnost – viši prioritet (bliže -5), manja interaktivnost – niži prioritet (bliže +5)
- ◆ Dinamički prioritet se ažurira kada procesu istekne interval; kada *active* i *expired* zamene uloge, svi procesi imaju ažuriran prioritet i novi interval vremena u skladu sa njim

# Raspoređivanje

- ◆ Kernel svoje sopstvene poslove raspoređuje bitno drugačije nego korisničke procese
- ◆ Kada se prelazi u kernel mod?
  - kada korisnički proces izvrši sistemski poziv eksplicitno ili implicitno (npr. *page fault*)
  - kada neki I/O uređaj generiše hardverski prekid i pokrene se prekidna rutina
- ◆ Problem: obezbediti međusobno isključenje kritičnih sekcija kernela tako da asinhroni prekid ne prekine izvršavanje kritične sekcije samog kernela i uđe u svoju kritičnu sekciju
- ◆ Pre V2.6, Linux kernel je ovo rešavao tako što nije bio *preemptive* – nije dozvoljavao prekide u kernel modu
- ◆ Sada je Linux kernel *preemptive*, pa se procesor može preuzeti i kada izvršava kod u kernel modu

# Raspoređivanje

- ◆ Za zaključavanje kratkih kritičnih sekcija unutar kernela, Linux koristi:
  - zabranu/dozvolu preuzimanja (prekida) za jednoprocesorski sistem
  - *spinlock* za višeprocesorski sistem
- ◆ Ako izvršavanje drži neki *spinlock*, onda nije dozvoljeno ni preuzimanje. Rešeno brojanjem ključeva (preempt\_count)
- ◆ Za duže kritične sekcije unutar kernela, Linux koristi semafore

# Upravljanje memorijom – Alokacija

- ◆ Dve komponente upravljanja memorijom:
  - alokacija fizičke memorije: stranica, grupa stranica i manjih blokova
  - upravljanje straničnom organizacijom virtuelne memorije
- ◆ Tri zone fizičke memorije:
  - **ZONE DMA**: memorija za prenos preko DMA, za one arhitekture gde je to ograničeno (npr. na 80x86, neki ISA uređaji mogu da pristupaju samo najnižim 16MB fizičke memorije preko DMA); ako ograničenje ne postoji, ova zona ne postoji (koristi se sledeća)
  - **ZONE NORMAL**: zona koju koristi procesor za većinu memorijskih potreba
  - **ZONE HIGH**: fizička memorija koja se ne preslikava u adresni prostor kernela (npr. 32-bitne Intel aritekture, od 4GB adresnog prostora, kernel prostor je prvih 896MB)
- ◆ Kernel održava listu slobodnih stranica i opslužuje zahteve za alokacijom memorije iz odgovarajuće zone

# Upravljanje memorijom – Alokacija

- ◆ Svaka zona ima svoj alokator stranica (*page allocator*) koji može da alocira jednu ili više susednih stranica na zahtev
- ◆ Sistem parnjaka (*buddy system*) za alokaciju više susednih stranica:
  - kad god se oslobodi jedan blok i njegov parnjak (*buddy*) iste veličine je slobodan, ova dva se spajaju u dvostruko veći blok slobodne memorije (*buddy heap*), itd. rekursivno naviše dok može
  - ako se zahtev ne može zadovoljiti odgovarajućim malim slobodnim blokom, onda se prvi sledeći veći slobodni parnjak deli na dva dela, i dalje rekursivno, sve dok se ne dobije blok zahtevane veličine koji se alocira
  - najmanji blok je veličine stranice
  - za svaku dozvoljenu veličinu bloka vodi se zasebna lista slobodnih blokova

# Upravljanje memorijom – Alokacija

- ◆ Alokacija u kernelu vrši se ili statički (drajveri koji alociraju svoj prostor prilikom podizanja sistema) ili dinamički
- ◆ Kernel ne mora da koristi osnovni alokator stranica, već postoje posebni memorijski podsistemi za različite potrebe koji se oslanjaju na osnovni alokator stranica:
  - podsistem virtuelne memorije
  - `kmalloc()` za alokaciju blokova promenljive veličine unutar kernela; alocira stranice na zahtev, ali ih onda deli na manje delove po potrebi; kada se zahteva blok određene veličine, traži se takav deo u listama slobodnih delova već alociranih stranica; ako takvog nema, alocira se nova stranica ili više stranica
  - *slab* alokator za kernel strukture
  - *page cache* za keširanje stranica koje pripadaju fajlovima, blokovskim I/O uređajima ili mrežnim baferima

# Upravljanje memorijom – VM

- ◆ Logički pogled na virtuelni adresni prostor procesa:
  - skup kontinualnih, nepreklapajućih regiona, poravnatih na stranice, koji čine podskup virtuelnog adresnog prostora koji proces koristi
  - svaki od ovih regiona opisan je strukturom `vm_area_struct`: prava pristupa (*read*, *write*, *execute*) i fajl preslikan u region
  - ove strukture su za svaki proces povezane u balansirano binarno stablo da bi obezbedile brzu pretragu regiona koji odgovara nekoj virtuelnoj adresi
- ◆ Fizički pogled na virtuelni adresni prostor:
  - informacije smeštene u tabele preslikavanja adresa koje koristi hardver (PMT: lokacija u fizičkoj memoriji, da li je u memoriji)
  - ovom tabelom manipulišu prekidne rutine kernela koje se pozivaju na *page fault*
  - svaka `vm_area_struct` sadrži pokazivač na tabelu funkcija koje implementiraju rukovanje stranicama za taj region (polimorfizam); na njih se preusmeravaju pozivi iz prekidnih rutina na *page fault*

# Upravljanje memorijom – VM

## ◆ Regioni virtuelne memorije se karakterišu:

- šta je u "pozadini" (*backing store*) – odakle je stranica inicijalno dobijena:
  - ◆ ništa - *demand-zero memory*: stranica je inicijalno popunjena nulama
  - ◆ fajl: prozor na deo memorijski preslikanog fajla; ulaz u PMT je usmeren na istu fizičku stranicu iz keša stranica koja odgovara tom delu fajla
- reakcija na upis:
  - ◆ privatni region: *copy-on-write* semantika
  - ◆ deljeni region: upisi se manifestuju u svim procesima koji dele region

## ◆ Postupak kernela pri kreiranju procesa (`fork()`):

- kopira deskriptore `vm_area_struct` i tabele preslikavanja roditelja u prostor deteta; tabele ukazuju na iste stranice (inkrementira se brojač referenci); roditelj i dete dele sve stranice
- privatni regioni roditelja se u detetu označavaju kao *read-only* i *copy-on-write*, tako da se dele sve dok proces-dete ne želi da ih menja

# Upravljanje memorijom – VM

- ◆ Algoritam zamene stranica - modifikovani *second-chance* (*clock*) algoritam:
  - svakoj stranici se pridružuje *starost* (*age*) - mera koliko je stranica bila aktivna u poslednje vreme (zapravo odslikava "mladost")
  - smanjuje se ka 0 pri svakom obilasku "kazaljke"
  - umesto binarnih indikatora, koriste se brojači (*age*), pa se algoritam ponaša približnije LRU
- ◆ Mehanizam zamene stranica:
  - zamena je moguća i u posebno namenjene particije i u fajlove (znatno sporije)
  - alokacija blokova na particiji za zamenu pomoću bit-vektora zauzetosti koji se uvek drži u memoriji
  - *next-fit* algoritam alokacije kontinualne sekvence blokova na disku za zamenu, radi povećanja performansi

# Upravljanje memorijom – VM

- ◆ Unutar virtuelnog adresnog prostora svakog procesa, Linux odvaja poseban deo fiksne veličine (zavisne od arhitekture) za svoju internu upotrebu; ove stranice su zaštićene od pristupa u korisničkom režimu. Dva regiona:
  - preslikan na tačno određeno fizičko mesto sa glavnim delom kernela
  - preslikan proizvoljno i dostupan procesu preko sistemskih poziva `vmalloc()` (alocira deo memorije željene veličine) i `vremap()` (preslikava deo virtuelnog prostora u prostor koga koristi drajver ili memorijski preslikan I/O)
- ◆ Prilikom pokretanja programa (`exec()`) unutar istog procesa, kreira se novi izvršni kontekst koji prepisuje stari. *Loader* obezbeđuje novo preslikavanje programa u virtuelni prostor

# Upravljanje memorijom – Učitavanje

- ◆ Ne postoji jedinstvena rutina za učitavanje, već tabela raspoloživih rutina, jer su podržani različiti formati binarnih izvršnih fajlova:
  - a.out: stari UNIX jednostavni format
  - ELF: noviji format koji dozvoljava proširenje novim sekcijama (npr. sa debug informacijama)
- ◆ *Loader* ne učitava program u memoriju, već preslikava stranice programskog fajla u virtuelni prostor procesa. Stranice se dalje učitavaju na zahtev tokom izvršavanja
- ◆ Linux podržava biblioteke sa dinamičkim povezivanjem (DLL)

# Fajl sistemi – VFS

- ◆ U UNIXu i Linuxu fajl ne mora da bude samo na disku (lokalnom ili udaljenom), već je to bilo koji apstraktni objekat koji ima operacije davanja i prihvatanja toka podataka – *virtuelni fajl sistem (virtual file system, VFS)*
- ◆ VFS podržava objektni pristup. Vrste objekata (klase):
  - *inode object*: predstavlja fajl
  - *file object*: predstavlja otvoreni fajl
  - *superblock object*: predstavlja ceo fajl sistem
  - *dentry object*: predstavlja jedan ulaz u direktorijumu
- ◆ Za svaku od ovih vrsta objekata (klasa), definiše se skup operacija. Svaki objekat predstavljen je struktrom podataka koja sadrži i pokazivač na tabelu funkcija (*virtual table pointer, VTP*) – polimorfizam i dinamičko vezivanje kao u C++u

# Fajl sistemi – VFS

- ◆ *Inode* objekat sadrži informacije o poziciji fajla na disku. Globalan je (deljen između procesa). Da bi mu pristupio, proces mora najpre da dobije *file* objekat koji na njega ukazuje
- ◆ *File* objekat sadrži informacije o pravima pristupa procesa do otvorenog fajla, kao i pokazivač trenutne lokacije za sekvensijalni pristup. Lokalan je za svaki proces, ali se kešira prilikom oslobođanja, ako je potreban drugom procesu u bliskoj budućnosti
- ◆ Operacije nad fajlovima koji predstavljaju direktorijume izvršavaju se specifično i definisane su za *inode*, a ne za *file* objekte, jer ne zahtevaju prethodno otvaranje fajla
- ◆ Za svaki montirani fajl sistem vodi se *superblock* objekat. On obezbeđuje pristup do *inode* objekata

# Fajl sistemi – VFS

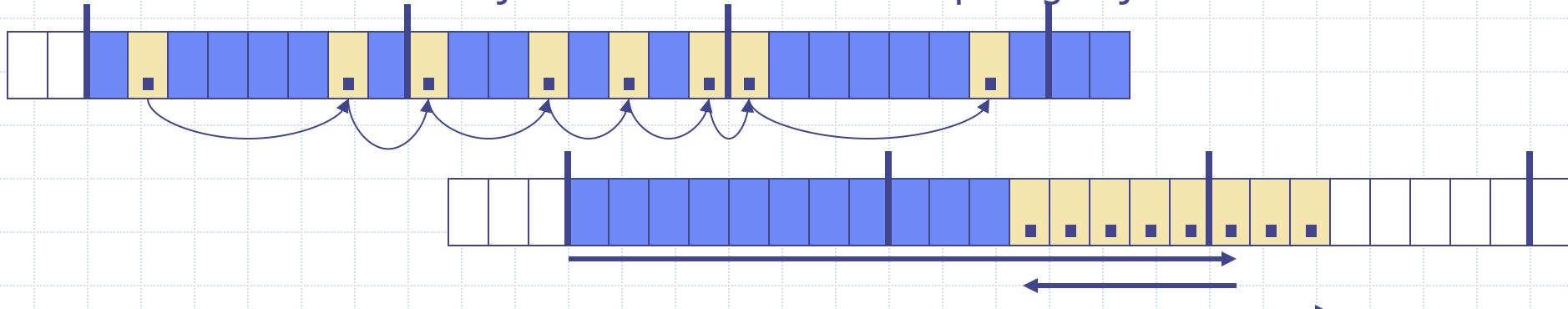
- ◆ U cilju bržeg preslikavanja simboličkih imena u *inode* objekte, vode se *dentry* objekti (keš preslikavanja čvorova u hijerarhiji direktorijuma): za svaki čvor u strukturi direktorijuma kome se pristupalo po jedan. On preslikava simboličko ime čvora u hijerarhiji direktorijuma u *inode*
- ◆ Najstariji Linux fajl sistem Minix imao je mnogo ograničenja (14 znakova za ime fajla, 64MB za fajl)
- ◆ Standardni Linux fajl sistem: *ext2fs* (*second extended file system*). Ima mnogo sličnosti sa BSD Fast File System:
  - indeksni pristup alokaciji, sa jednim do tri nivoa indirekcije
  - direktorijumi se smeštaju kao obični fajlovi, samo im se sadržaj drugačije interpretira: svaki blok u fajlu je ulančana lista čiji su elementi promenljive veličine i sadrže imena fajlova i broj *inode* koga taj element predstavlja

# Fajl sistemi – ext2fs

- ◆ Podrazumevana veličina bloka je 1KB, a podržani su i blokovi veličine 2KB i 4KB
- ◆ U cilju povećanja performansi (pristupati što više susednim blokovima na disku), prostor za smeštanje fajlova deli se na *grupe blokova (block groups)* (u FFS grupe cilindara, ali je to prevaziđeno jer noviji diskovi nemaju cilindre iste veličine)
- ◆ Kada alocira fajl, ext2fs najpre određuje grupu blokova za fajl:
  - za smeštanje blokova sa podacima fajla, traži se najpre mesto unutar grupe blokova gde je smešten i *inode* za taj fajl
  - za smeštanje *inode* strukture za fajlove koji nisu direktorijumi, traži se najpre u grupi blokova u kojoj je smešten roditeljski direktorijum
  - *inode* za direktorijume se rasipaju po raznim grupama blokova

# Fajl sistemi – ext2fs

- ◆ Unutar grupe blokova, blokovi se alociraju na sledeći način:
  - vodi se bit-vektor slobodnih blokova za svaku grupu blokova
  - forsira se kontinualna alokacija sa izbegavanjem fragmentacije
  - kada se alocira prvi blok za fajl, traži se najpre blok na početku grupe
  - kada se fajl proširuje, blok se traži najpre u grupi u kojoj je najskorije alociran blok tog fajla, u dva prolaza:
    - ◆ prvo se traži ceo bajt slobodnih blokova, kako bi se prealocirao niz fizički susednih 8 blokova za taj fajl; ako se ovako nađe ceo bajt, traže se unazad slobodni biti sve do poslednje popunjeno bloka, kako bi se popunila rupa, a zatim se u novom bajtu alocira ostatak do 8 blokova
    - ◆ ako to ne uspe, traži se bilo koji slobodan blok u toj grupi; zatim se odmah alociraju svi slobodni blokovi do punog bajta



# Fajl sistemi - *Journaling*

## ◆ Jedna popularna funkcionalnost mnogih Linux fajl sistema - *beleženje (journaling)*:

- modifikacije (akcije) nad fajl sistemom koje se vrše u okviru jedne nedeljive celine – *transakcije (transaction)* ne vrše se sinhrono, već se najpre upisuju u "zapisnik" (*journal*)
- kada se transakcija (npr. `write()`) završi (*commit*) i cela upiše u zapisnik, kontrola se vraća korisničkom procesu
- u međuvremenu, zapisane akcije se asinhrono redom obavljaju nad fajl sistemom; zapisnik je zapravo kružni bafer koji ima pokazivač (kurzor) koje zapisane akcije su zaista izvršene u fajl sistemu
- kada su sve akcije jedne transakcije završene, one se brišu iz zapisnika
- zapisnik se čuva kao posebna sekcija fajl sistema
- ako sistem padne, cele transakcije koje su potvrđene (*committed*) a nisu izvedene u celini u fajl sistemu se naknadno izvršavaju; delimično izvršene transakcije se poništavaju ("razmotavaju", *undo, rollback*), kako bi sistem ostao u konzistentnom stanju

# Fajl sistemi – *Process File System*

## ◆ UNIX/Linux *process file system* (`/proc`):

- virtuelni fajl sistem koji se ne odnosi na perzistentne podatke, već koristi VFS da bi vratio informaciju na svaki zahtev za čitanjem
- svaki “poddirektorijum” unutar `/proc` odnosi se na jedan tekući proces sa imenom koje predstavlja ASCII reprezentaciju PID-a
- “fajlovi” unutar ovih “poddirektorijuma” nude razne režijske i *debug* informacije o pokrenutim procesima
- ovaj sistem nudi drugim procesima mogućnost da ovim informacijama pristupaju kao običnim tekstualnim fajlovima; npr. UNIX komanda `ps` (lista stanja procesa) je na Linuxu implementirana kao najobičniji neprivilegovani program koji čita i ispisuje ove fajlove
- posebni globalni “fajlovi” u ovom sistemu nude informacije i statistiku o celom sistemu i kernelu
- `/proc/sys` sadrži “fajlove” koji se odnose na parametre kernela i koji dozvoljavaju i upis (promena parametara); na ovaj način administrator utiče na sistem prostim “upisom” u ove “fajlove”

# Ulaz i izlaz

- ◆ Kao i u UNIXu, drajveri uređaja se spolja koriste kao fajlovi – oni su objekti u fajl-sistemu:
  - da bi pristupao uređaju, proces treba da otvori kanal ka uređaju kao i svaki drugi fajl
  - administrator može da kreira poseban “fajl” u sistemu koji referiše određeni drajver, tako da se ulaz/izlaz s tim fajлом usmerava na drajver
  - prava pristupa do uređaja definišu se i kontrolišu kao za fajlove
- ◆ Blokovski orijentisani uređaji (svi diskovi i fleš memorije):
  - za svaki uređaj se vodi posebna lista zahteva
  - lista zahteva je sortirana po redosledu početnog sektora koji se traži
  - raspoređivanje zahteva je po C-SCAN algoritmu
  - tek kada se zahtev opsluži i operacija završi, izbacuje se iz liste i prelazi na sledeći, dok su u međuvremenu stizali novi zahtevi, u cilju poboljšanja performansi; problem: moguće izgladnjivanje!

# Ulaz i izlaz

- ◆ Zato je od verzije 2.6 algoritam C-SCAN modifikovan tako da spreči izladnjivanje:
  - pored glavnog reda sortiranog po broju sektora, postoje još dva pomoćna reda: zahtevi za čitanje i zahtevi za upis
  - kada stigne zahtev, on se smešta u glavni red i u jedan od pomoćnih prema svom tipu
  - pomoćni redovi su sortirani prema *isteku vremenskog roka* (*deadline*): kada pristigne, zahtevu se dodeli vremenski rok u kome se mora opslužiti (0.5s za čitanje i 5s za upis)
  - zahtevi se podrazumevano opslužuju po redosledu u glavnom redu; međutim, ukoliko nekom zahtevu u pomoćnim redovima istekne vremenski rok, on se opslužuje "preko reda"

# Ulaz i izlaz

## ◆ Znakovno-orientisani uređaji:

- svaki drajver registruje se kernelu sa skupom svih funkcija koje implementiraju one operacije fajl sistema koje drajver podržava
- kernel uopšte ne filtrira zahteve uređaju, već ih sve prosleđuje preko registrovanih funkcija; drajver je dužan da adekvatno reaguje na zahteve koje ne podržava
- izuzetak su uređaji koji implementiraju terminale; oni imaju poseban interfejs

# Međuprocesna komunikacija

## ◆ UNIX asinhroni *signali* (*signal*):

- jedan proces ili kernel može da pošalje signal drugom procesu
- signal ne nosi nikakvu drugu informaciju, već je prost događaj
- kada primi signal, odredišni proces prekida trenutni tok kontrole i prelazi na izvršavanje rutine za obradu signala (*signal handler*) koja je definisana za taj signal u tom procesu
- odredišni proces može i da čeka na signal (blokiran) posebnim sistemskim pozivom

◆ Unutar kernela, čekanje na događaje implementira se čekanjem kernel procesa u redovima čekanja; događaj deblokira sve procese koji na njega čekaju

◆ Postoje i klasični semafori za upotrebu u korisničkim procesima

# Međuprocesna komunikacija

## ◆ Razmena podataka između procesa:

- standardni UNIX *pipe* mehanizam u Linuxu je implementiran kao poseban tip "fajla" u fajl sistemu
- mrežna komunikacija
- deljenje memorije između procesa

# Glava 9: Primer operativnog sistema – Windows

Istorijat

Komponente sistema

Programski interfejs Win32

# Istorijat

- ◆ Prva verzija sistema Windows 1.0 1985. je samo školjka za MS-DOS. Isto važi i za verziju 3.1, sve do verzije Windows 95 i 98 (iako i oni koriste 16-bitni DOS kernel)
- ◆ Sredinom 80-ih, Microsoft i IBM sarađuju na izgradnji OS/2 koji je bio pisan na asembleru za jednoprocесorske 80286
- ◆ 1988. Microsoft odlučuje da počne ispočetka razvoj sistema nove generacije ("new technology", NT) koji će podržavati i OS/2 i POSIX API
- ◆ 1988. Microsoft angažuje Dejva Katlera (Dave Cutler), arhitektu DEC VAX/VMS sistema, koji započinje ovaj razvoj
- ◆ Inicijalno je ideja bila da OS/2 API bude prirodno okruženje, ali se vremenom odluka menja u Win32 API
- ◆ Prva verzija Windows NT 3.1 1993. (u vreme 16-bitnog Windows 3.1)

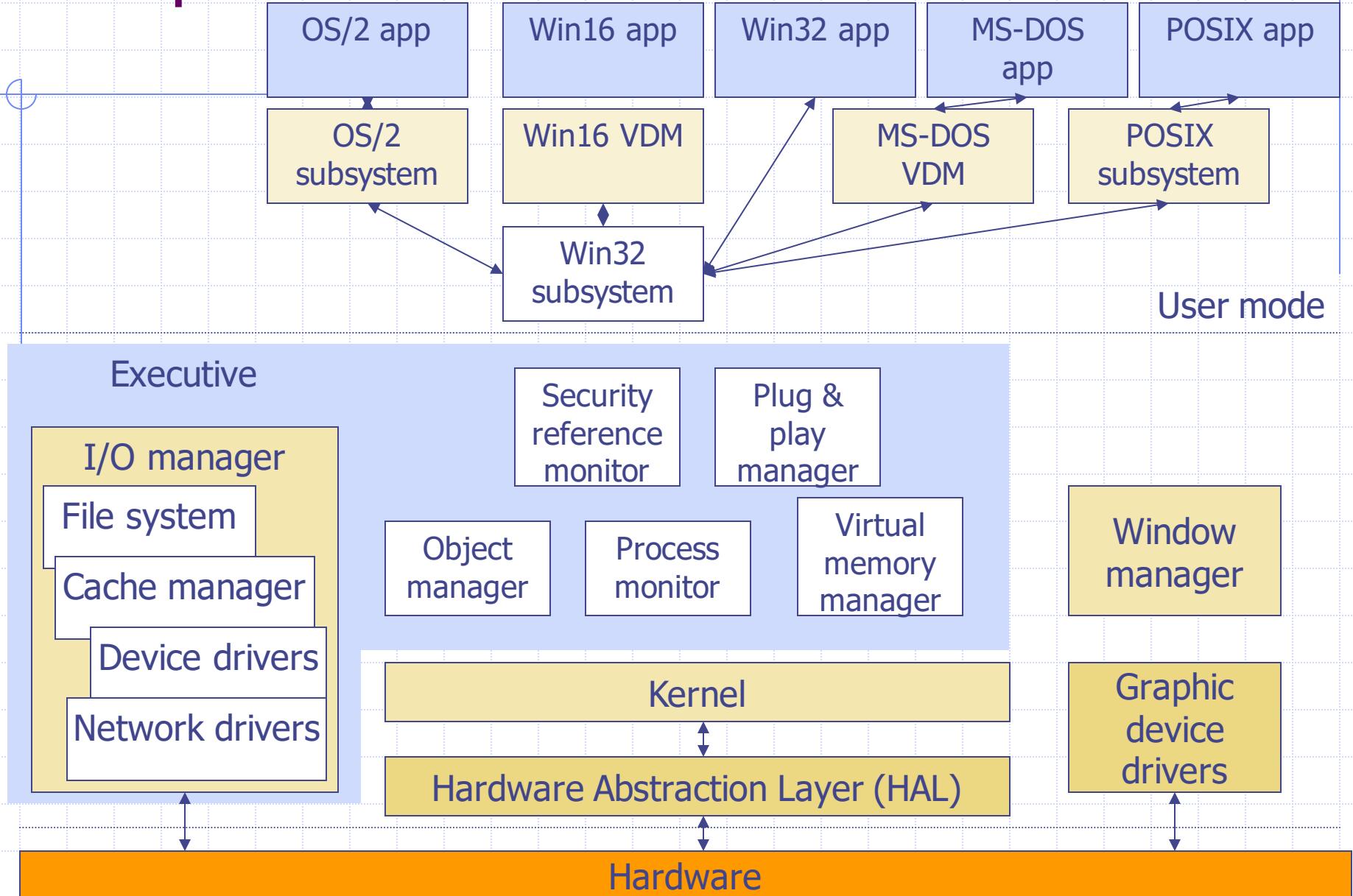
# Istorijat

- ◆ Windows NT 4.0 preuzima GUI od Windows 95 i ugrađeni Web server i browser softver; sve GUI i grafičke rutine premeštene su u kernel radi poboljšanja performansi, uz smanjenje pouzdanosti
- ◆ Windows 2000 (februar 2000) napušta podršku svega osim Intel i kompatibilnih procesora iz tržišnih razloga; uvodi Active Directory, bolju podršku umrežavanju i laptop računarima, *plug-and-play*, distribuirani fajl sistem, podršku više procesora i više memorije
- ◆ Windows XP (oktobar 2001) je nadgradnja Windows 2000 i zamena za Windows 95/98: moderniji GUI za napredniji hardver, bolje umrežavanje i komunikacija (npr. Messenger), značajno povećanje performansi i povećana pouzdanost i sigurnost. Windows XP je prva 64-bitna varijanta Windowsa
- ◆ Slede: Windows Vista (2006), Windows 7 (2009), Windows 8 (2012), Windows 10 (2014)

# Istorijat

- ◆ Windows je višekorisnički OS koji dozvoljava simultani pristup kroz distribuirane usluge ili kroz višestruke instance GUIa preko terminal servera:
  - serverska varijanta terminal servera dozvoljava simultani pristup sa klijentskih računara
  - desktop varijanta dozvoljava multipleksiranje tastature, miša i ekrana između sesija više aktivnih korisnika
- ◆ Paralelno sa verzijama za desktop računare, postoje i serverske verzije: Windows Server 2008, 2011, 2012, 2016, 2019
- ◆ Windows Server ima iste osnovne komponente, ali nudi mnoge druge usluge za podršku Webserver farmi, print/file servera, klastera i velikih centara podataka (do 64GB memorije i 32 procesora na IA32, odnosno 128GB memorije i 64 procesora na IA64)

# Komponente sistema



# Programski interfejs Win32

## ◆ Pristup kernel objektima:

- **CreateXXX()** : kreira/zauzima kernel objekat tipa XXX i vraća "ručku" (*handle*) tipa **HANDLE**; ako ne uspe, vraća 0 ili **INVALID\_HANDLE\_VALUE**
- **CloseHandle()** : oslobađa kernel objekat na koji ukazuje data "ručka" (*handle*); ako objekat više niko ne koristi, sistem ga briše

## ◆ Tri načina da procesi dele kernel objekte:

1. proces-dete nasleđuje objekat (ručku) roditelja:

- ◆ kada kreira objekat sa **CreateXXX()**, roditelj prosleđuje strukturu **SECURITY\_ATTRIBUTES** kao argument sa poljem **bInheritHandle** postavljenim na **true**
- ◆ kada kreira proces-dete sa **CreateProcess()**, roditelj prosleđuje **true** kao argument **bInheritHandle** da bi se ručke nasledile
- ◆ proces-dete mora da zna koje se ručke dele i kako do njih da pristupi (npr. preko komandne linije)
- ◆ proces-dete traži pristup do deljenog objekta
- ◆ dalje procesi mogu da komuniciraju preko deljenog objekta

# Programski interfejs Win32

```
SECURITY_ATTRIBUTES sa ;
sa.nLength = sizeof(sa) ;
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
Handle aSemaphore = CreateSemaphore(&sa,1,1,NULL) ;
char commandLine[132];
ostrstream ostring(commandLine,sizeof(commandLine)) ;
ostring<<aSemaphore<<ends ;
CreateProcess("program.exe",commandLine,NULL,NULL,TRUE,...)
```

2. jedan proces kreira objekat i pri kreiranju mu zadaje simboličko ime; drugi proces otvara postojeći (ne kreira novi) objekat sa zadatim simboličkim imenom (`OpenXXX()`); problem: sukob imena (*namedclashing*) jer je prostor simboličkih imena globalan za ceo sistem, nezavisno od tipa objekta

```
// Process A:
HANDLE aSemaphore = CreateSemaphore(NULL,1,1,"MySem1") ;
// Process B:
HANDLE bSemaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
FALSE,"MySem1") ;
```

# Programski interfejs Win32

- 3. proces koji je kreirao objekat koristi neki drugi metod da prosledi vrednost svoje ručke do objekta, a onda onaj koji je primio tu vrednost poziva **DuplicateHandle()** da bi dobio svoju ručku do istog deljenog objekta
- ◆ **Ručka instance (instance handle):**
  - svaki DLL ili izvršni fajl učitan u adresni prostor procesa identificuje se ručkom instance – virtualna adresa njegove lokacije
  - program može dobiti ručku za neki modul u svom adresnom prostoru pozivom **GetModuleHandle()** sa argumentom koji zadaje ime modula; ako je ime **NULL**, dobija se bazna adresa samog procesa
- ◆ Nit (*thread*) se može kreirati u stanju suspenzije – ne izvršava se dok ga neka druga nit ne pokrene sa **ResumeThread()**. Nit se suspenduje sa **SuspendThread()**. Broje se pozivi ove dve funkcije, nit se pokreće kada je **Resume** pozvano isti broj puta kao i **Suspend**)

# Programski interfejs Win32

- ◆ Postoje sinhranizacioni objekti – semafor i mutex
- ◆ Postoji i sinhronizacija pomoću `WaitForSingleObject()` i `WaitForMultipleObjects()`
- ◆ Međusobno isključenje kritične sekcije:  
`InitializeCriticalSection()`,  
`EnterCriticalSection()`,  
`LeaveCriticalSection()` – efikasnije nego korišćenje drugih sinhronizaconih primitiva jer ne alocira kernel objekte
- ◆ Razmena poruka između procesa:
  - asinhrono: `PostMessage()`, `PostThreadMessage()`
  - sinhrono: `SendMessage()`, `SendThreadMessage()`,  
`SendMessageCallback()`
  - ako proces ne pozove `GetMessage()` da primi poruku tokom oko 5s, sistem je označa kao "Not Responding"
  - svaka Win32 nit ima svoj red primljenih poruka (u Win16 red poruka je bio zajednički)

# Programski interfejs Win32

## ◆ Alokacija i dealokacija virtuelne memorije:

- **VirtualAlloc()** i **VirtualFree()**
- moguće je definisati virtuelnu adresu gde se alocira memorija; mora biti veća od 10000h (prvih 64KB adresnog prostora je zabranjeno za korišćenje)
- veličina se izražava u multiplima veličine stranice
- proces može da zaključa deo memorije (spreči zamenu stranica) sa **VirtualLock()**; najviše 30 stranica, osim ako ne poveća veličinu svog radnog skupa sa **SetProcessWorkingSetSize()**

## ◆ Memorijski preslikani fajlovi:

- još jedan način da procesi dele podatke – oba preslikaju isti fajl u deo svog adresnog prostora
- ako pozove **CreateFileMapping()** sa argumentom -1 kao ručkom za fajl, ne koristi se fajl, već se deo adresnog prostora samo deli sa drugim procesima

# Programski interfejs Win32

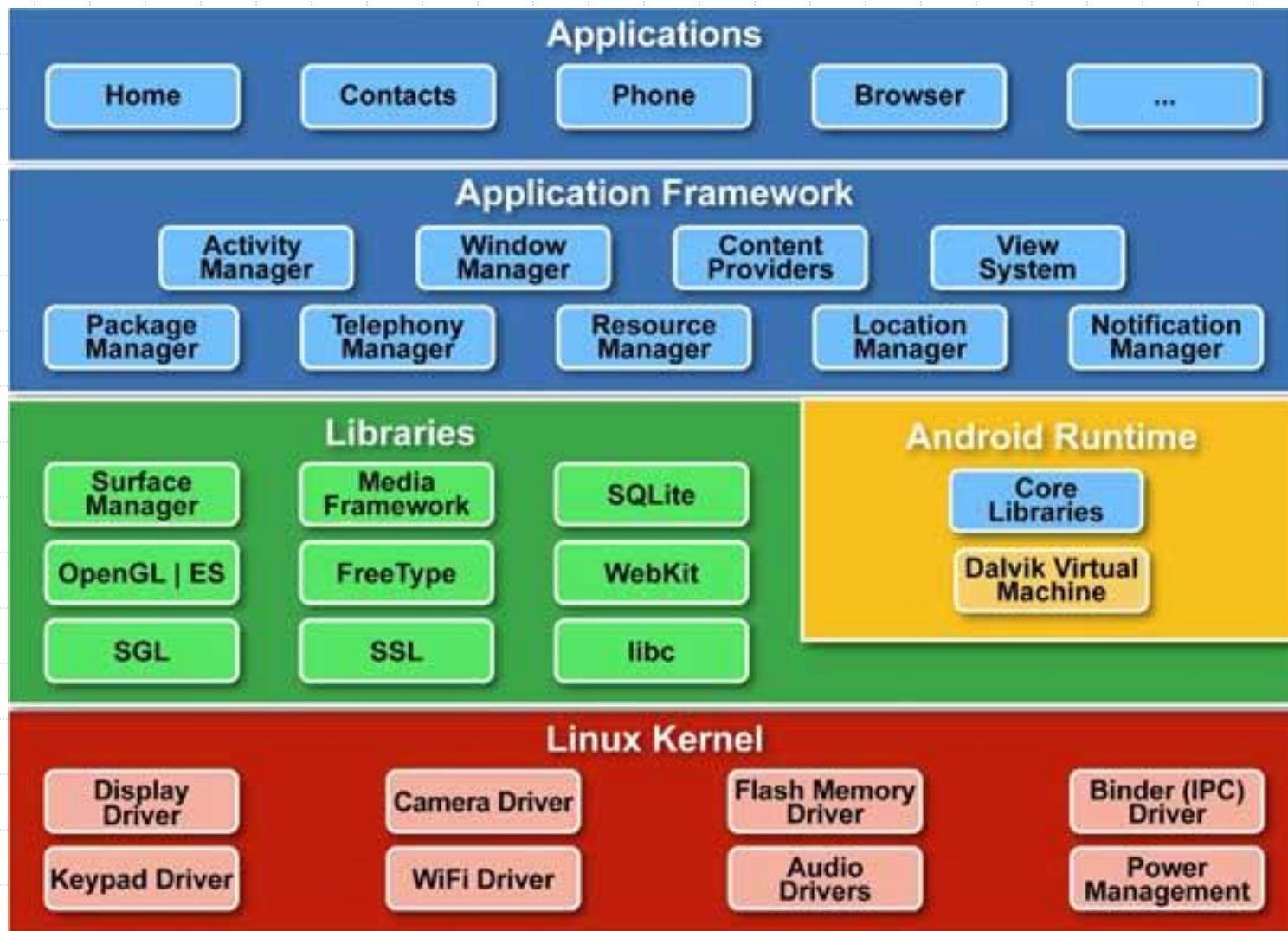
- ◆ Posebni rezervisani delovi adresnog prostora koji se koriste za dinamičku alokaciju i dealokaciju memorije – *heap*:
  - svaki proces ima podrazumevani *heap* veličine 1MB; mnoge Win32 API funkcije koriste ovaj *heap*, pa je pristup do njega sinhronizovan za konkurentne niti tog procesa (međusobno isključenje)
  - proces može kreirati i uništavati *heap* sa `HeapCreate()` i `HeapDestroy()`
  - alokacija i dealokacija prostora: `HeapAlloc()`, `HeapFree()`
  - međusobno isključenje pristupa jednom *heap*-u (ne zaključavanje stranica od zamene): `HeapLock()`, `HeapUnlock()`
- ◆ Statičke programske promenljive koje su zasebne (lokalne) za svaku nit:

```
_declspec(thread) long int cur_pos = 0;
```

# Glava 10: Primer operativnog sistema – Android

## Arhitektura i komponente sistema

# Arhitektura i komponente sistema



# Glava 11: Zaključak

Šta je naučeno

Šta dalje

Domaći zadatak

Ispit

Pitanja i diskusija