

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 1

Početni kurs

# Sadržaj

---

I Uvod

II Osnovi arhitekture računara

III Upravljanje procesima

IV Upravljanje memorijom

V Ulazno/izlazni sistemi

VI Fajl sistemi

VII Zaključak

---

# I Uvod

---

O predmetu

Pojam, namena i funkcije operativnih sistema

Istorijat i vrste operativnih sistema

# Glava 1: O predmetu

---

Sadržaj, ciljevi i preduslovi

Organizacija nastave

Praktičan rad

Kolokvijumi

Ispit

Literatura

Kontakti

---

# Sadržaj, ciljevi i preduslovi

## ◆ Sadržaj:

- Namena i funkcije operativnih sistema
- Osnovni koncepti operativnih sistema
- Osnovni algoritmi, problemi i rešenja
- Principi funkcionisanja, projektovanja i implementacije

## ◆ Ciljevi:

- Upoznati se sa namenom, funkcijama, konceptima i principima funkcionisanja, projektovanja i implementacije operativnih sistema
- Steći opšte, fundamentalno znanje primenjivo na operativne sisteme uopšte, nevezano ni za jedan konkretan sistem
- Osposobiti se za razumevanje i korišćenje postojećih i projektovanje sopstvenih specijalizovanih sistema
- Mnogim ljudima su OS magija: ovaj kurs ih demistifikuje!

# Sadržaj, ciljevi i preduslovi

## ◆ Preduslovi:

- Poznavanje korišćenja računara i osnovnih delova hardvera i softvera (Praktikum iz korišćenja računara)
- Dobro poznavanje osnovnih principa i tehnika proceduralnog i OO programiranja
- Poznavanje osnova arhitekture računara
- Dobro savladano gradivo predmeta: Programiranje 1 i 2, Praktikum iz programiranja, Objektno orijentisano programiranje 1, Algoritmi i strukture podataka, Osnovi računarske tehnike, Arhitektura računara
- Veština u programiranju na jezicima C i C++
- **Samostalan, praktičan i kontinuiran rad: operativni sistemi se nabolje 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 (projekat)
- student samostalno izgrađuje i integriše delove jednog malog, ali potpuno funkcionalnog operativnog sistema
- radi se i brani samostalno, usmeno i na računaru
- ulazi u konačnu ocenu sa 30% plus 10% bonusa

# 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 (jedan ili više) od 3 kolokvijuma, ali samo u jednom terminu godišnje koji je u septembarskom ispitnom roku (zajedno sa ispitom)
- ◆ za prolaz je potrebno u zbiru prikupiti bar 25% 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 junskog roka	U redovnom terminu junskog 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", Mikro knjiga, 2005.

## ◆ 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/)

## ◆ Koristan materijal:

- D. Milićev, "Programiranje u realnom vremenu", skripta za predavanja, ETF, <http://prv.etf.rs>

# Kontakti

◆ Sajt predmeta:

<http://os.etf.rs>

◆ Nastavnik: prof. dr Dragan Milićev

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

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

◆ Asistenti:

- Živojin Šuštran, [zika@etf.rs](mailto:zika@etf.rs)
- Milana Prodanov, [milana@etf.rs](mailto:milana@etf.rs)

# Glava 2: Pojam, namena i funkcije OS

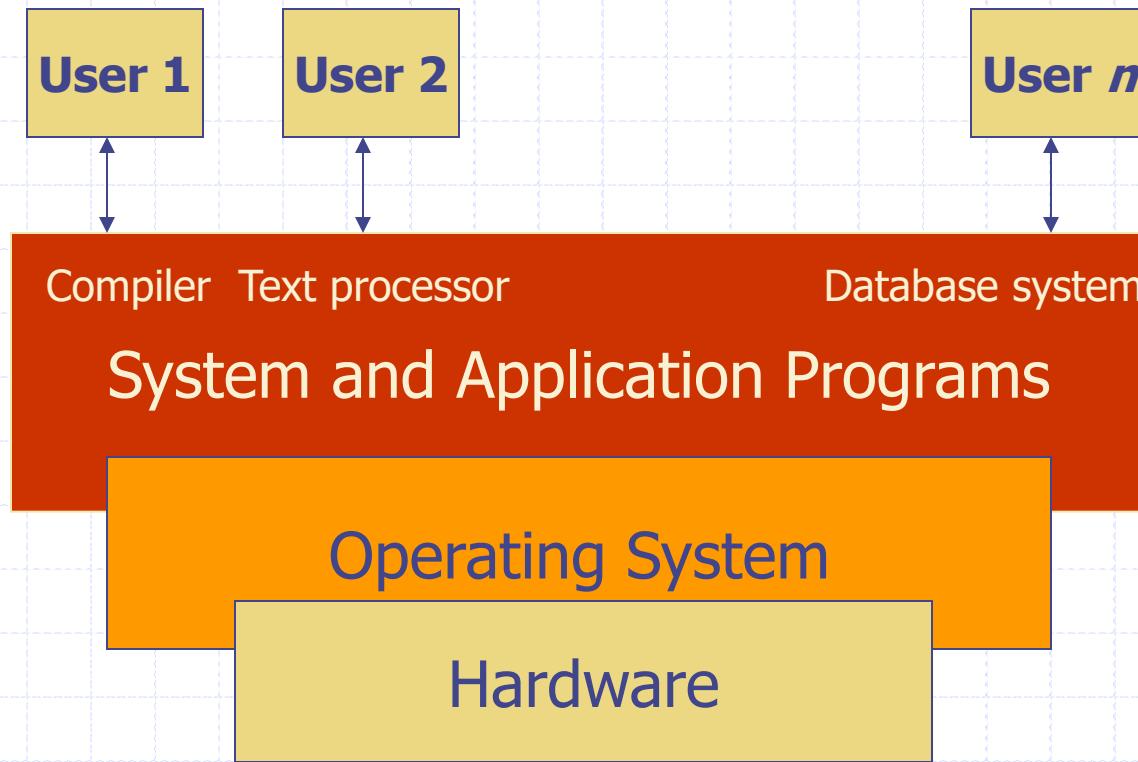
Šta je operativni sistem?

Zašto postoji OS?

Osnovni zadaci i funkcije OS

Šta sve obuhvata OS?

# Šta je operativni sistem?



OS je program koji služi kao posrednik između korisnika, odnosno njegovih programa, i računarskog hardvera

# Zašto postoje OS?

- ◆ OS postoje jer su oni razuman način da se reši problem izgradnje upotrebljivog računarskog sistema
- ◆ Aplikativni programi obuhvataju neke zajedničke operacije, kao što su operacije sa U/I uređajima
- ◆ Upravo te zajedničke operacije upravljanja HW resursima računara skupljene na jedno mesto i pogodne za upotrebu od strane više programa čine OS
- ◆ Čini aplikacije prenosivim apstrahujući HW

# Osnovni zadaci i funkcije OS

## ◆ Osnovni zadaci OS:

- Učiniti računarski sistem *pogodnim* za upotrebu: lakše sa njim nego bez njega
- Postići *efikasnost* računarskog sistema: posebno važno za velike, serverske sisteme koje deli mnogo korisnika

## ◆ Osnovne funkcije OS:

- upravljanje resursima: CPU, OM, I/O, ... – konačno i ograničeno učiniti (prividno) neograničenim
- zaštita računarskih resursa od nepravilne upotrebe ili zloupotrebe

# Šta sve obuhvata OS?

## ◆ Različita shvatanja opsega operativnih sistema:

- Tradicionalno shvatanje:
  - ◆ *kernel* (jezgro) – program koji obavlja osnovne funkcije OS i uvek se nalazi u memoriji
  - ◆ skup uslužnih sistemskih programa
  - ◆ *shell* (školjka) – komandni ili grafički korisnički interfejs (GUI) prema funkcijama OS
- Moderno shvatanje: sve ovo i još
  - ◆ luksuzan GUI
  - ◆ skup uslužnih aplikativnih programa
  - ◆ programi za Internet usluge
  - ◆ ...
- “Sve što proizvođač isporuči pod tim nazivom”.  
Primer: Microsoft Windows

# Glava 3: Istorijat i vrste OS

---

Paketni sistemi

Sistemi sa vremenskom raspodelom

Personalni računarski sistemi

Multiprocesorski sistemi

Distribuirani sistemi

Sistemi za rad u realnom vremenu

---

# Paketni sistemi

## ◆ Paketni sistemi (*batch system*):

- prvi računari 1960-ih i 70-ih
- ulazni uređaji: čitač kartica i magnetne trake
- izlazni uređaj: linijski štampač, bušač kartica i magnetne trake
- nema interakcije sa korisnikom – korisnik pripremi posao (*job*) za obradu (program+ulazni podaci), operater to postavi na sistem, pokrene izvršavanje i vrati korisniku rezultat (izlazni podaci ili izveštaj o grešci)

## ◆ OS je sasvim jednostavan:

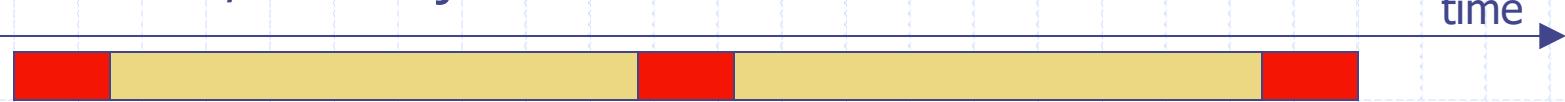
- jedini zadatak je preći sa jednog posla na drugi
- stalno prisutan u memoriji

## ◆ Operater pravi *paket* (*batch*) srodnih poslova i pokreće ih kao grupu



# Paketni sistemi

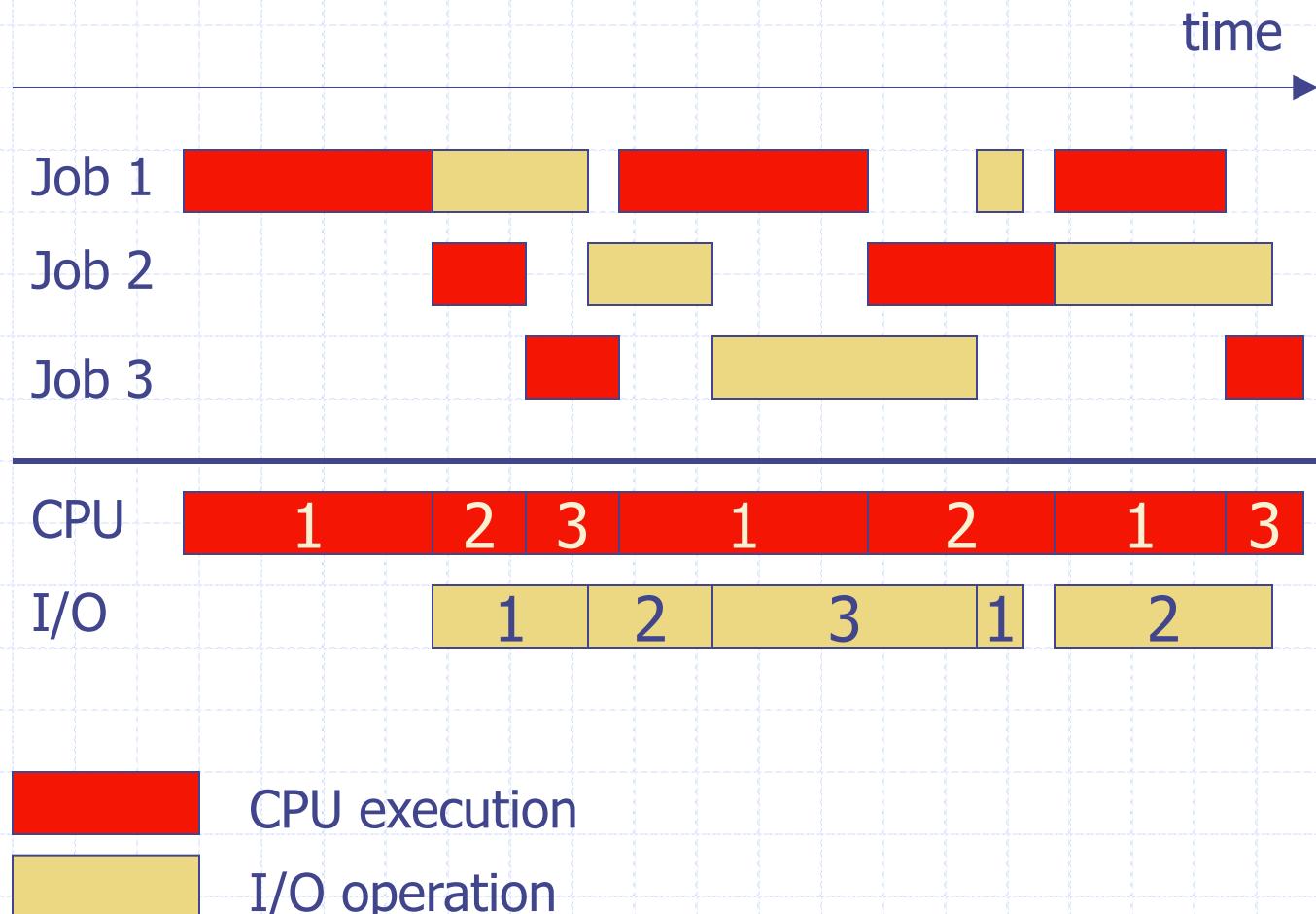
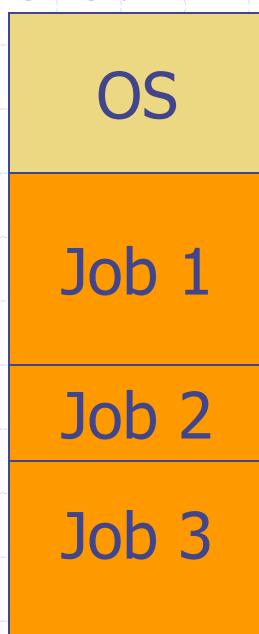
- ◆ Problem: CPU je često neiskorišćen čekajući na I/O operaciju:
  - CPU ciklusi: mikrosekunde ili nanosekunde
  - Odziv I/O uređaja: milisekunde ili sekunde
- ◆ Uvođenje diskova omogućilo je direktni pristup pojedinačnom poslu, bez potrebe njihove sekvencijalizacije
- ◆ *Raspoređivanje poslova (job scheduling)*: iz skupa raspoloživih poslova (*pool*), izabratи onaj ili one koji će se učitati u memoriju i izvršavati
- ◆ *Multiprogramiranje (multiprogramming)*: izvršavati više poslova upredо; dok jedan čeka na završetak I/O operacije, CPU izvršava drugi posao



# Multiprogramiranje

Generički oblik posla:

```
begin  
    compute;  
    I/O;  
    compute;  
    I/O;  
    ...  
end.
```



# Multiprogramiranje

## ◆ Otvorena pitanja:

- kako iz skupa svih poslova izabrati one koji će se učitati u memoriju i izvršavati – *job scheduling*
- kako smestiti poslove različite veličine u memoriju – *memory management*
- kada posao koga CPU trenutno izvršava zatraži I/O operaciju, kako izabrati sledeći koga će CPU izvršavati – *CPU scheduling*
- kako preći sa izvršavanja jednog na izvršavanje drugog posla – *context switch*
- kako opsluživati zahteve za I/O operacijom na deljenom uređaju – *I/O device scheduling*
- kako sprečiti da jedan posao slučajno ili namerno ugrozi memorijski sadržaj drugog – *protection*
- kako se izbaviti iz situacije kada jedan program krene u beskonačnu petlju - *preemption*

# Sistemi sa raspodelom vremena

- ◆ Paketni sistemi omogućuju efikasno deljenje resursa, ali nisu interaktivni (ne interaguju sa korisnikom tokom rada)
- ◆ Posao sada, osim na završetak I/O operacije sa uređajem, može da čeka i na (re)akciju korisnika preko tastature ili miša
- ◆ Vreme ljudske reakcije je reda  $\sim 1/2$  do najmanje  $\sim 1/7$  sekunde. Vreme reakcije računara na akciju korisnika je tipično kratko (reda nekoliko desetina ili stotina ms)
- ◆ Važno za interaktivne sisteme je vreme odziva (*response time*) sistema na akciju korisnika – treba da bude reda sličnog kao i vreme ljudske reakcije, tj. do  $\sim 1\text{s}$

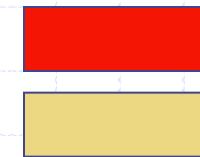
# Sistemi sa raspodelom vremena

Generički oblik interaktivnog procesa:

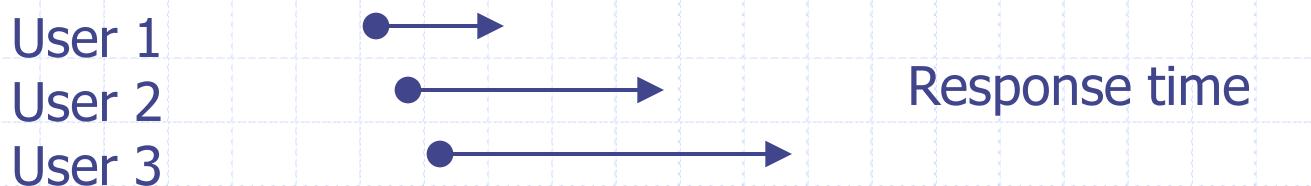
loop

```
    wait for user input;  
    compute reaction;  
    display result;
```

end.



CPU execution  
User action



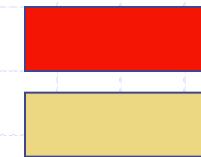
# Sistemi sa raspodelom vremena

## Generički oblik interaktivnog procesa:

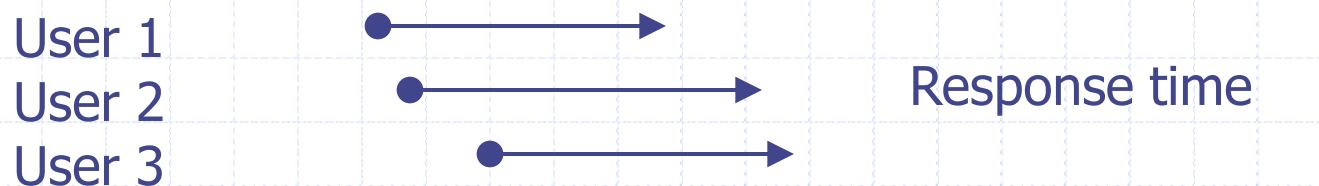
loop

```
    wait for user input;  
    compute reaction;  
    display result;
```

end.



CPU execution  
User action



# Sistemi sa raspodelom vremena

- ◆ Uopštenje principa multiprogramiranja na interaktivne višekorisničke sisteme – raspodela vremena (*time-sharing, multitasking*)
- ◆ Ideja: osim kada sam zatraži I/O operaciju, posao može da izgubi procesor i kada mu istekne dodeljeno CPU vreme
- ◆ OS dodeljuje CPU vreme svakom poslu i relativno često preuzima procesor
- ◆ Rezultat: svaki korisnik ima utisak da računar radi samo za njega sa dovoljno dobrim i ujednačenim vremenom odziva, a računar opslužuje više korisnika
- ◆ Program koji je učitan u memoriju i izvršava se uporedo sa drugim programima naziva se *proces* (*process*)

# Sistemi sa raspodelom vremena

- ◆ Otvorena pitanja: ista kao i kod multiprogramiranja, i još:
  - Da bi se posao izvršavao sa razumnim vremenom odziva, potrebno je ponekad izbacivati delove posla iz memorije i ubacivati delove drugog posla, pošto ne mogu svi celi da stanu. Posao nije uvek ceo u memoriji.

Tehnika koja ovo omogućava – *virtuelna memorija (virtual memory)*.

- Fajl sistem
  - Upravljanje diskovima
- ◆ Početak 1960-ih. Šira upotreba 1970-ih.

Danas su svi višekorisnički interaktivni sistemi opšte namene ovakvi!

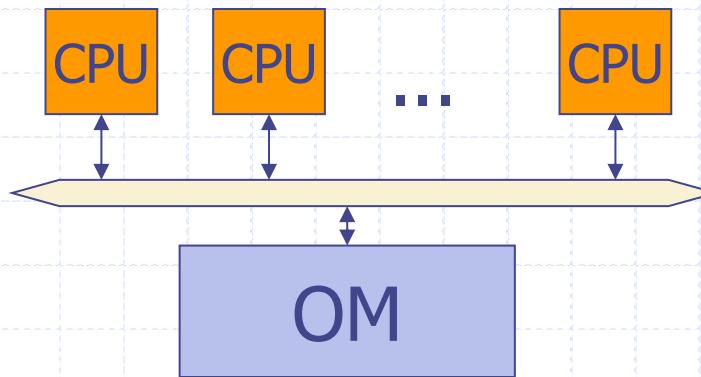
# Sistemi sa raspodelom vremena

- ◆ Razlikovati pojam *programa* od *procesa*: proces predstavlja izvršavanje nekog programa nad datim skupom podataka. Više procesa može biti aktivirano nad istim programom
- ◆ Primeri kombinacija:
  - monoprogramske, monoprocesne: specijalizovani ili ugrađeni (*embedded*) sistem za upravljanje nekim HW sistemom
  - multiprogramske, monoprocesne: personalni računar starije generacije koji može imati u memoriji i izvršavati samo jedan program u jednom trenutku; stariji paketni sistemi
  - monoprogramske, multiprocesne: specijalizovani sistem posebne namene sa više korisnika
  - multiprogramske, multiprocesne: sistemi sa raspodelom vremena, svi moderni OS opšte namene
- ◆ Višekorisnički sistem:
  - samo jedan korisnik u datom trenutku (npr. personalni sistem)
  - više korisnika u jednom trenutku; implicitno multiprocesni

# Personalni računarski sistemi

- ◆ Počeci 1970-ih: monoprocesni, monokorisnički
- ◆ Današnji: multiprocesni, jedno- ili višekorisnički
- ◆ Aspekt efikasnosti upotrebe manje važan nego ranije (jedan ili malo korisnika u jednom trenutku)
- ◆ Važniji aspekti:
  - interaktivnost
  - vreme odziva
  - multitasking
  - pogodnost upotrebe
  - umrežavanje i pristup Internetu
  - zaštita
  - pouzdanost

# Multiprocesorski sistemi



◆ Više procesora sa deljenom memorijom

◆ Ciljevi:

- povećati propusnu moć (*throughput*) – količina urađenog posla u jedinici vremena; pažnja: povećanje nije linearno!
- ušteda zbog deobe periferijskih uređaja
- povećanje pouzdanosti tolerancijom otkaza (*fault tolerance*)

◆ Simetrični i asimetrični sistemi

◆ Asimetrični sistemi:

- specijalizovani I/O mikroprocesori (npr. za disk, tastaturu)
- *master/slave* sistemi

# Distribuirani sistemi

- ◆ Skup procesora bez zajedničke memorije, povezanih komunikacionom mrežom:
  - specijalizovani homogeni distribuirani sistemi
  - lokalne mreže (*Local Area Network*, LAN)
  - mreže šireg područja (*Wide Area Network*, WAN)
  - Internet
- ◆ Svi moderni OS podržavaju umrežavanje:
  - moduli za komunikacione protokole (TCP/IP, PPP)
  - Internet usluge
  - pristup fajlovima preko mreže (distribuirani fajl sistemi)
- ◆ Distribuirani OS: OS koji na distribuiranom sistemu stvara utisak jedinstvenog skupa dosupnih resursa

# Sistemi za rad u realnom vremenu

- ◆ Sistem za rad u realnom vremenu (*real-time system*, RTS): sistem za koga nije samo važno da rezultat bude logički ispravan, već i da je proizведен *na vreme*
- ◆ *Hard RTS*: definisani vremenski rokovi moraju da budu ispoštovani, inače će doći do tragičnih posledica po živote i zdravlje ljudi, materijalna sredstva ili životnu sredinu
- ◆ *Soft RTS*: definisane vremenske rokove treba poštovati, ali je dopustivo da se oni ponekad i prekorače, sve dok su *performanse* sistema u okvirima zadovoljavajućeg

# Sistemi za rad u realnom vremenu

## ◆ Karakteristike Hard RTS OS:

- ne postoje HW i SW komponente koje bi unosile neodređenost u vremenske karakteristike (kašnjenja, vreme odziva): diskovi, virtuelna memorija
- nema raspodele vremena
- procesi su periodični ili sporadični
- raspoređivanje je veoma strogo i karakteristično (po prioritetima ili po vremenskom roku)
- nijedan OS opšte namene nema karakteristike potrebne za hard RTS

## ◆ Karakteristike Soft RTS OS:

- mora postojati podrška za raspoređivanje po prioritetima i kontrolu vremenskih rokova
- mnogi moderni OS imaju ovakvu podršku

# II Osnovi arhitekture računara

Elementi arhitekture procesora

Mehanizam prekida

Ulag/Izlaz

Virtuelna memorija

Memorijska hijerarhija

# Glava 4: Elementi arhitekture procesora

Aritmetičko-logička jedinica

Programski dostupni registri

Programski brojač

Izvršavanje instrukcije

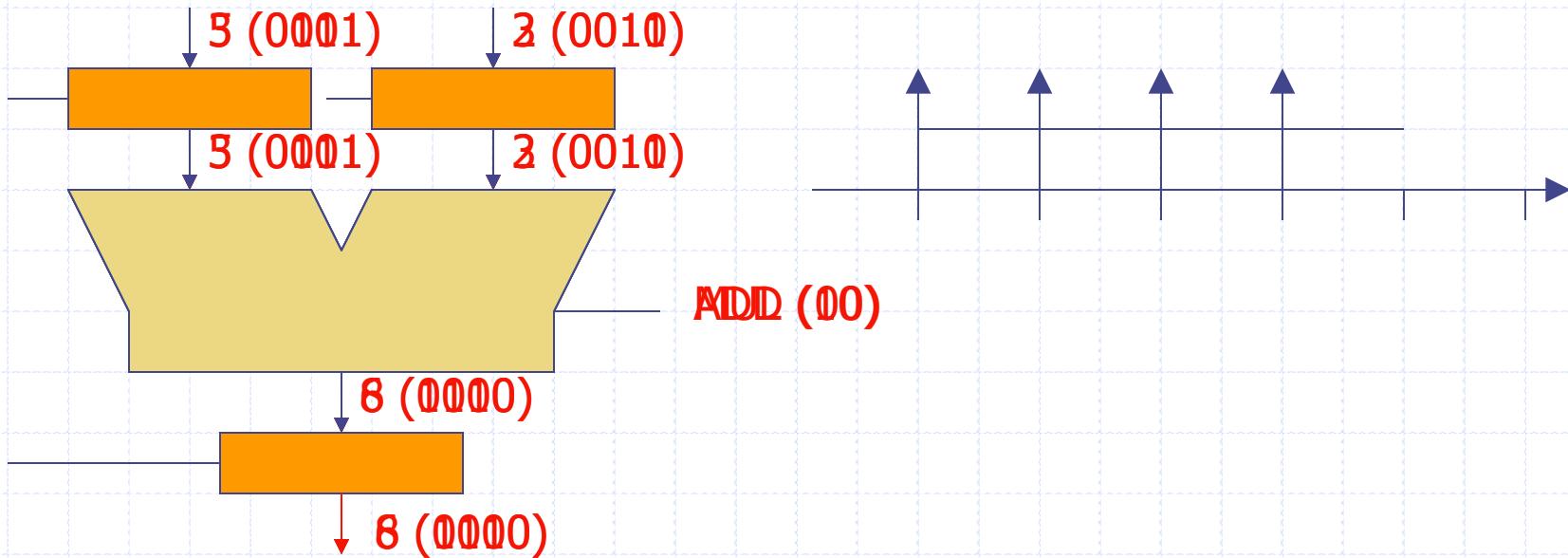
Načini adresiranja

Potprogrami i stek

Kontekst procesora

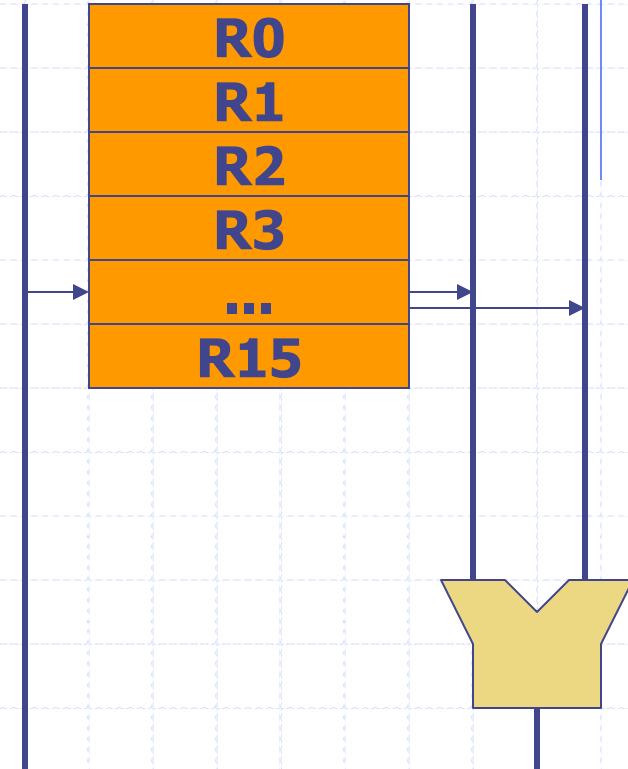
# Aritmetičko-logička jedinica

ALU (*Arithmetic-Logic Unit*) je logička mreža koja izvršava aritmetičku ili logičku operaciju zadatu kontrolnim signalima nad jednim ili dva binarna operanda i proizvodi binarni rezultat.



# Programski dostupni registri

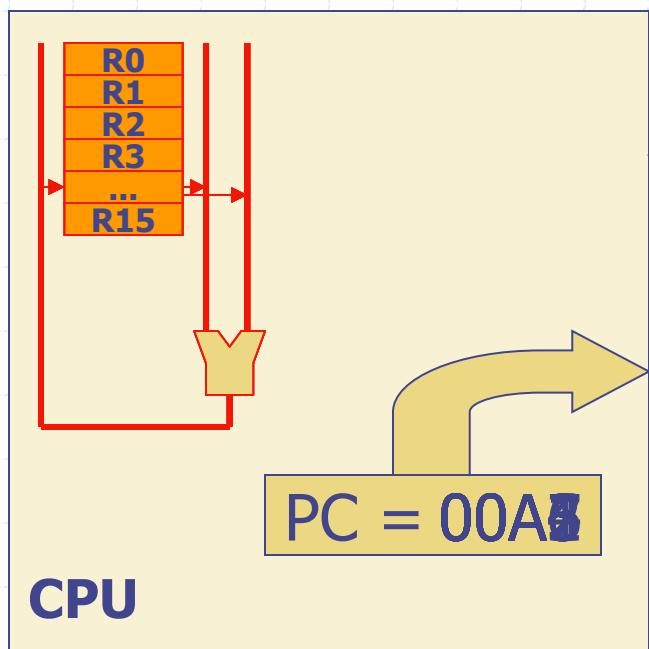
- ◆ Registri koji se mogu referencirati u instrukcijama. Čuvaju svoj sadržaj od jedne do druge instrukcije
- ◆ Registri za podatke (*data registers*): registri koji se isključivo ili dominantno koriste za smeštanje operanada i rezultata instrukcija
- ◆ Registri za adrese (*address registers*): registri koji se isključivo ili dominantno koriste za smeštanje adresa operanada i rezultata instrukcija koji se inače nalaze u memoriji
- ◆ RISC procesori: ortogonalna arhitektura, svi registri su ravnopravni, mogu se koristiti i za adrese i za podatke - registarski fajl



# Programski brojač

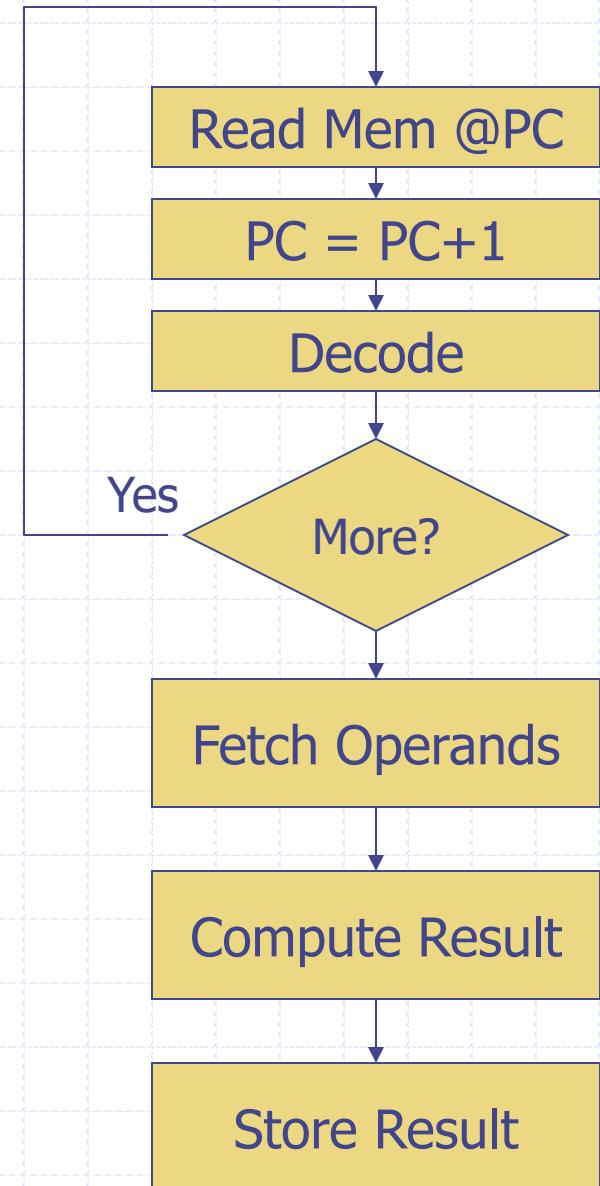
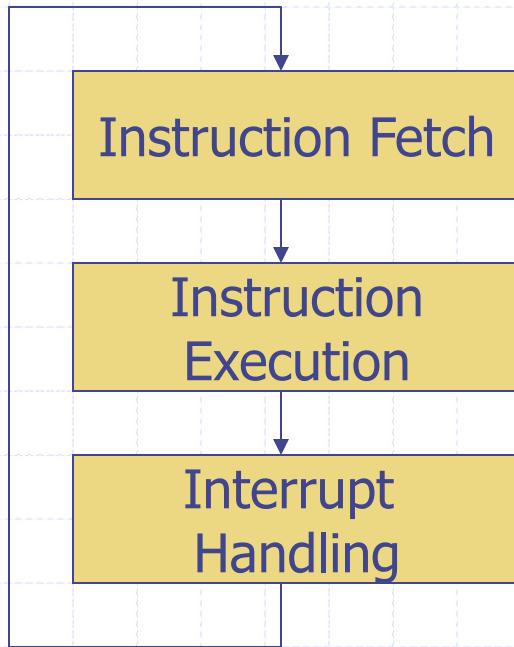
Programski brojač (*program counter*, PC) je registar čija vrednost predstavlja memoriju adresu instrukcije koja je naredna za izvršavanje

```
int a,b,c,d; // R1 ,R2 ,R3 ,R4  
...  
d=c- (a+b) *c
```



00A0	...
00A1	0000
00A2	0102
00A3	0200
00A4	0003
00A5	0104
00A6	0300
00A7	...
00A8	...
00A9	...
RD	...
WR	...
00AA	...
00AB	...
00AC	...
00AD	...

# Izvršavanje instrukcije



# Načini adresiranja

- ◆ Neposredno (*immediate*): operand (samo izvorišni) je neposredno zapisan u samoj instrukciji:

`LOAD ... , #0F1Ah`

`LOAD ... , #Const`

- ◆ Registarsko direktno (*register direct*): operand je u registru koji je zapisan u instrukciji:

`LOAD R2, ...`

- ◆ Memorijsko direktno (*memory direct*): operand je u memoriji na adresi koja je zapisana u instrukciji:

`LOAD ... , [0F1Ah]`

`LOAD ... , [VarX]`

# Načini adresiranja

- ◆ Registarsko indirektno (*register indirect*): operand je u memoriji na adresi koja je zapisana u registru koji je zapisan u instrukciji:

`LOAD . . . , [R0]`

- ◆ Registarsko indirektno sa pomerajem (*register indirect with displacement*): operand je u memoriji na adresi koja se dobija sabiranjem sadžaja registra koji je zapisan u instrukciji i konstante koja je zapisana u instrukciji:

`LOAD . . . , 0F1Ah [R0]`

`LOAD . . . , VarX [R0]`

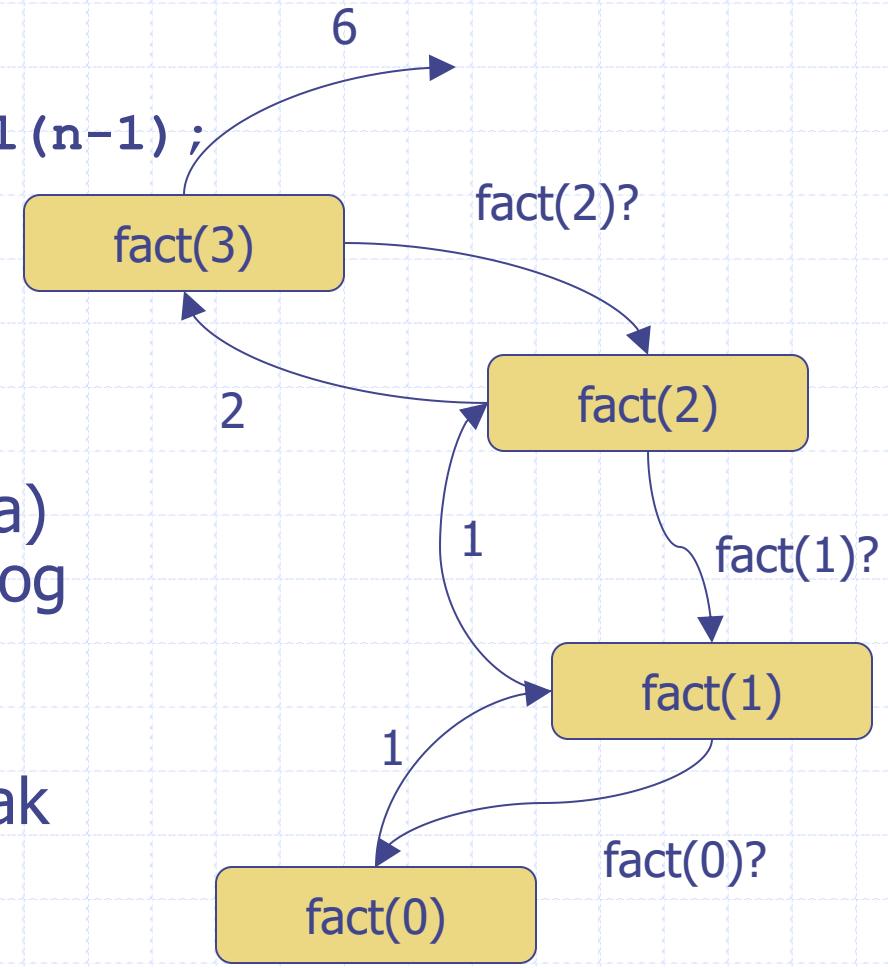
# Potpogrami i stek

Kako kodovati sledeću funkciju?

```
int factoriel (int n) {  
    if (n==0) return 1;  
    else return n*factoriel(n-1);  
}
```

Poenta: u datom trenutku postoji više poziva ("inkarnacija", aktivacija, izvršavanja, instancijalizacija) iste funkcije (izvršavanja istog programskog koda)  
**factoriel()**.

Svaka do njih ima svoj primerak ("inkarnaciju", instancu) podatka (argumenta) **n**.



# Potprogrami i stek

## ◆ Problem:

- Ako se isti potprogram poziva sa više mesta i to rekurzivno, kako znati gde se vratiti?
- Ako se isti potprogram poziva rekurzivno, kako da isti kod uvek radi sa odgovarajućom "inkarnacijom" svojih lokalnih podataka i argumenata?

## ◆ Ideja:

- Prilikom poziva potprograma, pamtiti (redosledom poziva) mesto odakle je potprogram pozvan, kako bi se izvršavanje moglo tamo vratiti (Ivica i Marica)
- Prilikom povratka iz potprograma, uvek uzimati poslednju sačuvanu adresu povratka i tamo skočiti; izbaciti tu adresu iz spiska

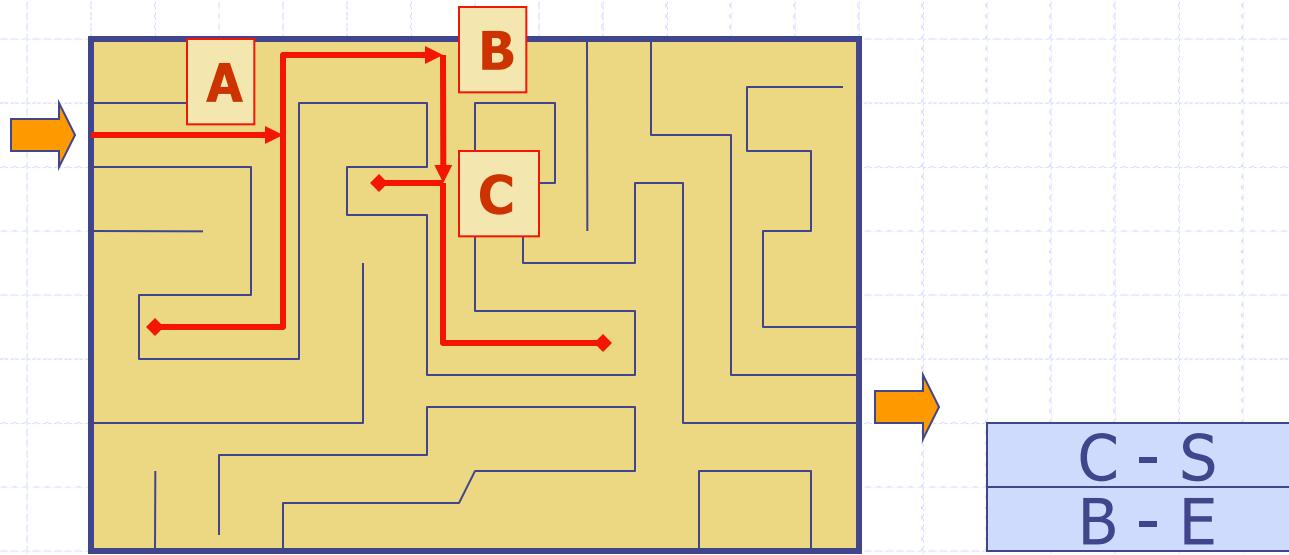
## ◆ Struktura koja podržava ovaj protokol - stek (*stack*):

- linearno uređena struktura sa dve operacije
- operacija *push* smešta datu vrednost na "vrh" steka (na kraj liste)
- operacija *pop* uzima vrednost sa "vrha" steka (sa kraja liste) i izbacuje je sa steka

◆ I lokalni podaci se mogu čuvati na steku, po istom principu.  
Potprogram treba da adresira podatke relativno u odnosu na vrh steka!

# Potpogrami i stek

## ◆ Primer: pronalaženje puta kroz labyrin



**Push A - N**  
**Pop**  
**Push B - E**  
**Push C - S**

# Potpogrami i stek

## ◆ Implementacija steka na nivou procesora:

- Jedan od registara opšte namene izdvaja se da služi kao "pokazivač vrha steka" (*stack pointer*) . Njegova vrednost predstavlja adresu lokacije u memoriji gde je poslednji stavljeni podatak (ili prva slobodna lokacija). Npr. neka je to R15
- Operacija *push*, npr. Push R0:  
`INC R15 ; stek raste "na gore"`  
`STORE [R15], R0`

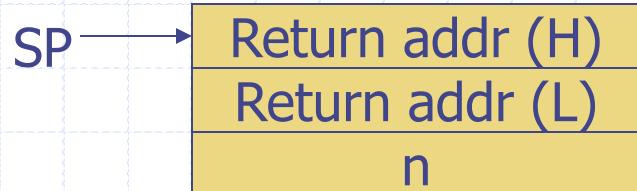
- Operacija *pop*, npr. Pop R0:  
`LOAD R0, [R15]`  
`DEC R15`

## ◆ Posebna podrška procesora: postoji poseban, specijalizovani registar koji služi kao pokazivač steka – SP i posebne instrukcije Push i Pop:

`PUSH R0`  
`POP R0`

# Potprogrammi i stek

```
int factoriel (int n) {  
    if (n==0) return 1;  
    else return n*factoriel(n-1);  
}
```



Fact:	LOAD R1, #n[SP]	; R1:=n; #n=-2 (11...110b)
	CMP R1, #0	; if (n==0)
	JMPNE Else	
Then:	LOAD R0, #1	; R0 is result
	POP PC	; return
Else:	SUB R0, R1, #1	; R0:=n-1
	PUSH R0	; call factoriel(n-1)
	CALL Fact	; PUSH PC, JMP Fact
	POP R1	; pop argument
	LOAD R1, #n[SP]	; R1:=n
	MUL R0, R1, R0	; R0:=n*factoriel(n-1)
	POP PC	; return

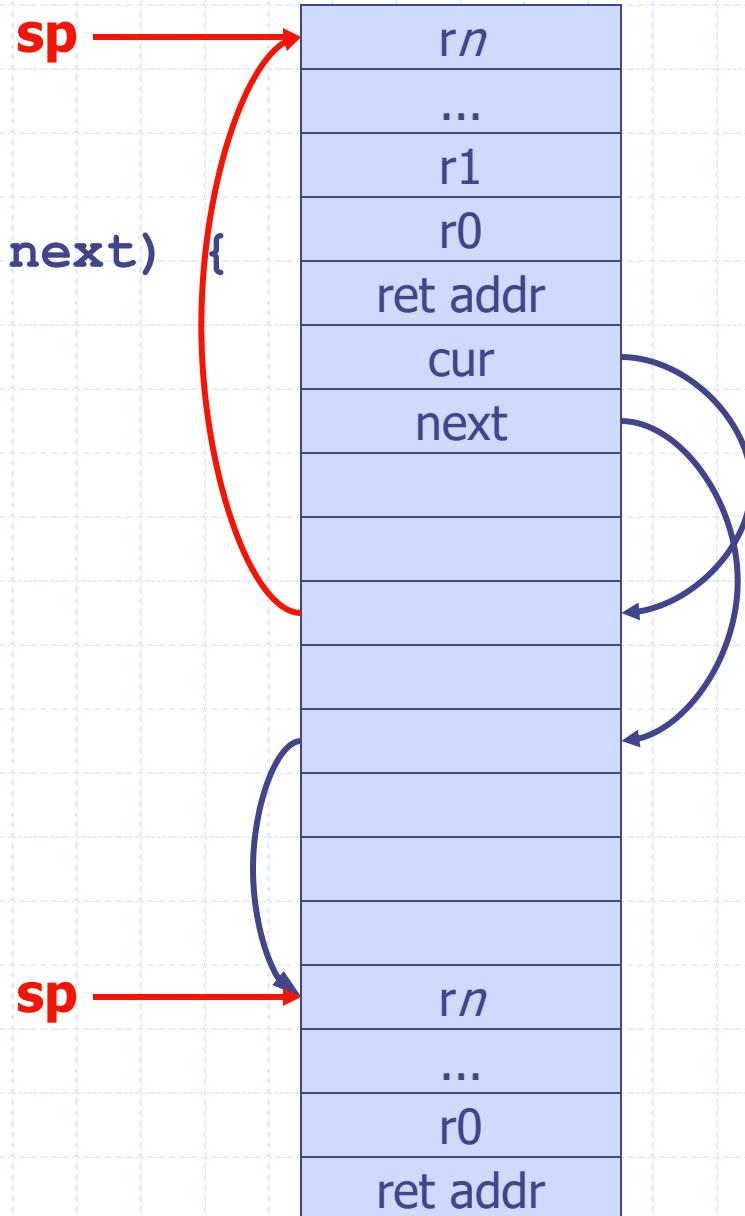
# Kontekst procesora

- ◆ Pitanje: kako preći sa izvršavanja jednog procesa na izvršavanje drugog procesa?
- ◆ Zahtev: sačuvati *sve što je potrebno za povratak izvršavanja napuštenog procesa*
- ◆ Proces: izvršavanje jednog *programa* nad datim *podacima*
- ◆ *Kontekst procesora (processor execution context)*: sve vrednosti iz procesorskih registara koje je potrebno sačuvati da bi se izvršavanje nastavilo od mesta napuštanja:
  - Mesto u programu na kome se stalo - PC
  - Podaci u procesoru – registri opšte namene
  - Lokalni podaci potprograma i “trag” izvršavanja – sve na steku – SP
- ◆ Prelazak sa izvršavanja jednog procesa na drugi – *promena konteksta (context switch)*:
  - sačuvati kontekst koji se napušta
  - povratiti kontekst na koji se prelazi

# Kontekst procesora

Jednostavna promena konteksta:

```
void yield (void* cur, void* next) {
asm {
    push r0
    push r1
    ...
    push rn
    mov r0,#cur[sp]
    mov [r0],sp
    mov r0,#next[sp]
    mov sp,[r0]
    pop rn
    ...
    pop r1
    pop r0
    pop pc ; return
}
}
```

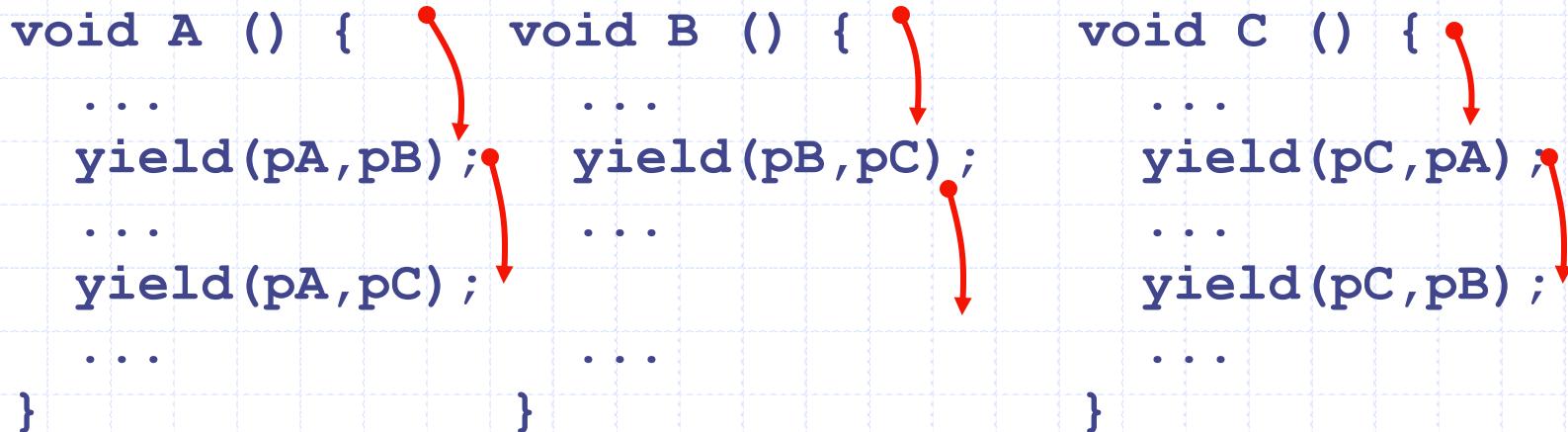


# Kontekst procesora

Sinhrona, eksplisitna promena konteksta –  
korutine (*coroutines*):

```
void* pA = ...;  
void* pB = ...;  
void* pC = ...;
```

```
void A () {    void B () {    void C () {  
    ...          ...          ...  
    yield(pA,pB);    yield(pB,pC);    yield(pC,pA);  
    ...          ...          ...  
    yield(pA,pC);    ...          yield(pC,pB);  
    ...          ...          ...  
    }          }          }  
}
```



Problem: kako uspostaviti polazni kontekst?

# Glava 5: Mehanizam prekida

---

Čemu prekidi?

Pojam prekida

Obrada prekida

Određivanje adrese prekidne rutine

Maskiranje prekida

Prioritiranje prekida

Izvori prekida

---

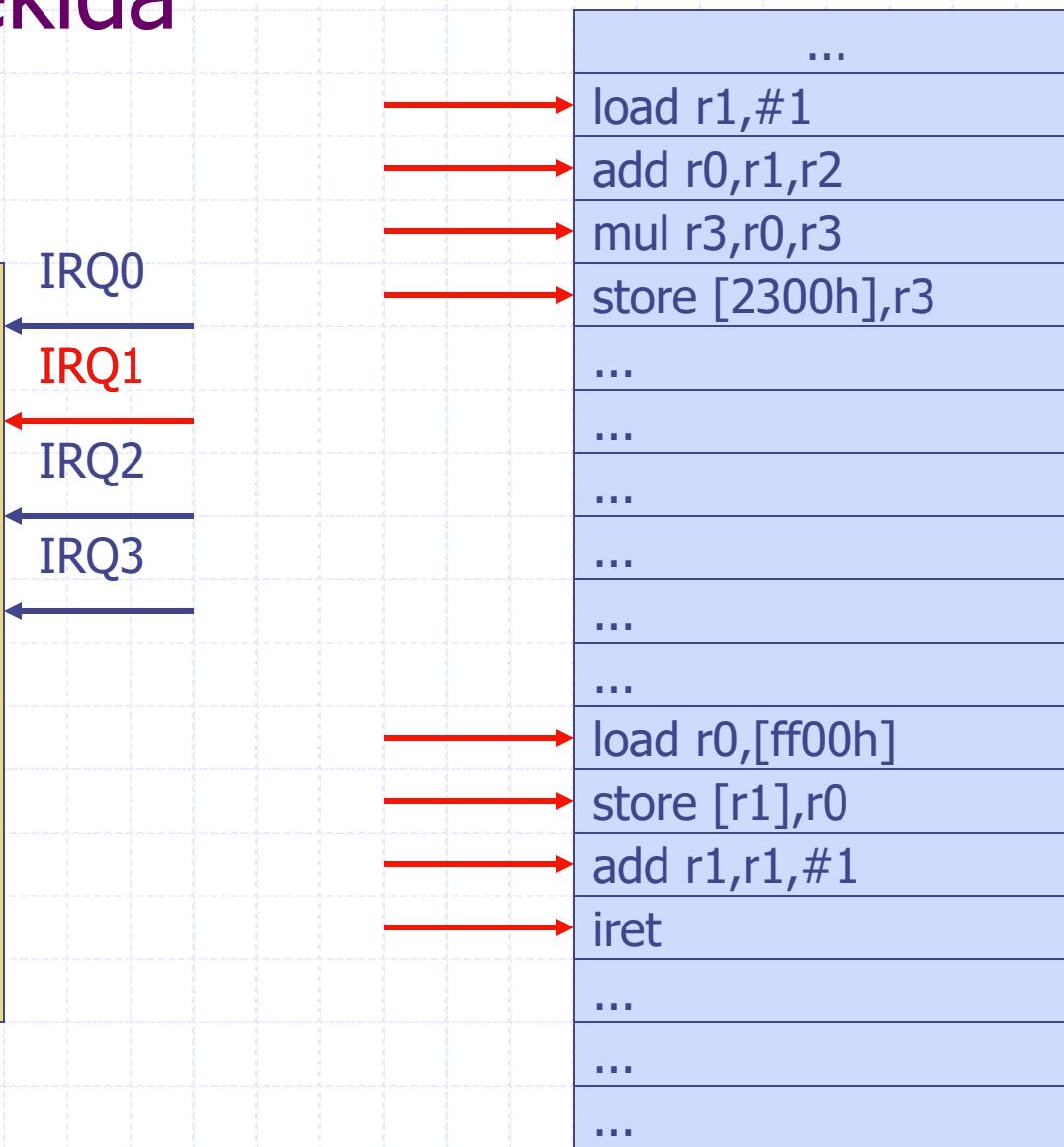
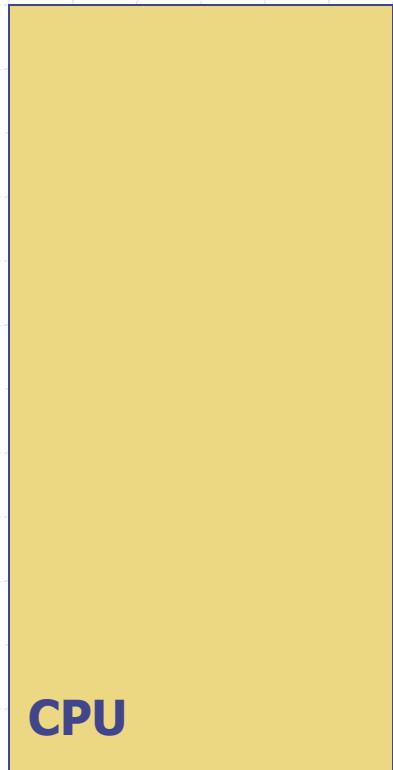
# Čemu prekidi?

- ◆ Kada bi procesor samo izvršavao program iz memorije, kako da ima informaciju da se nešto u okruženju desilo:
  - završena I/O operacija
  - isteklo vreme izvršavanja dodeljeno procesu
  - dogodio se događaj na koji treba reagovati (npr. taster, miš, senzor)
  - došao je trenutak aktivacije periodičnog procesa
  - itd.?
- ◆ Jedan način je da procesor po potrebi izvršava instrukcije kojima *ispituje* da li se to dogodilo i *čeka* da se dogodi ako nije
- ◆ Problem: neefikasnost – procesor izvršava “jalove” instrukcije čekajući na događaj, a mogao bi da radi neki drugi koristan posao

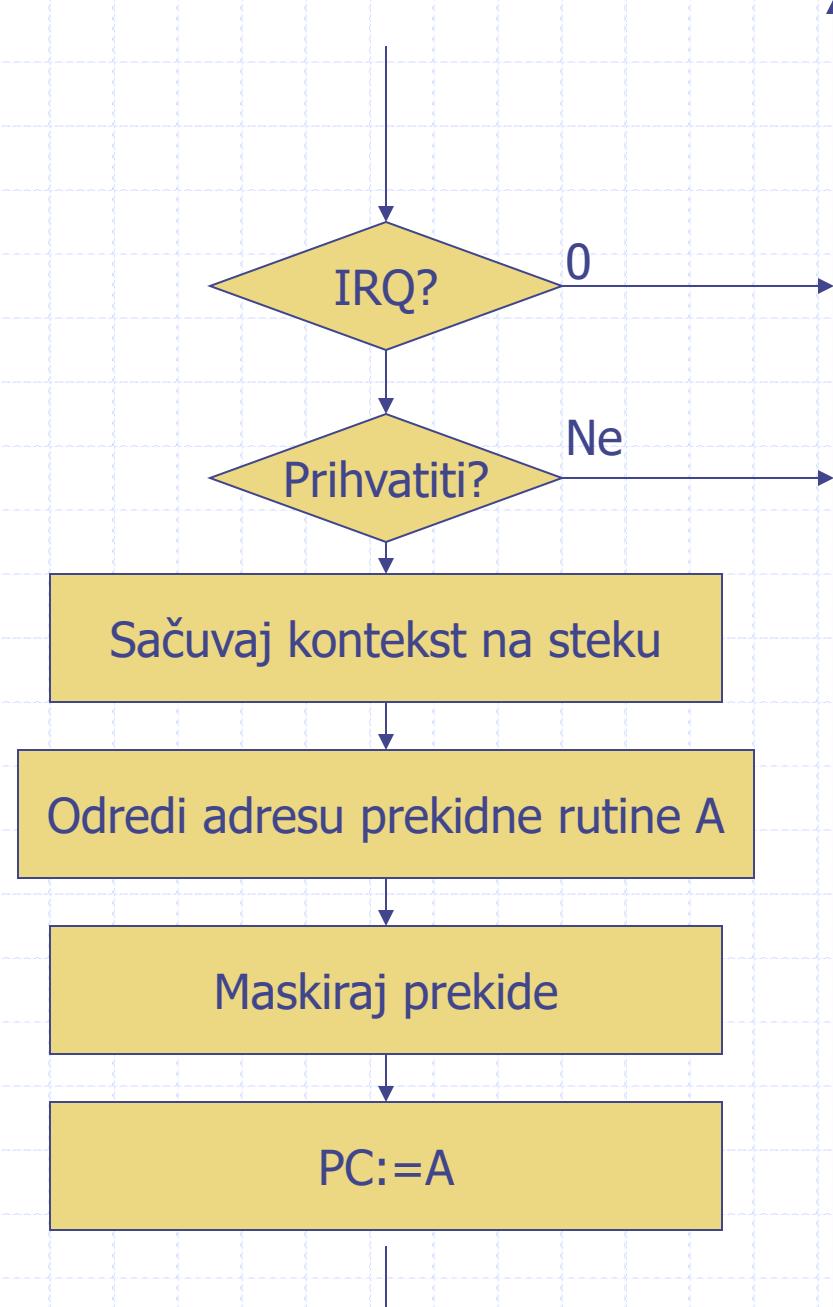
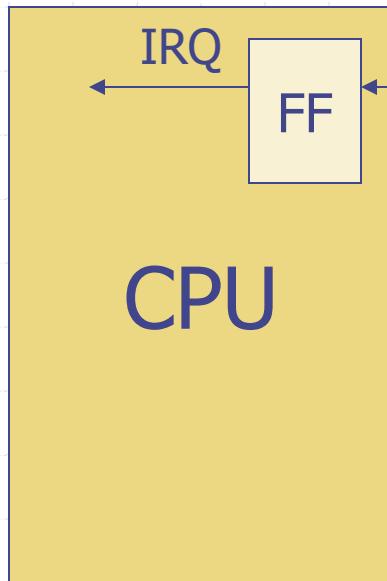
# Pojam prekida

- ◆ Ideja: procesor radi neki drugi posao, a događaj se signalizira posebnim HW signalom koji predstavlja *zahtev za prekid* (*interrupt request*)
- ◆ Kada stigne zahtev za prekid, procesor završava tekuću instrukciju, čuva kontekst na steku i prelazi na izvršavanje posebnog programa za obradu prekida – *prekidne rutine* (*interrupt routine*)
- ◆ Važno: procesor uvek završava izvršavanje tekuće instrukcije pre nego što pređe na obradu spoljašnjeg zahteva za prekid – izvršavanje instrukcije je atomično!
- ◆ Kada završi prekidnu rutinu, procesor se vraća na mesto gde je prekinuto izvršavanje, kao iz najobičnijeg potprograma

# Pojam prekida

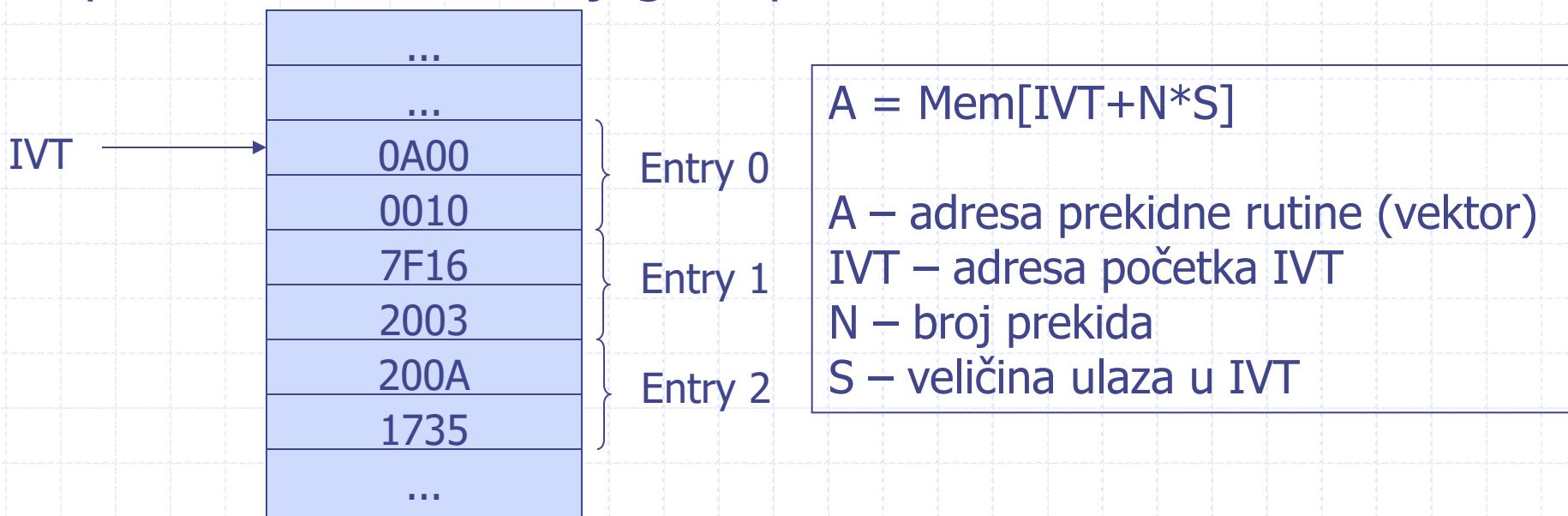


# Obrada prekida



# Određivanje adrese prekidne rutine

- ◆ Više načina, ali najčešći je putem *vektor tabele* (*interrupt vector table*, IVT)
- ◆ IVT je oblast u memoriji koja sadrži *vektore* – adrese prekidnih rutina. Svaki ulaz broj N u tabeli određuje adresu prekidne rutine za prekid sa brojem N – IVT preslikava broj prekida N u adresu njegove prekidne rutine



# Određivanje adrese prekidne rutine

## ◆ Kako se određuje broj prekida N?

- Fiksno: svakom signalu zahteva za prekid dodeljen je hardverski definisan broj N
- Poseban registar periferijskog uređaja sadrži broj prekida N dodeljen toj periferiji. Kada prihvati prekid sa date linije, procesor očitava taj registar da bi dobio N. Broj se može upisati u registar periferije programski (tipično pri inicijalizaciji sistema)

# Maskiranje prekida

- ◆ Moguće je programskim putem zabraniti spoljašnje prekide – “maskiranje” (*interrupt masking*)
- ◆ Poseban programski dostupan jednobitni registar dozvoljava ili ne dozvoljava prihvatanje spoljašnjih prekida
- ◆ Ovim registrom obično manipulišu posebne instrukcije:  
**INTE ; Interrupt Enable, STI – Set Interrupt**  
**INTD ; Interrupt Disable, CLI – Clear Interrupt**
- ◆ Neki procesori omogućuju i selektivno maskiranje prekida sa pojedinačnih linija: poseban programski dostupan registar maske (*interrupt mask register, IMR*) svakim svojim razredom omogućava ili ne prekid sa određene linije
- ◆ Prekid se prihvata samo ako nije selektivno ili globalno maskiran
- ◆ Prilikom prihvatanja prekida, prekidi se maskiraju, da se ne bi ugnezđivali – prekidna rutina je podrazumevano (ali ne uvek) atomična – atomičnost obezbeđuje HW

# Prioritiranje prekida

- ◆ Neki procesori omogućavaju *prioritiranje prekida* (*interrupt prioritizing*): svakom prekidu dodeljen je *prioritet (priority)*
- ◆ Prekid se prihvata samo ako se trenutno ne izvršava prekidna rutina za prekid višeg prioriteta
- ◆ Prema tome, uslov za prihvatanje prekida:
  - da nije maskiran (globalno ili selektivno)
  - da nije u toku prekidna rutina višeg prioriteta

# Izvori prekida

◆ Izvori prekida: hardverski ili softverski

◆ Hardverski prekidi:

- spoljašnji (*external*) - dolaze po signalima koji ulaze u CPU od strane:
  - ◆ I/O uređaja (tipično maskirajući): završetak rada, greška u operaciji
  - ◆ vremenskih brojača (*timer*), periodično ili u određeno vreme (tipično maskirajući)
  - ◆ uređaja za nadzor ispravnosti rada hardvera (napon napajanja, greška u memoriji itd., tipično nemaskirajući)
- unutrašnji (*internal*) – dolaze kao posledica greške u izvršavanju same instrukcije – *izuzeci* (*exception*):
  - ◆ nedozvoljen kod instrukcije (*illegal instruction code*)
  - ◆ nedozvoljen način adresiranja (*illegal addressing mode*)
  - ◆ prekoračenje (*overflow*), deljenje nulom (*division by zero*) ili druga aritmetička greška
  - ◆ stranična greška (*page fault*) – adresirana lokacija virtualne memorije nije trenutno u fizičkoj memoriji

# Izvori prekida

- ◆ Softverski prekidi (zamke, *trap*) - posebna instrukcija (**TRAP**, **INT**) sa parametrom N rezultuje istim ponašanjem kao da je stigao spoljašnji zahtev za prekid sa brojem N:

**TRAP 1Ch**

Prelazi se na izvršavanje prekidne rutine sa vektorom iz IVT ulaza N

- ◆ Upotreba: pozivi usluga (servisa, potprograma) van adresnog prostora programa pozivaoca – tipično usluga OS (*sistemske pozive*)
- ◆ Prenos parametara:
  - preko registara procesora
  - neki register procesora sadrži adresu bloka podataka u memoriji gde se nalaze parametri

# Glava 6: Ulaz/Izlaz

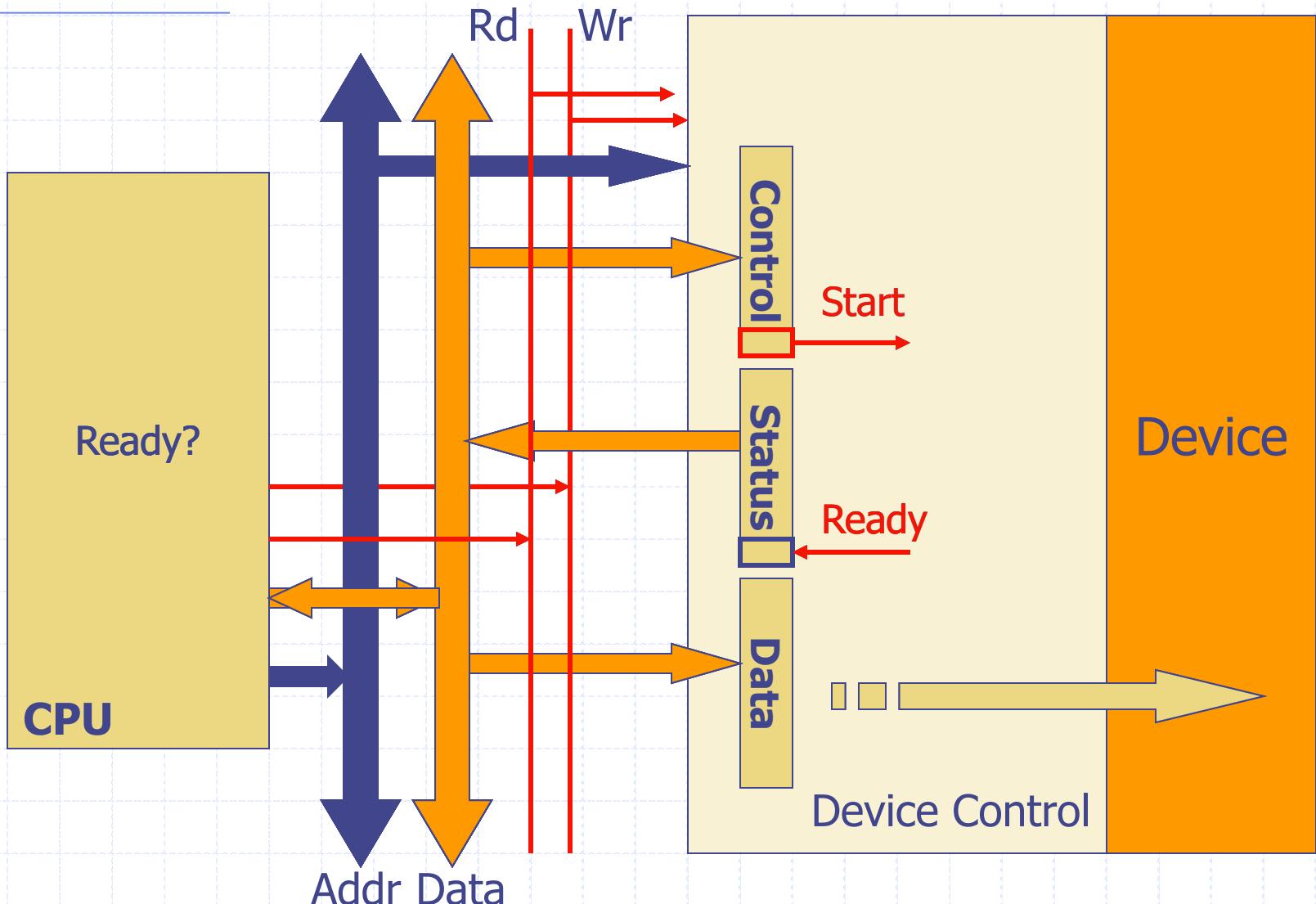
Generički model I/O uređaja

Uposleno čekanje

Korišćenje prekida

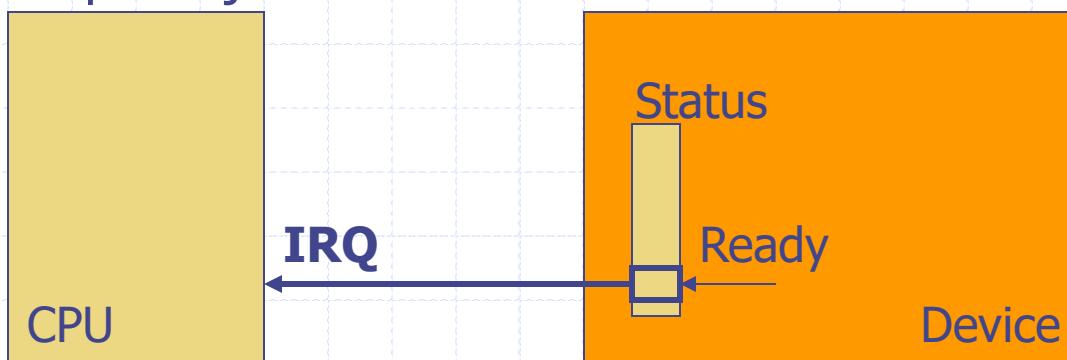
Direktan pristup memoriji (DMA)

# Generički model I/O uređaja



# Generički model I/O uređaja

- ◆ Problem: Kako CPU da sazna da je I/O uređaj završio zadatu operaciju i spreman je za sledeću?
- ◆ Dva načina:
  - Uposleno čekanje (*busy waiting*) ili prozivanje (*polling*): CPU očitava statusni registar uređaja, ispituje bit spremnosti i ponavlja to isto sve dok bit spremnosti ne bude postavljen
  - Mehanizam prekida (*interrupt*): signal sa bita spremnosti je povezan na ulaz za prekid procesora; kada uređaj postane spreman, procesor dobija prekid i u prekidnoj rutini zadaje novu operaciju

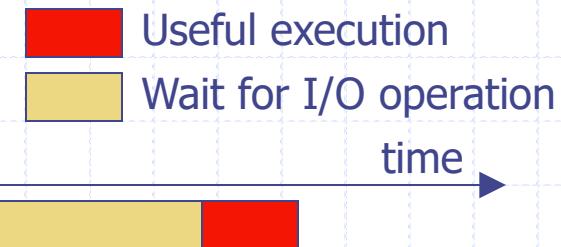


# Uposleno čekanje

```
Init_Transfer;  
loop  
    Wait_for_Ready;  
    Transfer_Data;        Wait: load r0, [Status]  
    Prepare_for_Next;    and r0,r0,#100..0b; Ready?  
    if last then exit;  
    jz Wait  
end loop;  
Stop_Transfer;  
  
load r1,#BlockAddr  
load r2,#Count  
store [Ctrl1],#00..1b; Start  
  
load r0, [Data]  
store [r1],r0  
  
inc r1  
dec r2  
jnz Wait  
  
store [Ctrl1],#0; Stop
```

# Korišćenje prekida

- ◆ Problem uposlenog čekanja: dok čeka na završetak I/O operacije, CPU izvršava "jalove" instrukcije u petlji – neiskorišćen za druge poslove



- ◆ Korišćenje prekida:

- ◆ CPU zada I/O operaciju, a onda izvršava neki drugi posao, nezavisan od zadate I/O operacije
- ◆ kada završi operaciju i bude spreman za sledeću, uređaj generiše prekid
- ◆ u prekidnoj rutini CPU zadaje novu operaciju i vraća se na prekinuti posao

# Korišćenje prekida

Main:

```
Init_Transfer;  
Do_something_else;  
Wait_for_end_of_transfer;
```

Interrupt\_Routine:

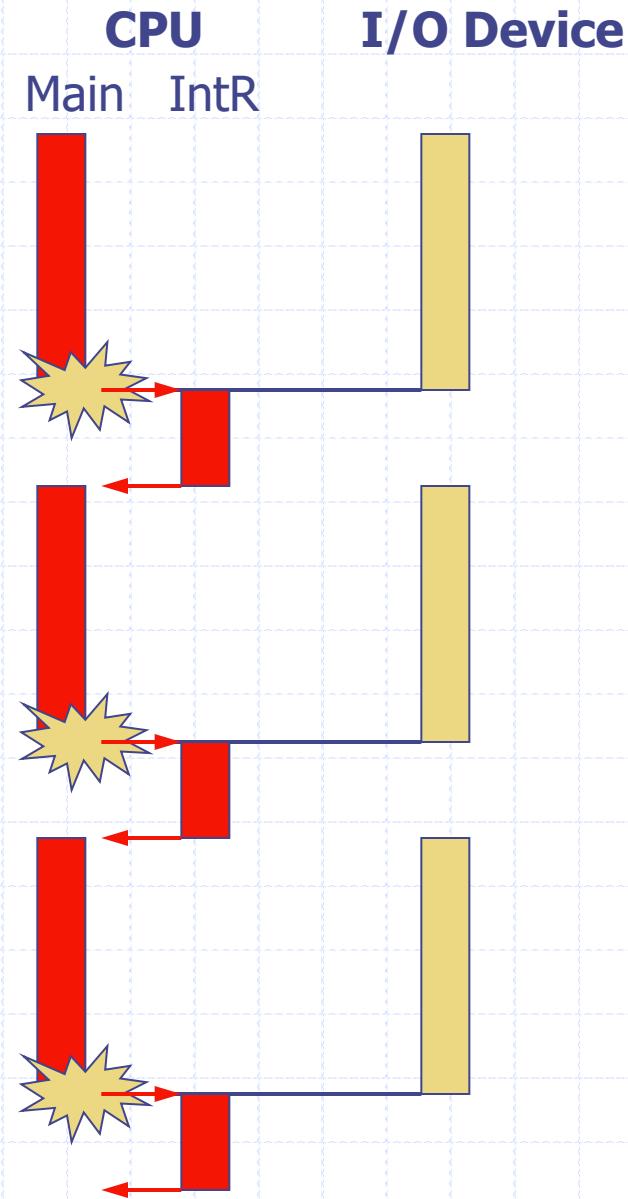
```
Transfer_Data;  
Prepare_for_Next;  
if last then Stop_Transfer;
```

# Korišćenje prekida

Main:

```
load r1,#BlockAddr  
load r2,#Count  
load r3,#0; End Flag  
store [Ctrl],#00..1b; Start  
...  
Wait: cmp r3,#1; End of transfer?  
jnz Wait  
...
```

```
IntR: load r0,[r1]  
store [Data],r0  
inc r1  
dec r2  
jnz Ret  
load r3,#1  
store [Ctrl],#0; Stop  
Ret:iret ; Interrupt return
```



# Direktan pristup memoriji (DMA)

- ◆ DMA (*Direct Memory Access*) kontroler je poseban uređaj specijalizovan da sam obavlja transfer bloka podataka sa ili na periferijski uređaj, na zahtev procesora
- ◆ Procesor samo zada parametre transfera (adresu i veličinu bloka) i potom radi neki drugi posao, nezavisan od transfera
- ◆ DMA sam interaguje sa I/O uređajem na isti način kao što bi to procesor radio (*polling* ili korišćenjem signala *ready-hand-shaking*) i prenosi podatke
- ◆ Kada završi prenos, DMA postavlja svoj indikator završetka u statusnom registru
- ◆ Kako da procesor zna da je DMA završio prenos?  
Isto kao i ranije:
  - ispitivanjem indikatora u statusnom registru DMA (*busy waiting*)
  - signal završetka prenosa generiše procesoru zahtev za prekid

# Direktan pristup memoriji (DMA)

Main:

```
Init_Transfer;  
Do_something_else;  
Wait_for_end_of_trans;
```

Interrupt\_Routine:

```
Stop_Transfer;
```

Main:

```
    store [DMAAddr], #BlockAddr  
    store [DMACount], #Count  
    load  r1, #0; End Flag  
    store [IOCtrl], #00..1b; Start  
    store [DMACtrl], #10..0b; Start
```

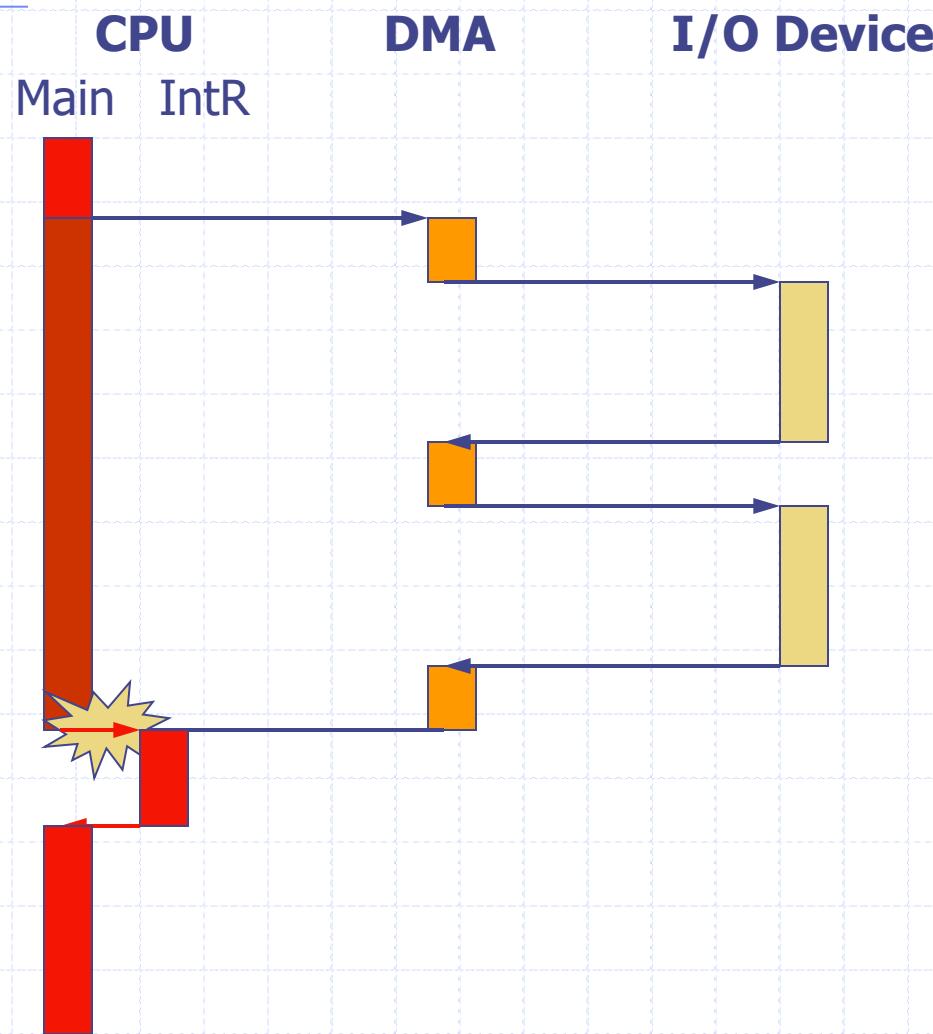
...

```
Wait: cmp r1, #1; End of transfer?  
      jnz  Wait
```

...

```
IntR: store [IOCtrl], #0; Stop  
      store [DMACtrl], #0; Stop  
      load  r1, #1; End of transfer  
      iret ; Interrupt return
```

# Direktni pristup memoriji (DMA)



# Glava 7: Virtuelna memorija

---

Pojam virtuelne memorije

Preslikavanje adresa

Stranična organizacija

Segmentna organizacija

Segmentno-stranična organizacija

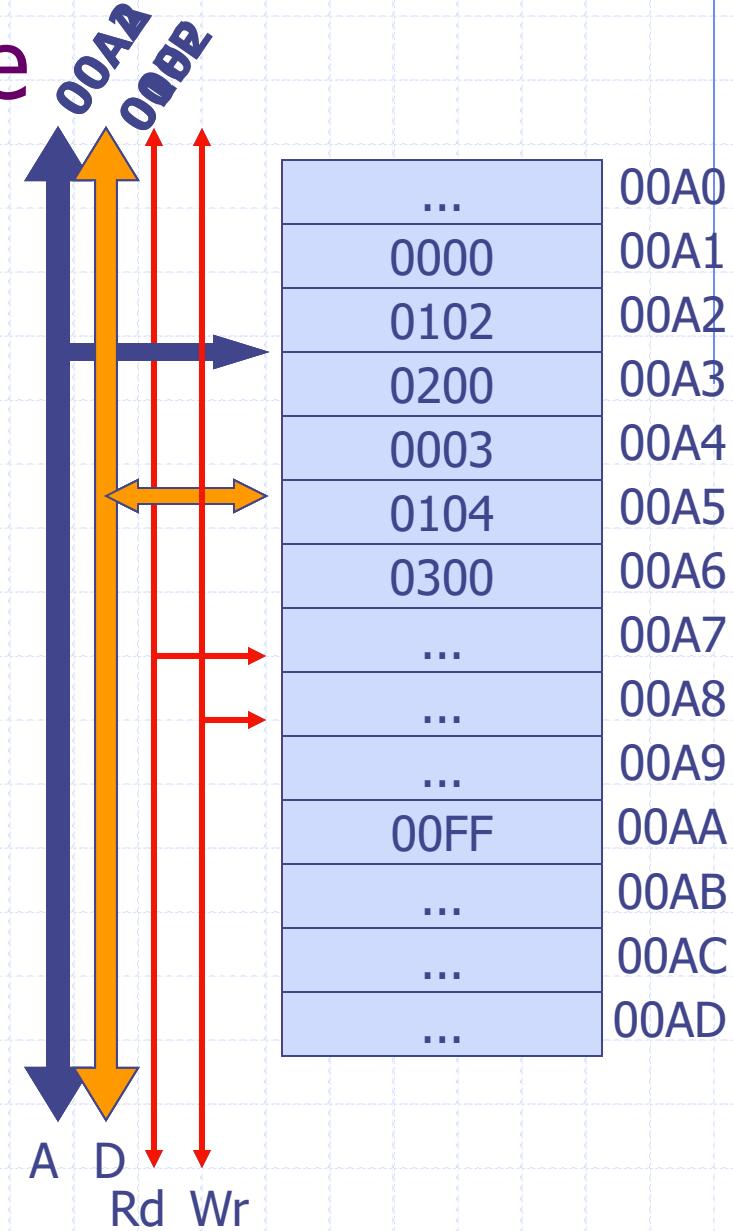
TLB

Podela odgovornosti između HW i OS

---

# Pojam virtuelne memorije

- ◆ Fizička memorija računara: linearno uređen niz ćelija sa mogućnošću *direktnog pristupa (direct access)*
- ◆ Direktan pristup: svaka ćelija ima svoju *adresu (address)* preko koje joj se može pristupiti
- ◆ RAM (*Random Access Memory*): memorija sa mogućnošću čitanja i upisa; gubi sadržaj gubitkom napajanja
- ◆ ROM (*Read Only Memory*): memorija samo sa mogućnošću čitanja; obično čuva sadržaj po gubitku napajanja; služi za smeštanje osnovnog sistemskog programa za "podizanje" sistema – učitavanje OS (*bootstrap program*)



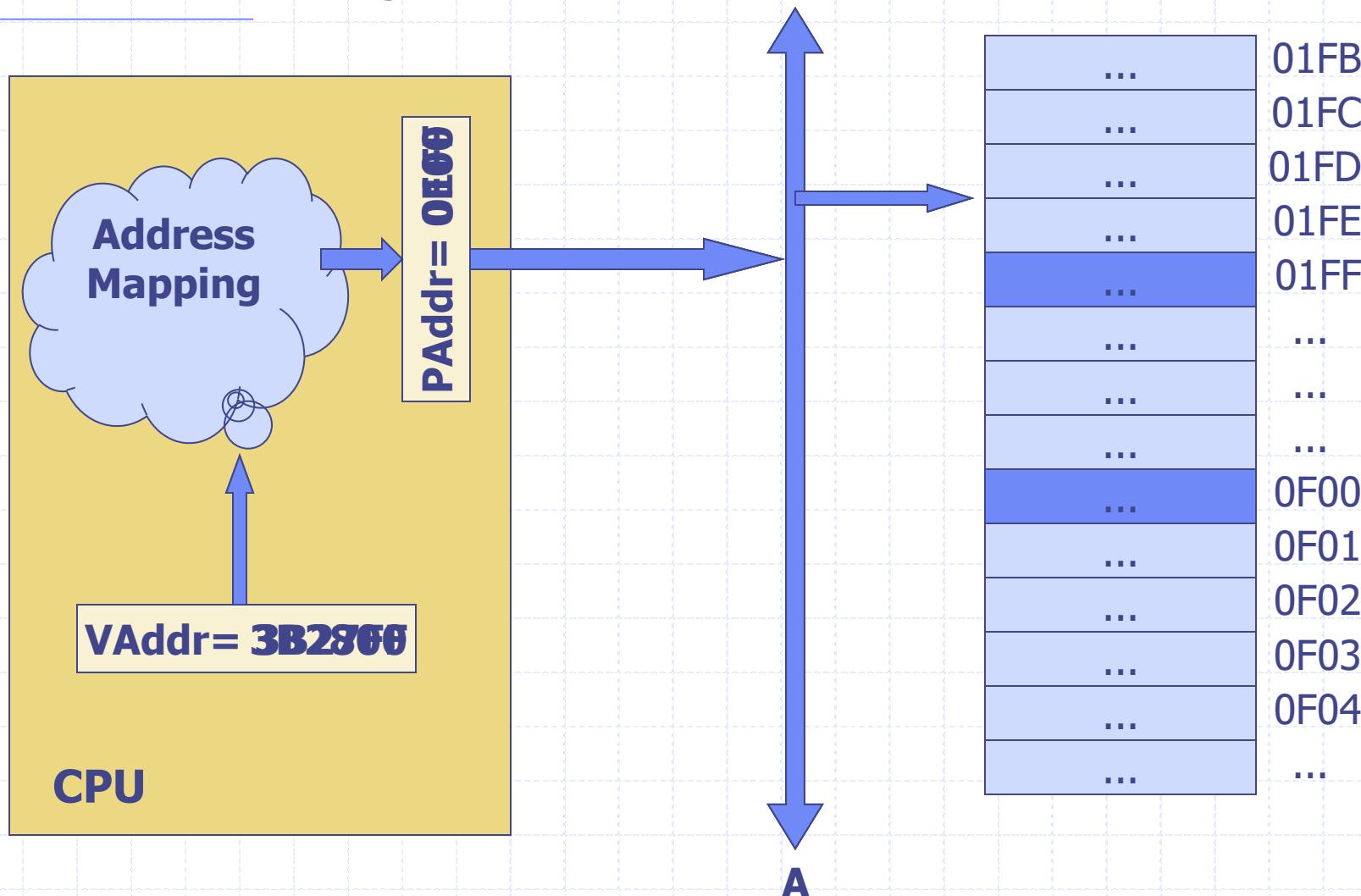
# Pojam virtuelne memorije

- ◆ Problem: potrebno je da više procesa bude raspoređeno po fizičkoj memoriji, možda i delimično učitanih sa diska, tako da se mogu i premeštati (relocirati).
  - Kako omogućiti izvršavanje programa koji nije u potpunosti učitan u memoriju?
  - Kako obezbediti relokabilnost procesa, tj. njegovog sadržaja u memoriji, a da to bude transparentno za izvršavanje programa?
- ◆ Ideja: sve adrese koje program koristi u adresiranju instrukcija i podataka jesu samo *logičke, virtuelne* adrese; poseban harver procesora preslikava traženu virtualnu adresu u fizičku adresu u memoriji, na osnovu podataka OS o rasporedu delova procesa u fizičkoj memoriji

# Pojam virtuelne memorije

- ◆ (*Virtuelni adresni prostor (address space) procesa*: skup virtuelnih (logičkih) adresa dostupnih programu za adresiranje)
- ◆ (*Fizički adresni prostor procesora*: skup svih adresa koje procesor može generisati na adresnoj magistrali)
- ◆ Fizička memorija: stvarno instalirana količina memorijskog prostora; može biti manja od fizičkog adresnog prostora)
- ◆ Posledica: virtuelni adresni prostor može biti veći, manji, ili čak jednak fizičkom adresnom prostoru)

# Preslikavanje adresa



# Preslikavanje adresa

- ◆ Zadatak hardvera za preslikavanje adresa:  
Virtuelnu adresu (VA) preslikati u fizičku adresu (PA).  
Kako?
- ◆ Tabele preslikavanja (*map table*, MT) VA u PA za svaki proces nalaze se u memoriji. Njihov sadržaj održava OS učitavajući u memoriju i izbacujući iz nje delove procesa
- ◆ Jedan ulaz u MT daje podatke (opisuje) o jednom delu virtuelnog adresnog prostora datog procesa – *deskriptor* (*descriptor*)
- ◆ Deskriptor sadrži bar sledeće informacije:
  - indikator da li je dati deo virtuelnog adresnog prostora trenutno u fizičkoj memoriji
  - ako jeste, koja je njegova PA
  - ako nije, gde je smešten na disku (adresa sektora na disku)

# Preslikavanje adresa

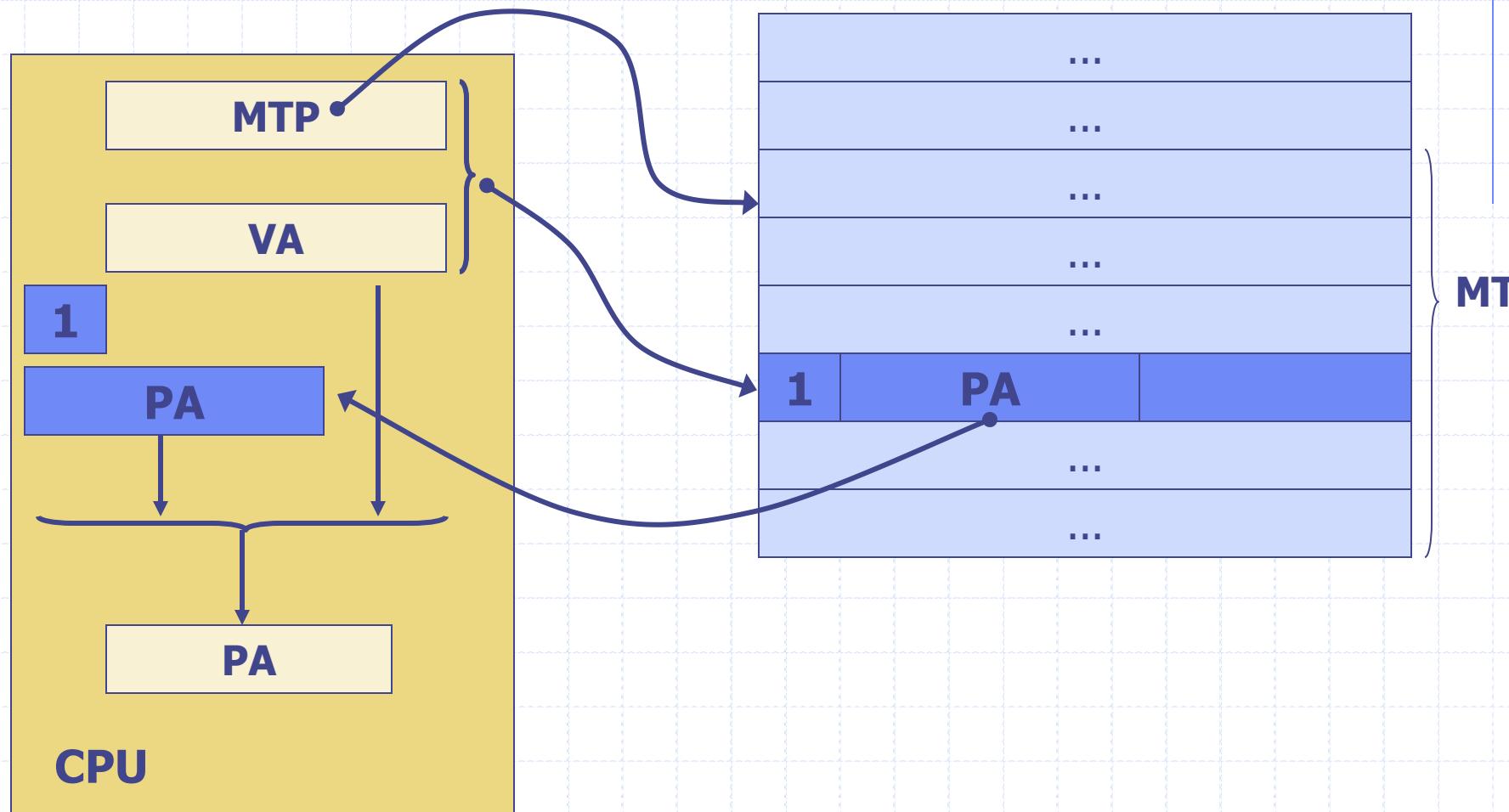
## ◆ Postupak:

- Na osnovu VA i početne (fizičke) adrese tabele preslikavanja (*map table pointer*, MTP) koja je upisana u neki specijalizovani registar procesora od strane OS kada je procesor dodelio datom procesu, odrediti broj ulaza i adresu deskriptora u MT
- Dovući deskriptor u procesor
- Ako VA jeste u fizičkoj memoriji, iz deskriptora se dobija PA
- Ako VA nije u fizičkoj memoriji, generiše se interni prekid (*page fault*); OS preuzima kontrolu i ima zadatak da dovuče dati deo sa diska u fizičku memoriju, eventualno izbacujući neki drugi deo istog ili drugog procesa i da ažurira MT

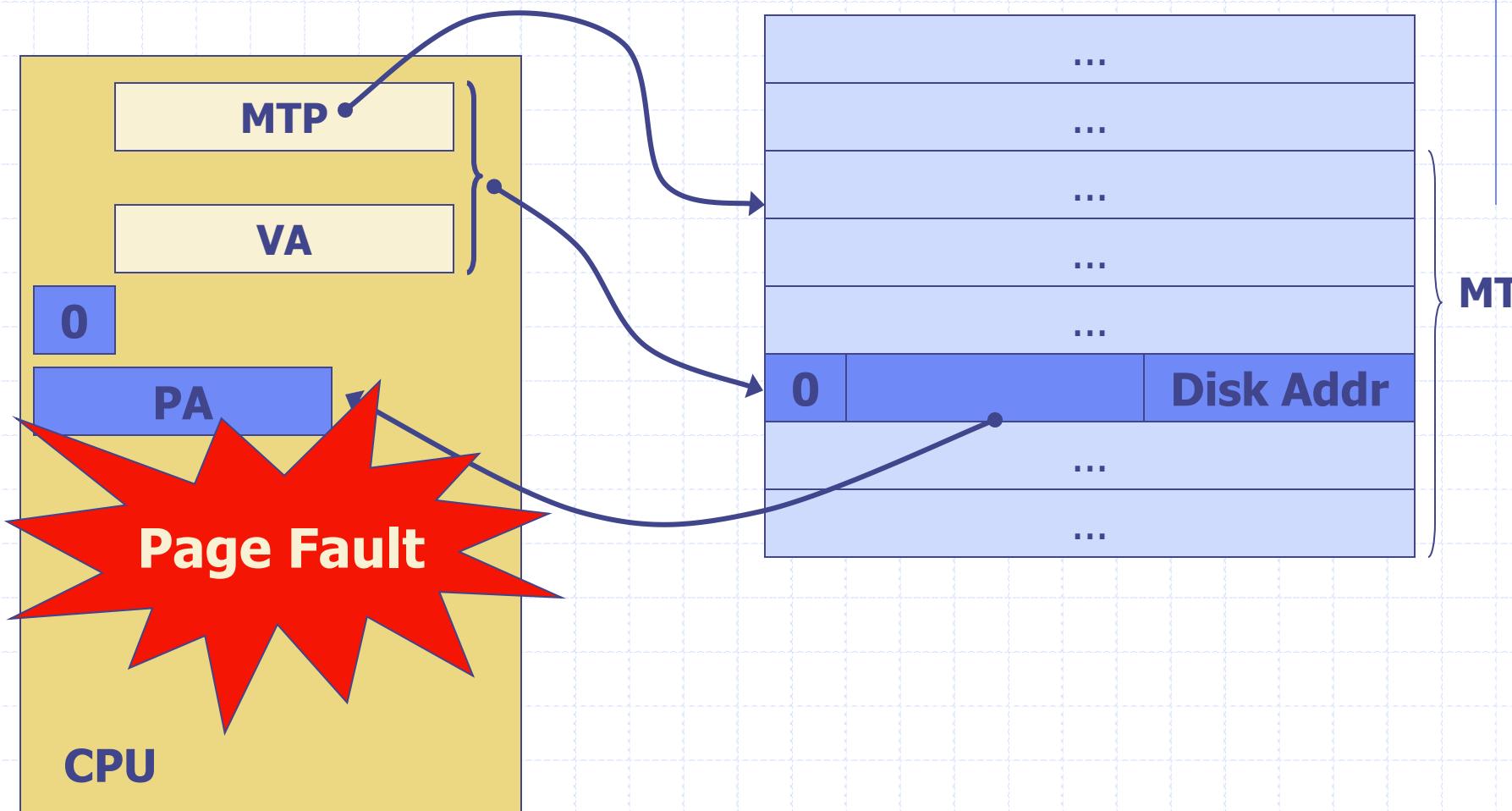
## ◆ Pažnja:

- Ovo preslikavanje vrši se pri *svakom* adresiranju tokom izvršavanja jedne instrukcije, i za dohvatanje instrukcije i za pristup podacima
- Tokom samog dohvatanja deskriptora nema preslikavanja adresa – adresiranje unutar MT je uvek fizičkim adresama; MT se nalaze na nekom određenom mestu u fizičkoj memoriji, pod kontrolom OS

# Preslikavanje adresa



# Preslikavanje adresa



# Preslikavanje adresa

- ◆ Prilikom izuzetka tipa stranične greške (*page fault*), OS treba da uradi sledeće:
  - pronaći slobodno mesto u fizičkoj memoriji za smeštanje traženog dela virtuelne memorije
  - ukoliko nema slobodnog mesta u fizičkoj memoriji, izbaciti neki deo iz fizičke memorije na disk i ažurirati njegov deskriptor
  - učitati dati deo sa diska i ažurirati njegov deskriptor (indikator i PA)
- ◆ Prilikom povratka konteksta prekinutog procesa, potrebno je ponoviti preslikavanje – ovoga puta uspešno (po pravilu)
- ◆ Pažnja: izvršavanje se vraća na *istu* prekinutu instrukciju, a ne na narednu!
- ◆ Pažnja: izvršavanje je prekinuto u toku izvršavanja instrukcije, pa je odgovornost HW procesora da obezbedi da se ista instrukcija može izvršiti ispočetka! Kako?

# Preslikavanje adresa

## ◆ Posledice:

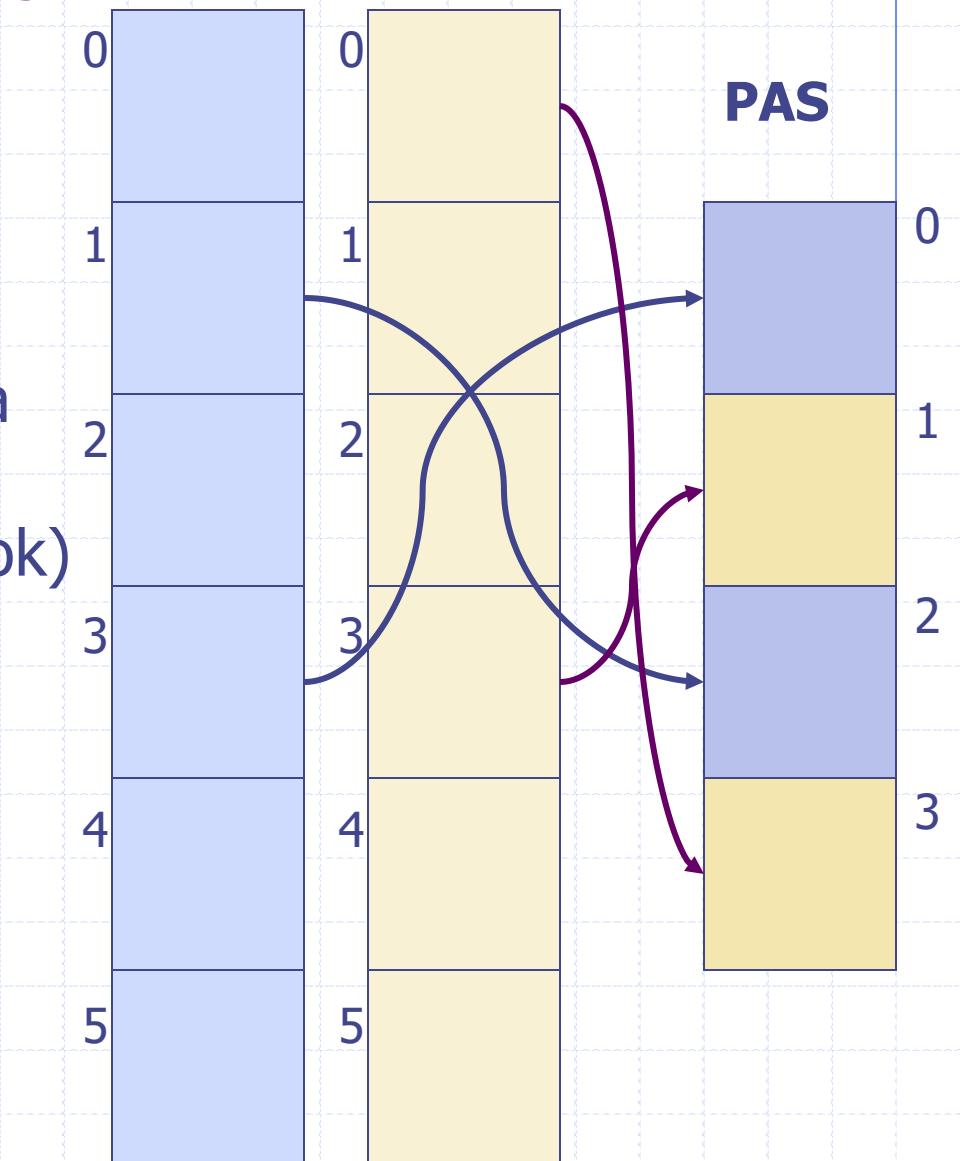
- potrebno je zaštiti oblast memorije u kojoj su MT od pogrešnog ili zlonamernog pristupa od strane korisničkih programa
- potrebno je obezbediti dva režima rada procesora:
  - ◆ sistemski: potencijalno nema preslikavanja adresa, ima pravo pristupa do sistemskih delova memorije, ima pravo izvršavanja nekih posebnih instrukcija
  - ◆ korisnički: uvek se preslikavaju adrese (pa zato i nema pravo pristupa do sistemskih delova memorije), nema pravo izvršavanja nekih posebnih instrukcija

# Stranično preslikavanje $VAS - P1$ $VAS - P2$

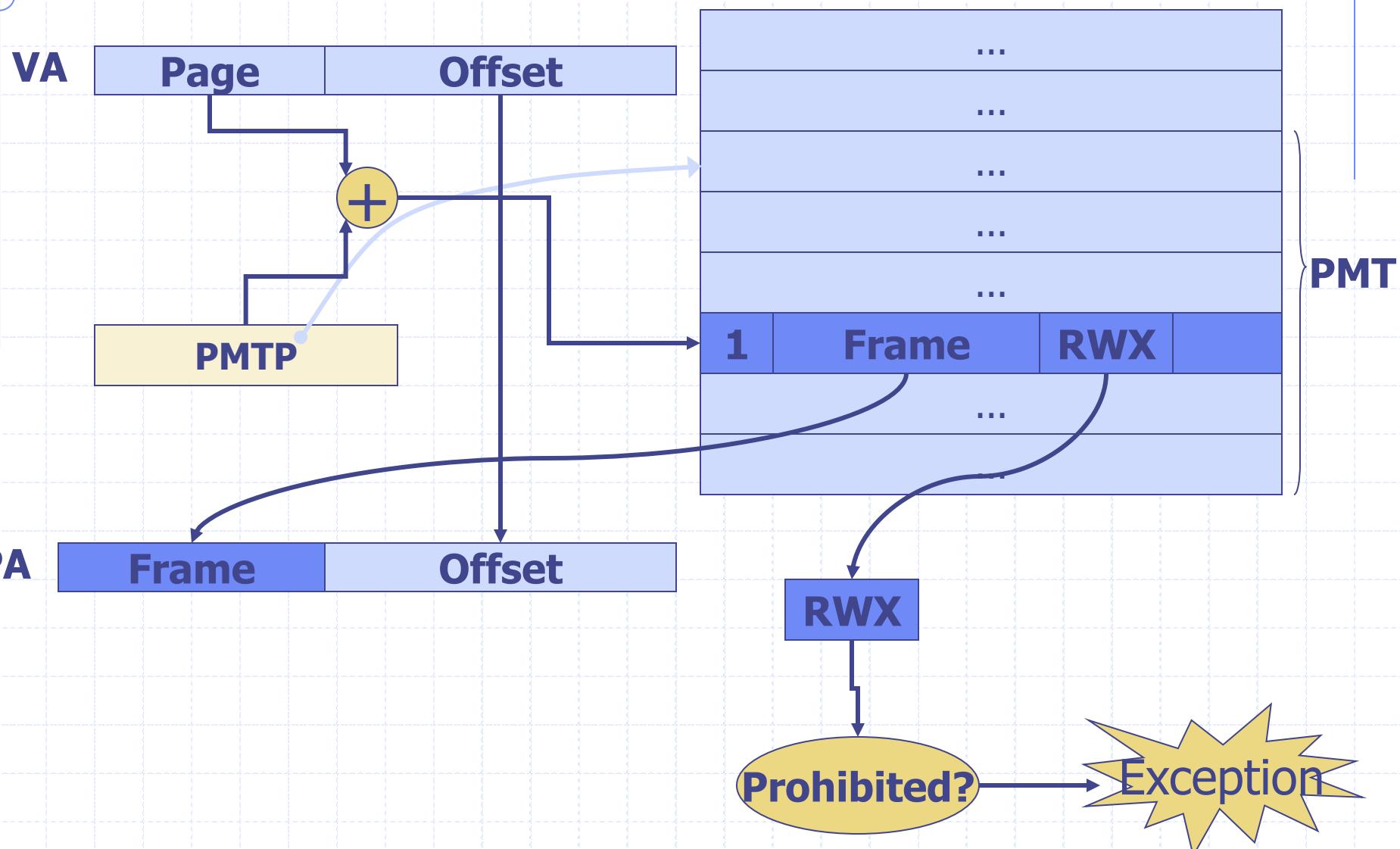
- ◆ Fizička i virtuelna memorija podeljene su na blokove iste veličine – *stranice (page)*
- ◆ Jedna stranica virtuelnog adresnog prostora preslikava se u jednu stranicu fizičkog adresnog prostora (okvir, blok)

## PMT - P1

0	...	...
1	2	...
0	...	...
1	0	...
0	...	...
0	...	...



# Stranično preslikavanje



# Stranično preslikavanje

## ◆ Prednosti:

- relativno jednostavan hardver, pravilna arhitektura, jednostavno i efikasno preslikavanje (u odnosu na ostale tehnike preslikavanja)
- nema eksterne fragmentacije (kao kod segmentne organizacije)
- jednostavan zadatak za OS u pogledu alokacije prostora – nema potrebe za "upasivanjem" bloka memorije

## ◆ Nedostaci:

- stranice nisu logičke celine
- zbog toga nije sasvim prirodno uvoditi zaštitu pristupa na nivou stranice (mada je moguće)
- interna fragmentacija



# Segmentno preslikavanje

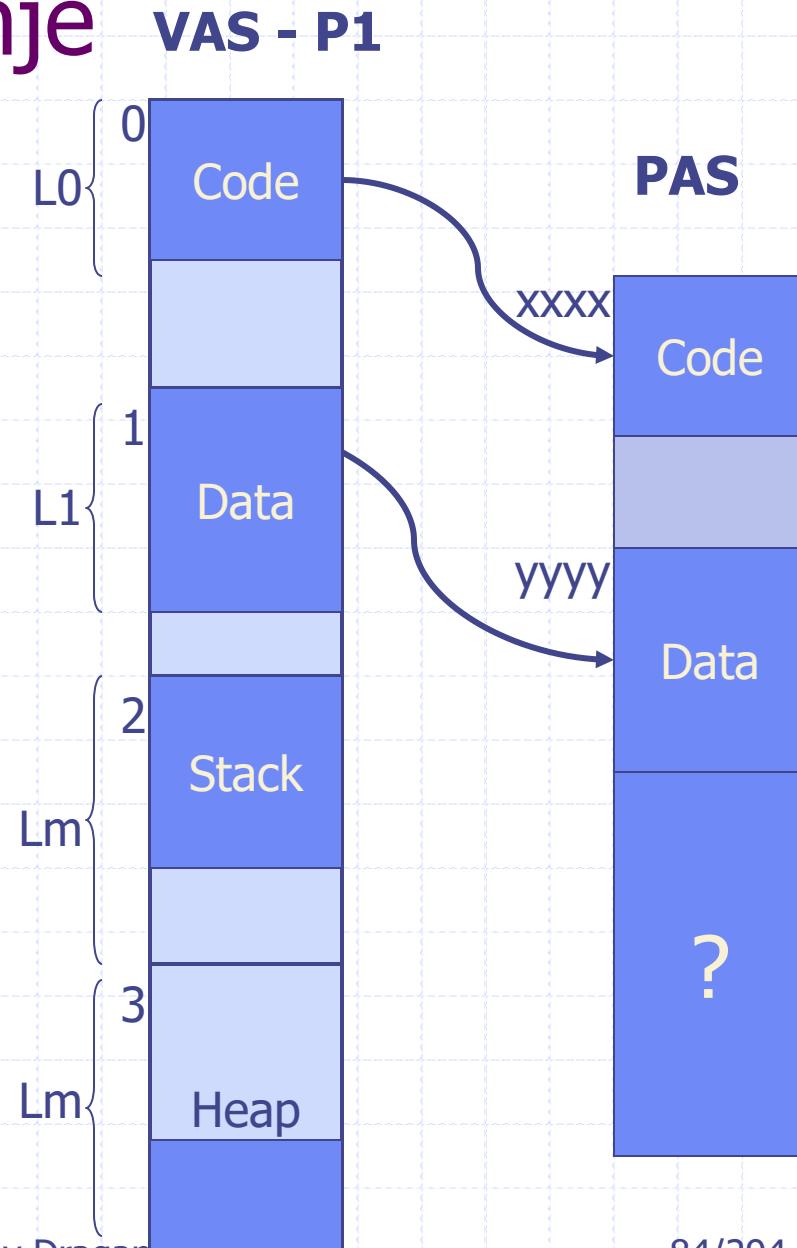
VAS - P1

- ◆ Virtuelna memorija je podeljena na logičke celine različite stvarne veličine - *segmente (segment)*, ali iste *maksimalne veličine*, npr. kod, statički podaci, stek, dinamička memorija itd.
- ◆ OS pronalazi prostor potrebne veličine za smeštanje segmenta

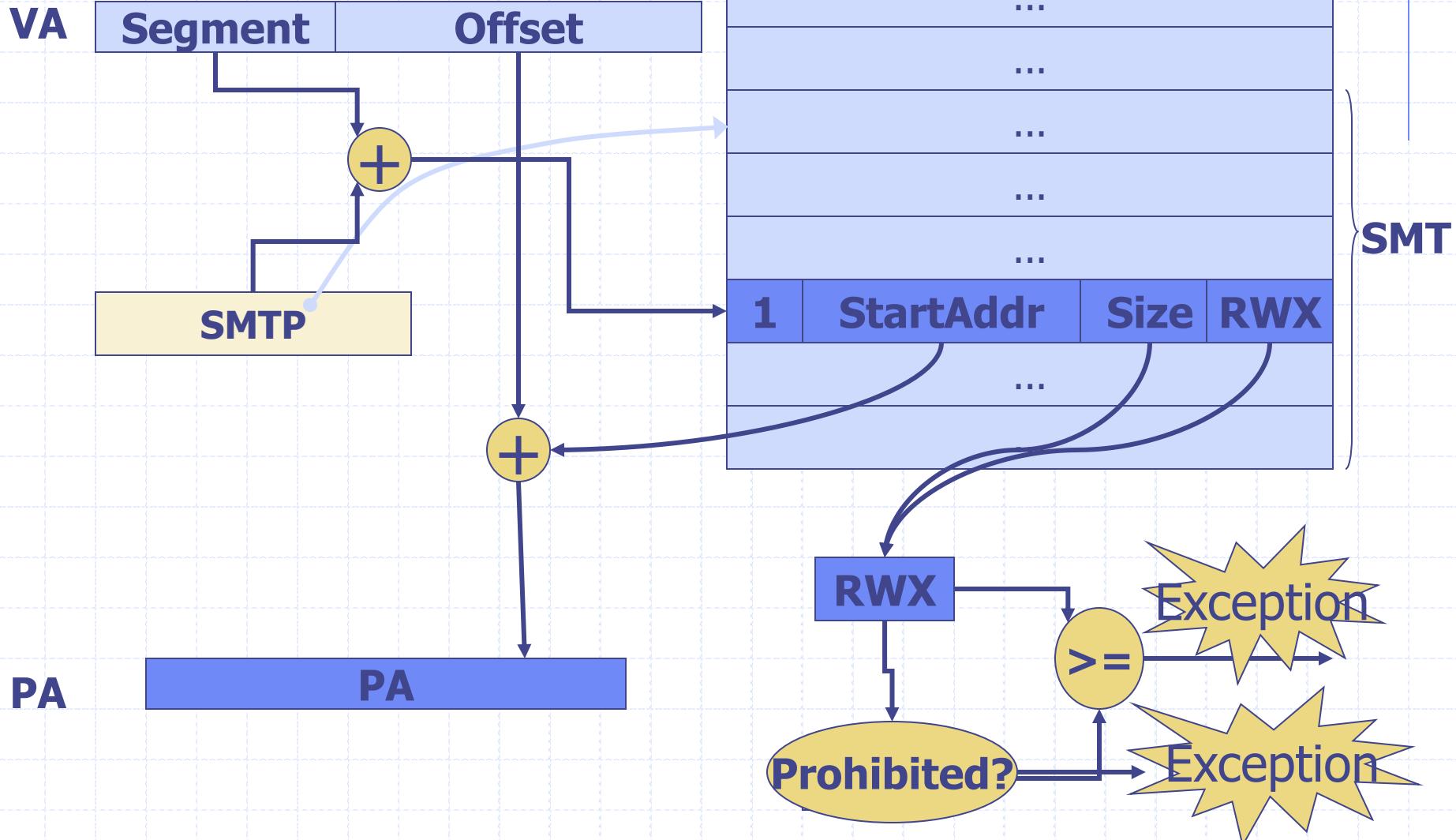
SMT - P1

Size

1	xxxx	L0	...
1	yyyy	L1	...
0	...	Lm	...
0	...	Lm	...



# Segmentno preslikavanje



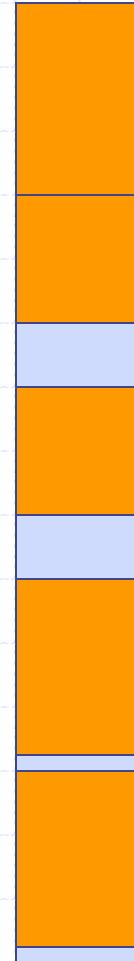
# Segmentno preslikavanje

## ◆ Prednosti:

- segmenti su logičke celine, pa je prirodno uvoditi zaštitu pristupa segmentu i prekoračenja veličine; npr. *code segment* je *read-only*
- nema interne fragmentacije

## ◆ Nedostaci:

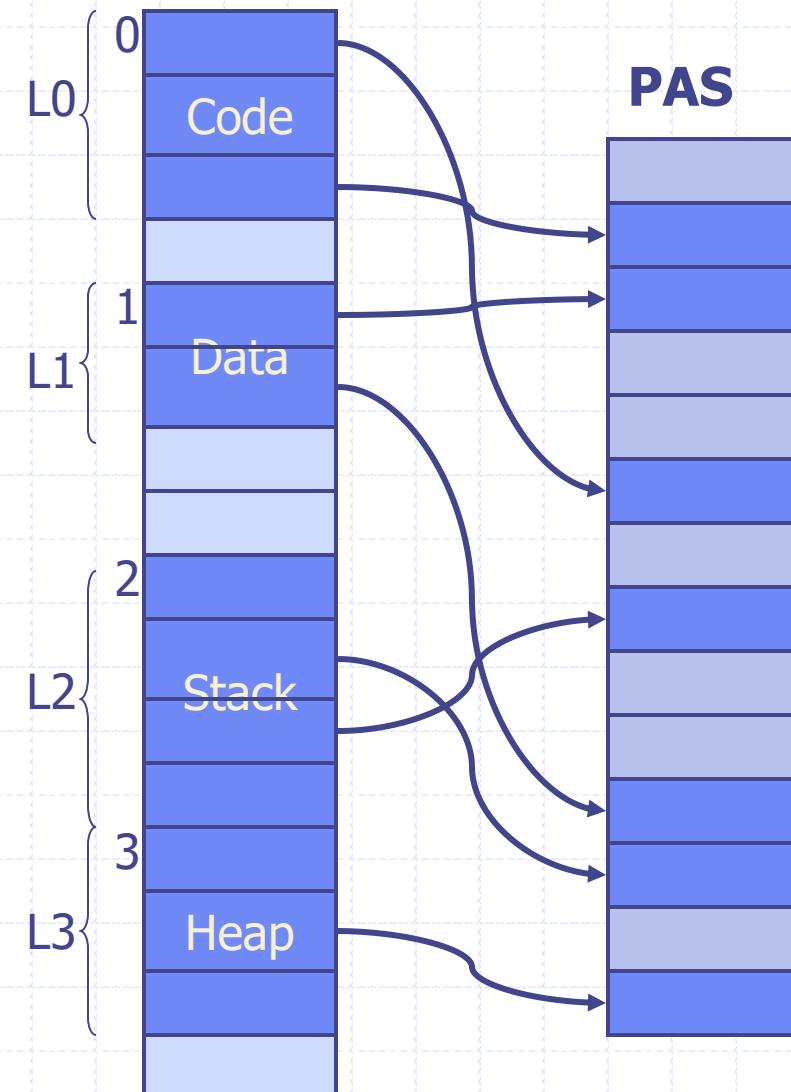
- složeniji hardver, manje efikasno preslikavanje (u odnosu na stranično preslikavanje)
- eksterna fragmentacija
- složeniji zadatak za OS u pogledu alokacije prostora – potreba za pronalaženjem dela slobodne memorije veličine dovoljne za smeštanje segmenta



# Segmentno-stranično preslikavanje

VAS - P1

- ◆ Kombinacija prethodna dva:  
virtuelna memorija je  
podeljena na logičke celine  
različite stvarne veličine -  
*segmente (segment)*, ali svaki  
segment ima ceo broj stranica  
iste veličine
- ◆ Stranice se preslikavaju u  
blokove (okvire) fizičke  
memorije iste veličine



# Segmentno-stranično preslikavanje

## SMT (Process X)

PMTP	Size	Protection (RWE)
	3	101
...	2	110
...	4	110
	3	110

## PMT (Proc X, Seg 0)

Frame	Disk Addr	...
1	05	...
0	...	...
1	01	...
0	...	...

## PMT (Proc X, Seg 3)

Frame	Disk Addr	...
0	...	...
1	0D	...
0	...	...
0	...	...

# Segmentno-stranično preslikavanje

- ◆ Zadatak: prikazati šematski strukturu VA i PA i postupak preslikavanja VA u PA.
- ◆ Prednosti:
  - segmenti su logičke celine, pa je prirodno uvoditi zaštitu pristupa segmentu i prekoračenja veličine; npr. *code segment* je *read-only*
  - nema eksterne fragmentacije
  - jednostavan zadatak za OS u pogledu alokacije prostora – nema potrebe za “upasivanjem” dela memorije
- ◆ Nedostaci:
  - najsloženiji hardver, najmanje efikasno preslikavanje (u odnosu na ostala preslikavanja)
  - interna fragmentacija

# TLB

- ◆ Problem: bez obzira na preslikavanje, procesor mora da dovlači deskriptor iz memorije za svako preslikavanje VA u PA (nekoliko puta tokom instrukcije) – neefikasno!
- ◆ Ideja: obezbediti malu, ali efikasnu memoriju koja čuva samo deo podataka potrebnih za preslikavanje, kao što to radi *keš* (*cache*) memorija –  
*Translation Lookaside Buffer* (TLB)
- ◆ TLB treba da obezbedi brzo preslikavanje određenog dela VA u podatke iz deskriptora potrebne procesoru:
  - ako je podatak u TLB-u, procesor ga dobija brzo
  - ako nije, TLB dovlači podatak iz deksriptora čitanjem iz tabele u memoriji

# TLB

## ◆ Šta TLB preslikava?

- Stranična organizacija: *Page -> (Frame, Protection)*
- Segmentna organizacija: *Segment -> (Start Addr, Size, Protection)*
- Segmentno-stranična organizacija:  
*(Segment, Page) -> (Frame, Size, Protection)*

## ◆ Transparentan za OS i ostali SW, **osim**:

ako TLB sadrži podatke za preslikavanje koji se odnose samo na tekući proces, OS mora da **obriše** (invaliduje) TLB prilikom promene konteksta – mora postojati odgovarajuća instrukcija koja to radi

## ◆ Inače, TLB može da sadrži i identifikaciju procesa koja takođe učestvuje u preslikavanju. Identifikacija tekućeg procesa je u specijalizovanom registru procesora. Ko upisuje tu identifikaciju i kada?

# Podela odgovornosti HW/OS

## ◆ HW treba da obezbedi:

- obavezno preslikavanje VA u PA (u korisničkom režimu)
- (poželjno-praktično obavezno) dva režima, sistemski i korisnički, i odgovarajuću zaštitu
- (poželjno-praktično obavezno) mehanizam zaštite u slučaju prekoračenja veličine segmenta ili ilegalne operacije nad segmentom
- *page fault* ako tražena VA nije u fizičkoj memoriji
- instrukciju za upis u MTP registar
- (poželjno-praktično obavezno) TLB i instrukciju za brisanje TLB, ili registar za identifikaciju procesa i instrukciju za upis u njega

# Podjela odgovornosti HW/OS

## ◆ OS treba da obezbedi:

- pri promeni konteksta procesa:
  - ◆ upis adrese MT tekućeg procesa u MTP registar
  - ◆ upis u registar identifikacije tekućeg procesa (ako postoji)
  - ◆ brisanje TLB (ako je potrebno)
- pri straničnoj grešci (*page fault*):
  - ◆ pronalaženje slobodnog mesta u memoriji za dovlačenje tražene stranice ili segmenta
  - ◆ ako slobodnog mesta nema, izbor nekog za izbacivanje, zatim njegovo snimanje na disk i ažuriranje deskriptora u MT
  - ◆ učitavanje tražene stranice ili segmenta u fizičku memoriju sa diska
  - ◆ ažuriranje deskriptora dovučene stranice ili segmenta
- pri ostalim greškama:
  - ◆ obradu greške, npr. gašenje procesa, izveštavanje korisnika

# Glava 7: Memorijalna hijerarhija

Keš memorija

Magnetski disk

Memorijska hijerarhija

# Keš memorija

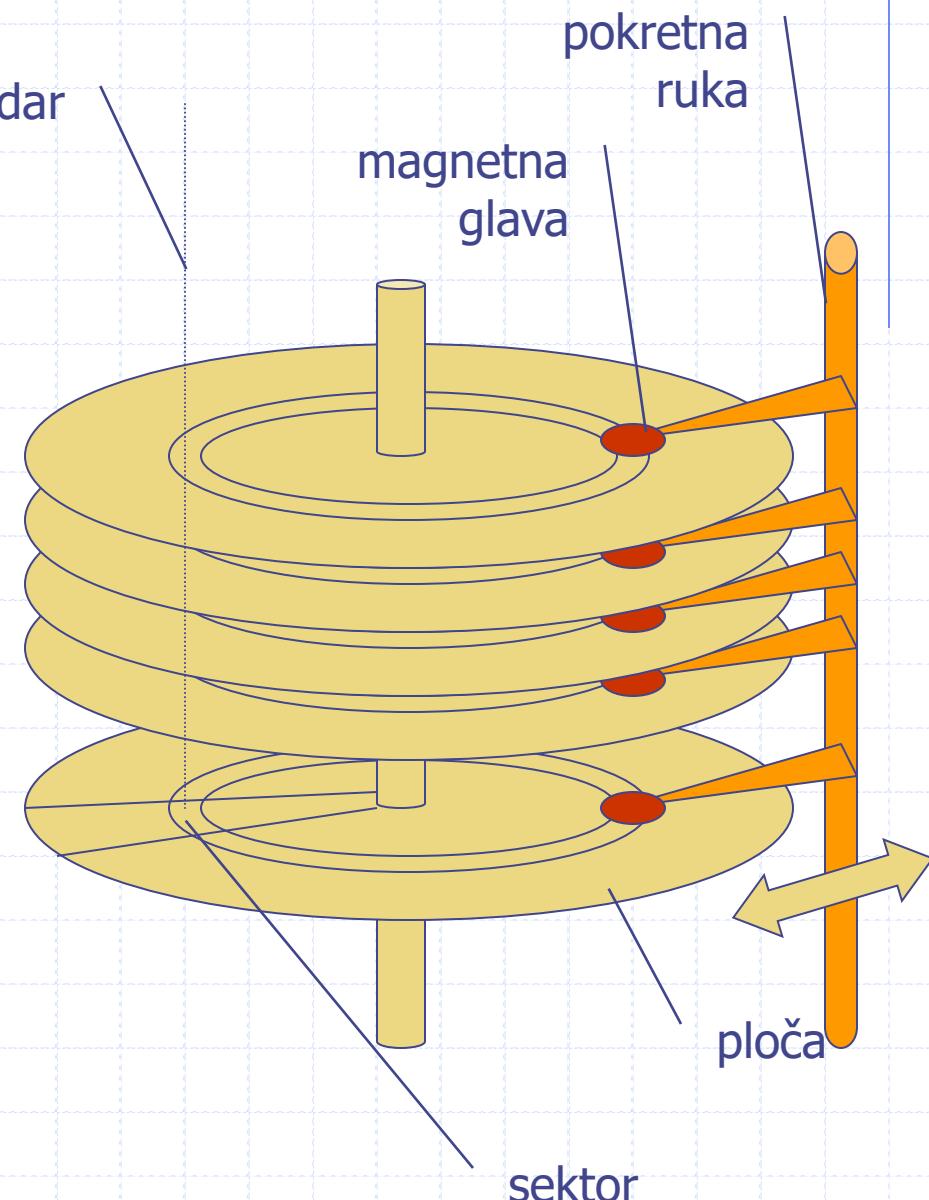
- ◆ Keš (*cache*) memorija: mala, ali brza memorija koja sadrži samo deo adresnog prostora – radni skup instrukcija i podataka
- ◆ Ideja je zasnovana na pretpostavci o vremenskoj i prostornoj lokalnosti izvršavanja programa:
  - ako je program nedavno pristupao nekoj lokaciji, velika je šansa da će u budućnosti ponovo pristupati njoj ("prošlost ukazuje na budućnost")
  - ako je program pristupao nekoj lokaciji, onda je velika šansa da će pristupati i nekoj bliskoj lokaciji
- ◆ Ako je sadržaj lokacije u keš memoriji, odziv je brz; inače, keš memorija očitava ili upisuje podatak prema operativnoj memoriji, uz potencijalno izbacivanje nekog drugog sadržaja

# Keš memorija

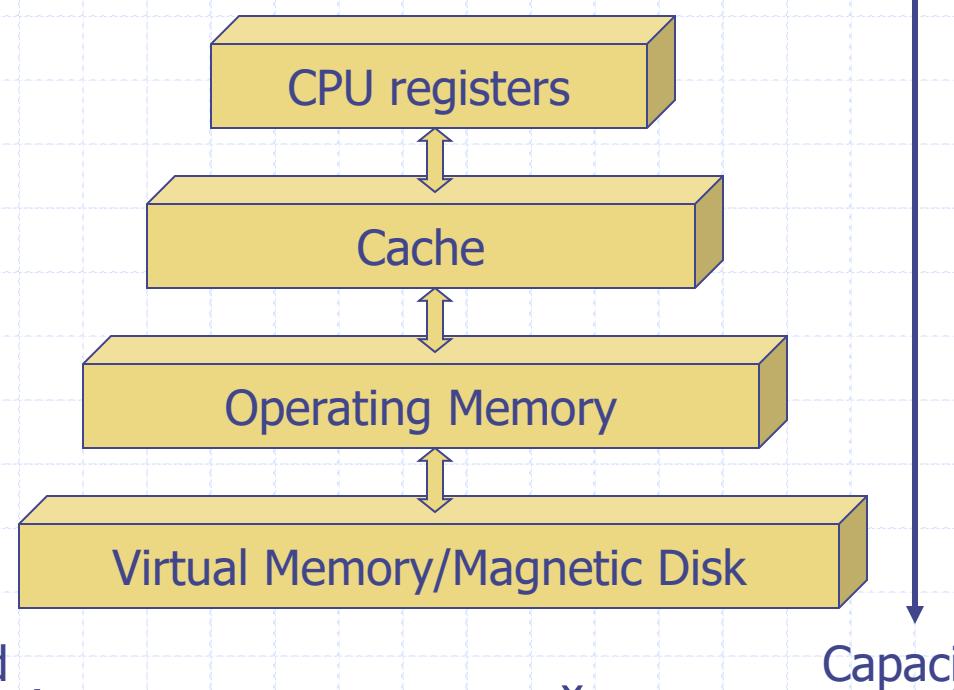
- ◆ Transparentna za OS i ostali SW, **osim:**  
ako keš memorija preslikava virtuelne adrese u sadržaj  
(samo za tekući proces), treba ga obrisati prilikom  
promene konteksta, slično kao i TLB;  
inače, keš preslikava fizičke adrese, ali šta se onda dešava  
prilikom zamene stranice?

# Magnetni disk

- ◆ Mogućnost čitanja i upisa
- ◆ Ne gubi sadržaj gubitkom napajanja (perzistentna memorija)
- ◆ Relativno brz pristup (reda milisekundi), ali značajno sporiji nego RAM
- ◆ Tipični kapacitet danas: nekoliko stotina GB ili TB
- ◆ Parametri:
  - brzina transfera (*transfer rate*)
  - vreme pozicioniranja (*random-access time, positioning time*)



# Memorijska hijerarhija



U datom trenutku, memorija na višem nivou sadrži podskup (*radni skup, working set*) memorijskog prostora nižeg nivoa.

Problemi:

- kako odabratи radni skup?
- šta činitи ako traženi podatak nije na datom nivou?
- kako održavati konzistentnost susednih nivoa – jedan se promeni, šta sa drugim?

# III Upravljanje procesima

---

Procesi i niti

Sinhronizacija i komunikacija između procesa

# Glava 8: Procesi i niti

---

Pojam procesa

Operacije nad procesima

Implementacija procesa

Pojam niti

Implementacija niti

# Pojam procesa

- ◆ *Proces (process)* je izvršavanje programa nad datim podacima:
  - program: statičan zapis instrukcija
  - proces: jedno izvršavanje (instanca, "inkarnacija") datog programa za dati skup podataka – sa jednim *adresnim prostorom*
  - moguće je kreirati više procesa nad istim programom, svaki radi nad svojim podacima – svaki ima svoj adresni prostor
- ◆ Svaki proces odlikuje:
  - *pozicija* u izvršavanju – mesto u programu dokle je izvršavanje stiglo (Program Counter), zajedno sa tragom izvršavanja
  - *stanje* – podaci nad kojima proces radi – registri i *adresni prostor*
- ◆ Pojam *posao (job)* u suštini znači isto što i proces, samo što je arhaičan (paketni sistemi)
- ◆ Pojam *zadatak (task)* ima različito značenje u različitim sistemima i programskim jezicima, ali uglavnom znači isto što i proces

# Operacije nad procesima

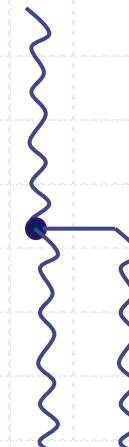
- ◆ Kreiranje procesa: *roditeljski (parent) proces* kreira proces-dete (*child*) sistemskim pozivom
- ◆ Varijante dozvoljavanja korišćenja resursa:
  - proces-dete može da traži bilo koji resurs od OS
  - proces-dete može da koristi samo podskup resursa roditelja
- ◆ Varijante izvršavanja:
  - kada kreira potomka, roditeljski proces se suspenduje (zaustavlja izvršavanje) sve dok se svi potomci ne završe
  - kada kreira potomka, roditeljski proces nastavlja izvršavanje uporedo sa svojim potomcima
- ◆ Varijante formiranja adresnog prostora:
  - potomak dobija kopiju adresnog prostora roditelja – “kloniranje”
  - potomak ima nezavisan program i (novi, prazan) adresni prostor (uključujući i prazan stek)

# Operacije nad procesima

## ◆ Primer – Unix:

- ID procesa – jedinstveni `int`
- sistemski poziv za kreiranje procesa: `fork()`
- proces-dete dobija kopiju adresnog prostora roditelja i kompletan kontekst, pa nastavlja izvršavanje od istog mesta kao i roditelj
- `fork()` vraća 0 u procesu-detetu, a ID deteta ( $\neq 0$ ) u roditelju
- sistemski poziv `wait()` vrši sinhronizaciju procesa – proces-roditelj čeka da se proces-dete završi da bi nastavio dalje
- primer: koliko ukupno procesa kreira sledeći kod?

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



# Operacije nad procesima

- sistemski poziv `execvp()` zamenjuje program pozivajućeg procesa drugim programom – učitava programski kod iz fajla u adresni prostor procesa pozivaoca i počinje njegovo izvršavanje:

```
char* programName = "...";
```

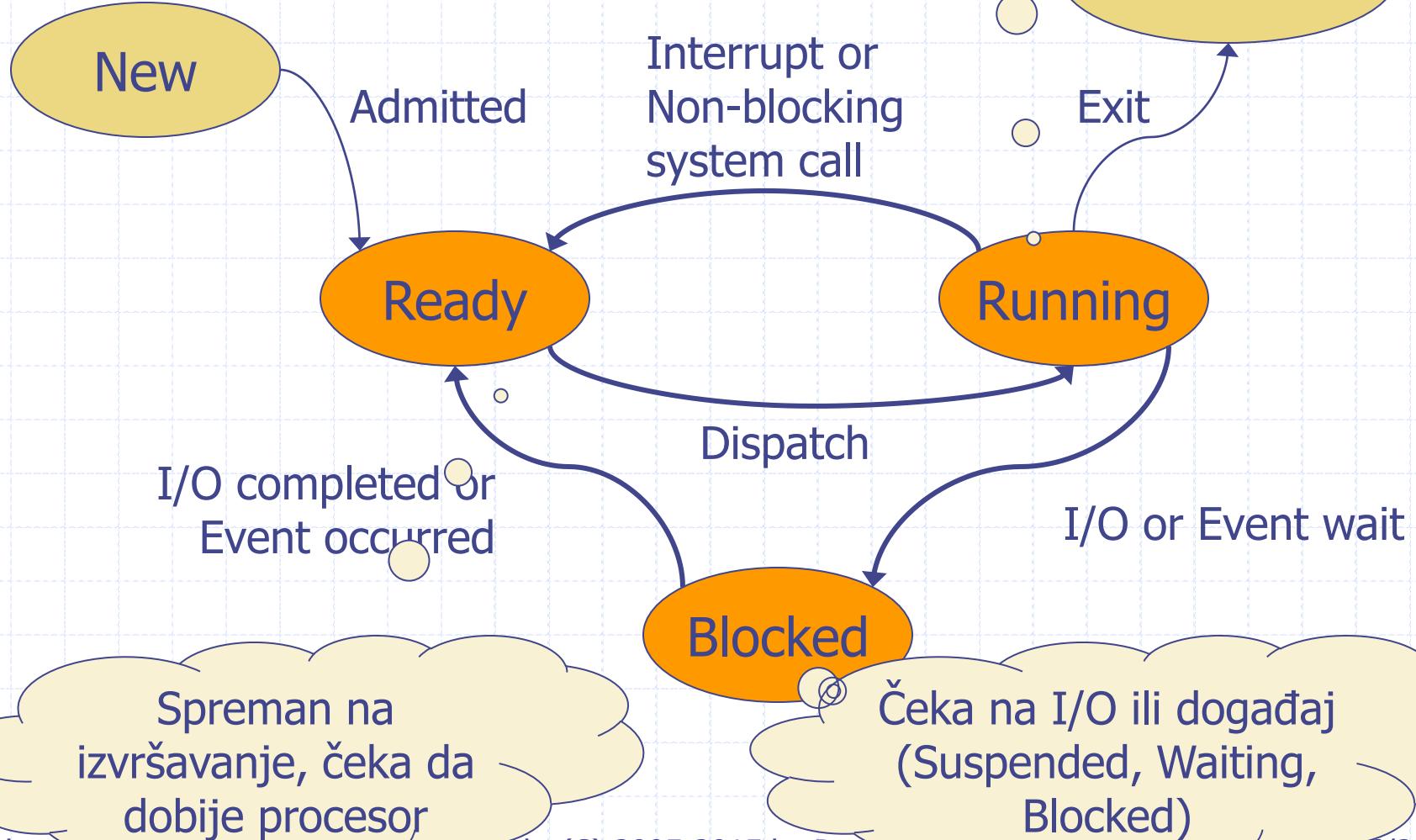
```
void main () {
    int pid = fork();
    if (pid<0) { // Error
        printf("Fork failed.\n"); exit(-1);
    }
    else if (pid==0) // Child process
        execvp(programName);
    else { // Parent process
        wait(NULL);
        printf("Child complete.\n");
        exit(0);
    }
}
```

# Operacije nad procesima

- ◆ MS Windows podržava oba načina:
  - roditeljski adresni prostor se klonira
  - roditelj specifično ime programa nad kojim se kreira nov proces
- ◆ Gašenje procesa:
  - kada završi izvršavanje glavnog programa (`main()` u jeziku C/C++)
  - kada sam proces to eksplicitno zatraži, npr. sistemski poziv `exit()`
  - jedan proces gasi ("ubija") neki drugi proces; moguća su ograničenja, npr. to može da uradi samo roditeljski proces; mogući razlozi:
    - ◆ potomak je iscrpio svoje resurse
    - ◆ nije više potreban
    - ◆ roditelj treba da se ugasi, a OS ne dozvoljava da njegovi potomci dalje rade (kaskadno gašenje)
  - OS gasi proces zbog neke nedozvoljene operacije
- ◆ Primer – Unix: `wait()` vraća ID završenog potomka

# Implementacija procesora

- Stanja procesa tokom životnog veka:



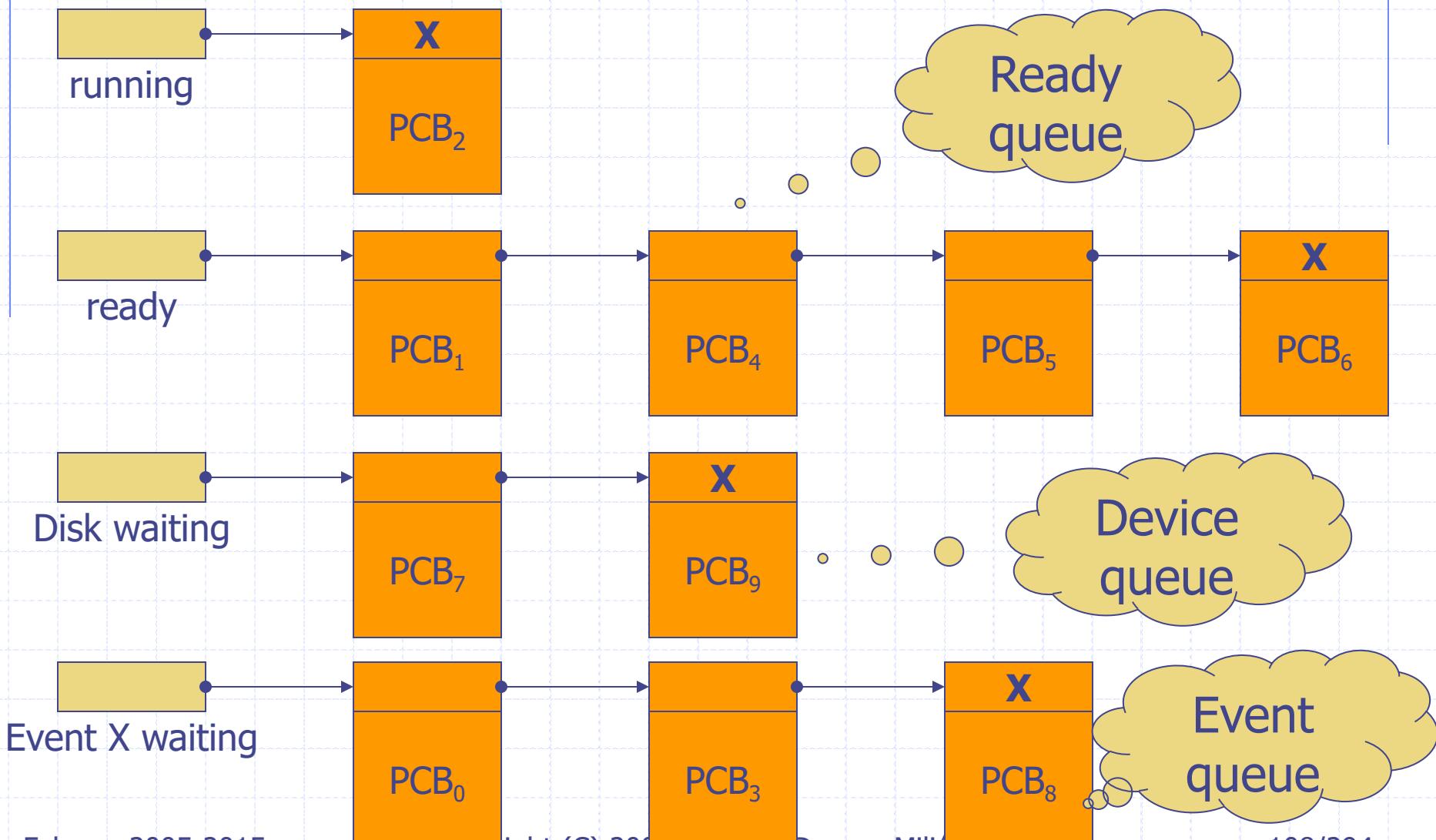
# Implementacija procesa

PCB

- ◆ *Kontekst procesa (process context)*: sve informacije potrebne da OS nastavi izvršavanje procesa, kao i da ga kontroliše i ugasi:
  - ID – interna identifikacija procesa unutar OS
  - kontekst procesora (PC, SP, programski dostupni registri)
  - memorijski parametri: veličina i pozicija dodeljenog memorijskog bloka ili PMTP/SMTP itd.
  - informacije o I/O resursima: spisak otvorenih fajlova, zauzetih resursa itd.
  - podaci potrebni za raspoređivanje: prioritet, dodeljeno CPU vreme, itd.
  - podaci potrebni za obračunavanje
  - ...
- ◆ Struktura podataka u kojoj se čuvaju ove informacije za svaki proces unutar OS – *Process Control Block (PCB)*

ID
Processor context
Memory params
I/O info
Scheduling params
Accounting params
...

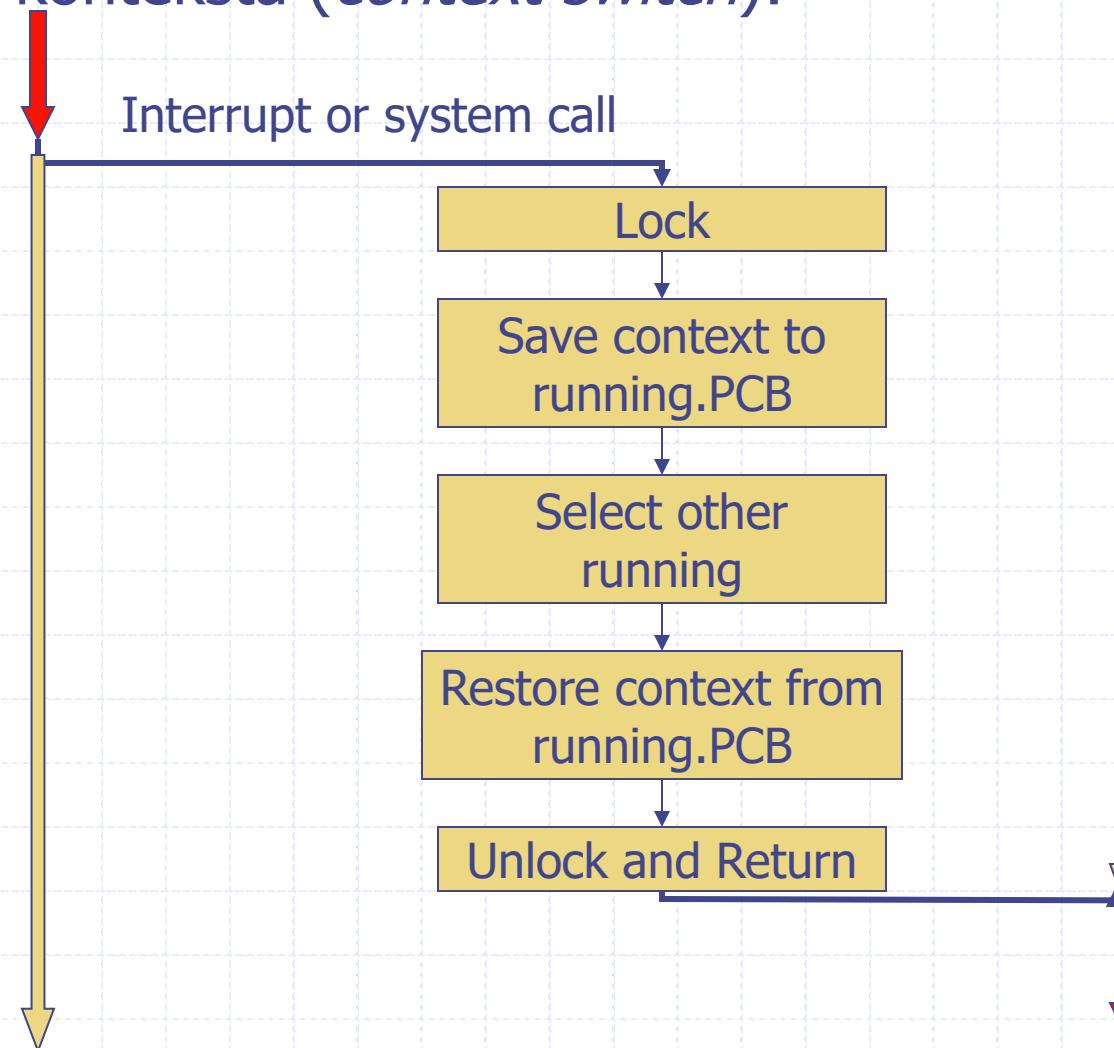
# Implementacija procesa



# Implementacija procesa

- ◆ Promena konteksta (*context switch*):

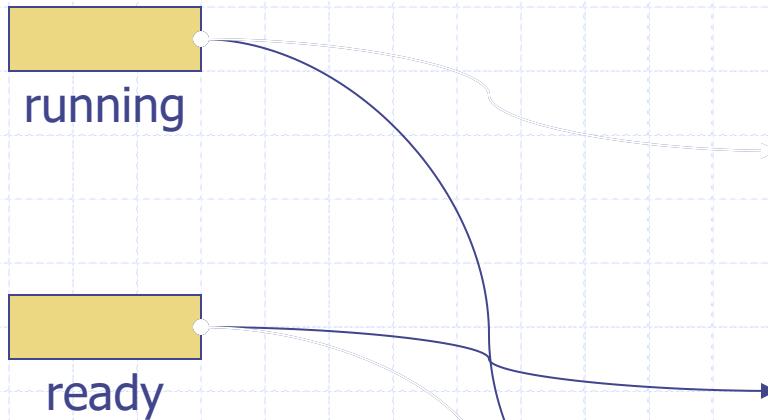
Process A



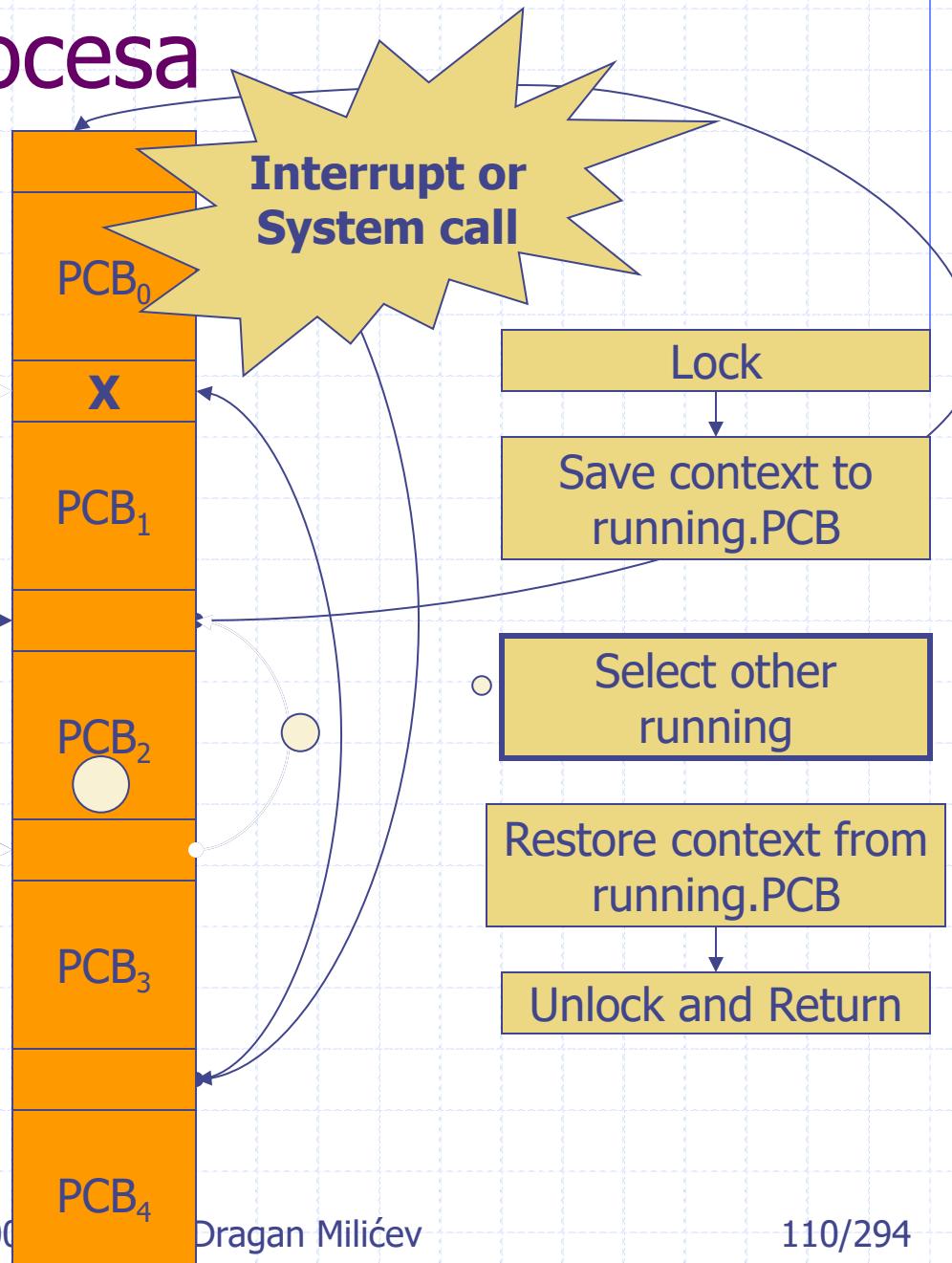
Process B

# Implementacija procesa

## ◆ Promena konteksta:

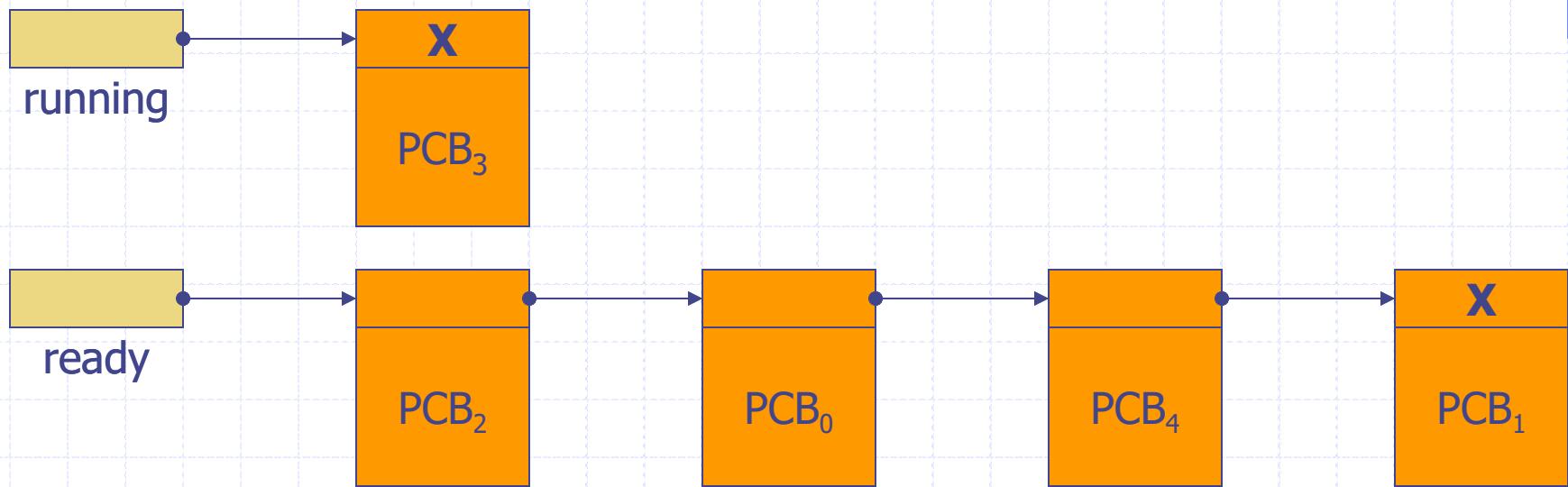


**Kako?  
Problem  
raspoređivanja!**



# Implementacija procesa

## ◆ Promena konteksta:



# Implementacija procesa

## ◆ Zašto su potrebne lock/unlock primitive?

- Da li se sme dozvoliti da se tokom promene konteksta prekine izvršavanje i procesor preuzme neko drugi?
- Šta bi se dogodilo ako se dogodi prekid i počne da se izvršava isti sistemski kod za preuzimanje?
- Šta bi se dogodilo sa strukturama podataka OS-a?

## ◆ Kako implementirati lock/unlock?

- Na jednoprocесорском систему, једноставно маскирати прекиде – прекид је једини извор асинхроног преузимања процесора!
- На мултипроцесорским системима, HW мора да обезбеди одговарајућу подршку! (Детаљи касније)

# Implementacija procesa

- ◆ Promena konteksta je čist režijski trošak – sistem ne radi ništa korisno za aplikacije? Ali koliko "košta"?
  - Zavisi od procesora: šta treba sačuvati kao kontekst i kakva je HW podrška (instrukcije, registri); primer: Sun UltraSPARC sadrži više registarskih fajlova i instrukciju za promenu tekućeg
  - Zavisi od OS: koliko informacija treba čuvati kao kontekst procesa i šta još treba uraditi
  - od 1 do 1000 mikrosekundi
- ◆ Kako smanjiti ove troškove? Umesto procesa na nivou OS, kad god je moguće koristiti *niti*

# Pojam niti

- ◆ *Nit (thread)* predstavlja jedno izvršavanje nekog dela koda programa unutar adresnog prostora nekog procesa kreiranog unutar OS-a
  - Proces na nivou OS-a (teški proces, *heavyweight process*): jedno izvršavanje jednog programa sa sopstvenim adresnim prostorom
  - Nit (laki proces, *lightweight process*): jedno izvršavanje dela koda programa unutar adresnog prostora "okružujućeg" procesa
- ◆ Više niti može biti kreirano unutar istog procesa:
  - svaka ima svoj *tok kontrole (thread of control)* – svoju poziciju izvršavanja u programskom kodu
  - sve niti dele isti adresni prostor i resurse – otvorene fajlove, globalne podatke programa
  - svaka ima svoje lokalne podatke za pozive potprograma
- ◆ Obično se kreiraju nad pozivom jednog potprograma (i svim ugnježdenim pozivima)

# Pojam niti

## ◆ Primer – Java:

- Nit se kreira kao objekat klase izvedene iz bibliotečne klase **Thread**
- Nit se pokreće pozivom operacije **start()**
- Telo niti (programske funkcije) predstavlja polimorfna operacija **run()**

```
public class UserInterface {  
    public int newSetting (int dim) { ... }  
    ...  
}  
  
public class Arm {  
    public void move(int dim, int pos) { ... }  
}
```

# Pojam niti

```
public class Control extends Thread {  
    private int dim;  
    private static UserInterface ui =  
        new UserInterface();  
    private static Arm robot = new Arm();  
  
    public Control(int dimension) {  
        super();  
        dim = dimension;  
    }  
  
    public void run() {  
        int position = 0, setting = 0;  
        while(true) {  
            robot.move(dim,position);  
            setting = ui.newSetting(dim);  
            position = position + setting;  
        }  
    }  
}
```

# Pojam niti

```
int xPlane = 0;  
int yPlane = 1;  
int zPlane = 2;  
  
Control c1 = new Control(xPlane);  
Control c2 = new Control(yPlane);  
Control c3 = new Control(zPlane);  
c1.start();  
c2.start();  
c3.start();
```

◆ Primeri upotrebe - ista aplikacija, isti proces, isti program, ali više uporednih aktivnosti-niti:

- tekst-procesor: snima dokument u pozadini, proverava gramatiku u pozadini, uporedo obrađuje pritiske tastera i druge akcije korisnika
- Web Browser: dovlači slike ili drugi sadržaj, prikazuje dohvaćeno, obrađuje akcije korisnika
- server: po jedna nit nad istim programom za svaki zahtev klijenta

# Pojam niti

## ◆ Koristi:

- Bolji odziv interaktivne aplikacije: duge aktivnosti se mogu raditi "u pozadini", a uporedo prihvati akcije korisnika
- Deljenje resursa: niti dele memoriju (adresni prostor) i otvorene fajlove
- Ekonomičnost: kreiranje procesa, promena konteksta, alokacija memorije i ostalih resursa je skupo; niti te režijske troškove ili eliminišu, ili drastično smanjuju; npr. Solaris: kreiranje 30:1, promena konteksta 5:1

# Implementacija niti

## ◆ Podrška nitima može biti:

- na nivou korisničkog programa (*user threads*): niti podržava izvršno okruženje ili biblioteka programskog jezika, OS nema koncept niti; prednosti: efikasnost i jednostavnost mane: kada se jedna nit blokira na sistemskom pozivu, ceo proces se blokira
- na nivou OS-a (jezgra, *kernel threads*): OS direktno podržava koncept niti

## ◆ Ako su podržane na oba nivoa, modeli preslikavanja:

- više u jedan (*many-to-one*): više korisničkih implementira jedna sistemska nit ili proces
- jedan na jedan (*one-to-one*): jednu korisničku nit implementira jedna sistemska nit
- više u više (*many-to-many*): više korisničkih implementira isti ili manji broj sistemačkih niti

# Implementacija niti

- ◆ Niti u okviru istog procesa imaju zajedničko:
  - globalne podatke
  - programski kod
  - resurse – fajlove i druge sistemske resurse
- ◆ Niti u okviru istog procesa imaju sopstveno:
  - poziciju u izvršavanju – PC i trag izvršavanja – stek (SP)
  - podatke lokalne za potprograme – stek (SP) i promenljive alocirane u registe
- ◆ Sledi: kontekst niti svodi se na kontekst procesora – PCB (TCB) je jednostavniji, promena konteksta je jednostavnija

# Implementacija niti

## ◆ Primer niti na jeziku C++ - "Školsko jezgro":

- korisničke niti implementirane bibliotekom za dati jezik
- semantika i upotreba slična nitima u Javi
- može se izvršavati i bez OS-a, na "goloj" mašini

```
#include "kernel.h" // uključivanje deklaracija Jezgra
#include <iostream.h>
```

```
void threadBody () {
    for (int i=0; i<3; i++) {
        cout<<i<<"\n";
        dispatch(); // Eksplicitno preuzimanje - yield
    }
}
void userMain () {
    Thread* t1=new Thread(threadBody);
    Thread* t2=new Thread(threadBody);
    t1->start();
    t2->start();
    dispatch();
}
```

# Implementacija niti

- ◆ Mana ovakvog "proceduralnog" stila kreiranja niti: niti nad istim kodom (istom funkcijom) ne mogu se razlikovati jer polazna funkcija nema argumente
- ◆ Moguće rešenje: obezbediti argument funkcije, ali unapred definisanog tipa (npr. `void*` na strukturu argumenata). Problem: nije tipizirano i nije u duhu OOP
- ◆ Objektno rešenje – poput Java:

```
class Thread {  
public:  
  
    Thread ();  
    Thread (void (*body) ()) ;  
    void start ();  
  
protected:  
  
    virtual void run () {}  
  
};
```

# Implementacija niti

- ◆ Promena konteksta na način nezavisan od procesora i prevodioca:
  - zaglavljje `<setjmp.h>` sadrži potrebne deklaracije
  - tip `jmp_buf` predstavlja PCB za nit (TCB), tako da sadrži sve programske dostupne registre koje koristi dati prevodilac na dатoj mašini (obavezno PC i SP)
  - `int setjmp(jmp_buf context)`: smešta kontekst procesora u TCB dat kao argument i vraća 0
  - `void longjmp(jmp_buf context, int value)`: restaurira kontekst dat kao argument, a koji je snimljen pomoću `setjmp()`; pošto se time skače unutar `setjmp()`, tada `setjmp()` vraća vrednost `value` koja mora biti !=0

# Implementacija niti

Promena konteksta:

```
void Thread::dispatch () {
    lock ();
    if (setjmp(runningThread->context)==0) {

        Scheduler::put(runningThread);
        runningThread = Scheduler::get();

        longjmp(runningThread->context,1);

    } else {
        unlock ();
        return;
    }
}
```

Problem: da li je ovaj pristup moguć kod asinhronog preuzimanja (zbog prekida)? Sta onda?

# Glava 9: Sinhronizacija i komunikacija između procesa

Kooperativni procesi

Mehanizmi interakcije

Međusobno isključenje

Uslovna sinhronizacija

Uposleno čekanje

Semafori

Implementacija semafora

Upotreba semafora

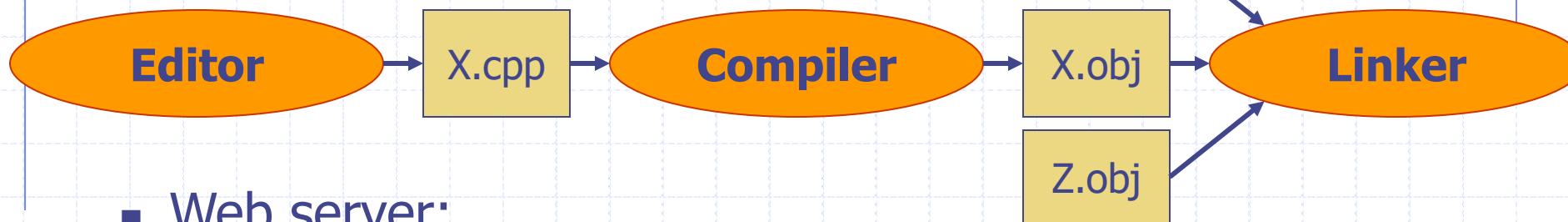
# Kooperativni procesi

- ◆ Procesi mogu biti međusobno logički
  - nezavisni, izolovani: nema interakcije niti razmene informacija između njih, procesi se izvršavaju uporedo i nezavisno
  - kooperativni: procesi moraju da razmenjuju informacije ili da se međusobno sinhronizuju - dele stanje (podatke)
- ◆ Međutim: u multiprocesnom OS izolovanih procesa praktično nema – svi oni dele resurse sistema (memoriju, fajlove, I/O uređaje)
- ◆ Čemu kooperativni procesi?
  - smanjiti troškove deobom resursa (jedan računar, više poslova; jedan autoput, mnogo vozila; jedna učionica, mnogo časova)
  - procesi zahtevaju informacije od drugih procesa da bi izvršili svoj zadatak;
    - ◆ Zašto se uopšte dele na odvojene procese? - Modularnost
    - ◆ Zašto da se izvršavaju konkurentno? – Povećanje efikasnosti

# Kooperativni procesi

## ◆ Primeri:

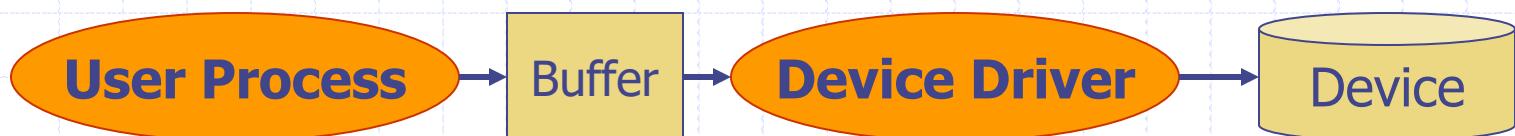
- editor, prevodilac i linker:



- Web server:



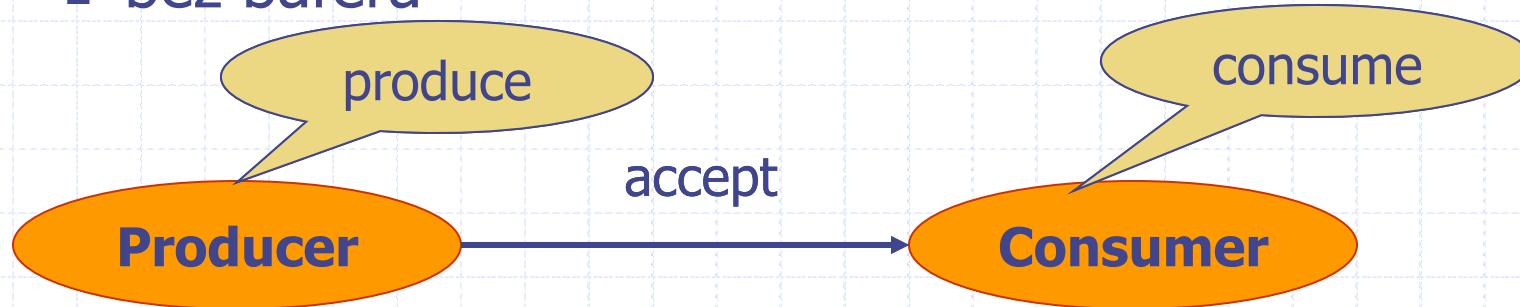
- korisnički proces i proces koji upravlja nekim izlaznim uređajem:



# Kooperativni procesi

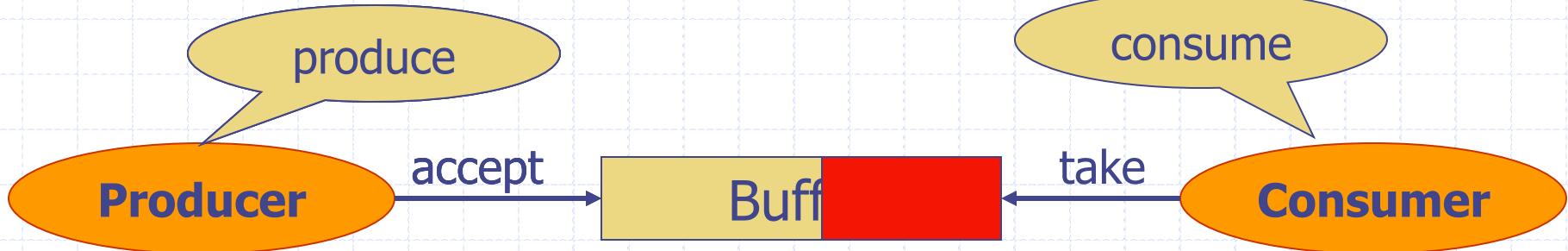
## ◆ Problem *proizvođač-potrošač* (*producer-consumer*):

- bez bafera



Problem: nepotrebna sekvencijalizacija (naizmeničnost), smanjena konkurentnost

- sa baferom



# Kooperativni procesi

## ◆ Ograničeni bafer (*bounded buffer*):



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

class BoundedBuffer {
public:
    BoundedBuffer ();

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

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

# Kooperativni procesi

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

# Kooperativni procesi

- ◆ Platforma – način izvršavanja uporednih procesa:
  - *Multiprogramiranje*: na jednom procesoru, uz preplitanje
  - *Multiprocesiranje*: na više procesora (multiprocesorski ili distribuirani sistem), fizički *paralelno* (istovremeno) – paralelno procesiranje
- ◆ *Konkurentnost (concurrency)*: konceptualno, uporedno izvršavanje, potencijalni parallelizam
- ◆ *Paralelno procesiranje (parallel processing)*: fizički istovremeno izvršavanje (na više procesora)
- ◆ Multiprogramiranje na jednom procesoru:
  - *bez preotimanja (non-preemptive)*: samo sinhrona promena konteksta
  - *sa preotimanjem (preemptive)*: asinhrona promena konteksta (prekidi)
- ◆ Parametri multiprogramiranja:
  - kada dolazi do promene konteksta: sinhrono/asinhrono
  - redosled izvršavanja: algoritam raspoređivanja

# Kooperativni procesi

- ◆ Međutim, konkurentnost i deljenje resursa uzrokuju probleme; primeri: jedna saobraćajna traka-vozila koja idu u susret
- ◆ Nezavisni procesi: rezultat izvršavanja procesa ne zavisi od redosleda izvršavanja i preplitanja sa drugim nezavisnim procesima – ne zavisi od raspoređivanja)
- ◆ Procesi koji dele podatke: rezultat izvršavanja zavisi od redosleda izvršavanja i preplitanja – zavisi od raspoređivanja. Problemi: rezultat je nedeterminisan, uočavanje i ispravljanje grešaka veoma teško, jer su one često nereproducibilne
- ◆ Osnovni postulat konkurentnog programiranja: rezultat (logička ispravnost) programa *ne sme* da zavisi od platforme i parametara (redosleda izvršavanja i preplitanja, tj. od raspoređivanja)

# Mehanizmi interakcije

## ◆ Vrste interakcija između procesa:

- *sinhronizacija (synchronization)*: zadovoljavanje ograničenja u pogledu preplitanja akcija različitih procesa (npr. neka akcija jednog procesa mora da se dogodi pre neke akcije drugog procesa i sl.); simultano dovođenje više procesa u predefinisano stanje
- *komunikacija (communication)*: razmena informacija između procesa

## ◆ Ovi pojmovi su povezani jer:

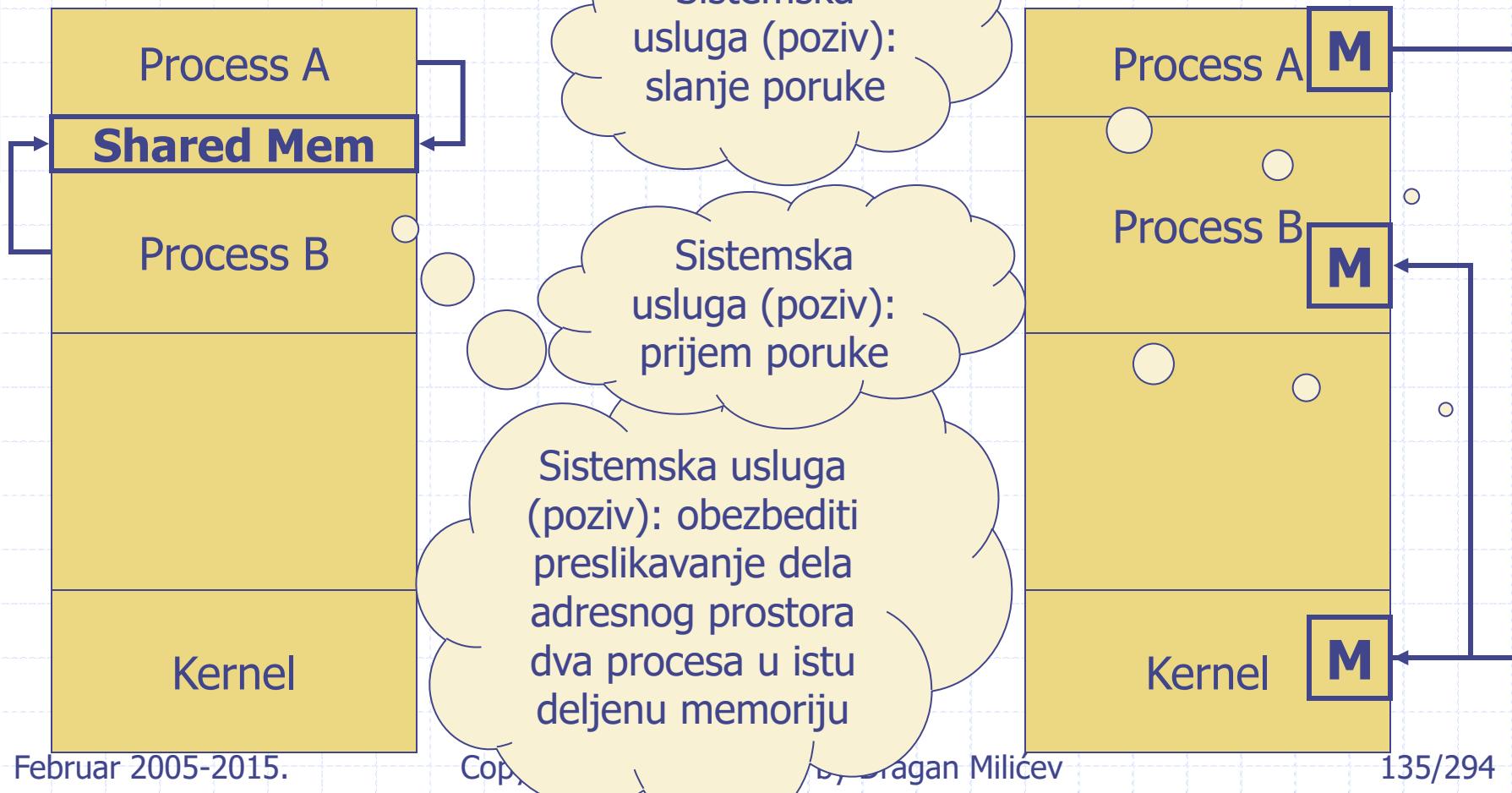
- neki mehanizmi komunikacije podrazumevaju prethodnu sinhronizaciju
- sinhronizacija se može smatrati komunikacijom bez razmene sadržaja

# Mehanizmi interakcije

- ◆ 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

# Mehanizmi interakcije

Implementacija na sistemu za deljenom memorijom, ali sa procesima koji nemaju isti adresni prostor – IPC kao usluga OS-a:



# Međusobno isključenje

- ◆ Primer 1: Šta se dešava ako dva procesa pristupaju uporedo istoj deljenoj promenljivoj  $x$ ?

$x := x + 1$

- ◆ Na većini procesora, ova naredba biće prevedena u (neatomičnu) sekvencu (atomičnih) instrukcija:

load reg with  $x$   
increment reg  
store reg to  $x$

- ◆ Ako je  $x$  incijalno bilo 0, koji su mogući rezultati?

Slučaj 1:

P1                  P2

load  $x(0)$

load  $x(0)$

inc

inc

store  $x(1)$

store  $x(1)$

**$x=1$**

Slučaj 2:

P1

load  $x(0)$

inc

store  $x(1)$

P2

load  $x(1)$

...

**$x=2$**

# Međusobno isključenje

- ◆ Primer 2: Policijski helikopter prati kriminalca-begunca i navodi policijski automobil koji ga juri

```
type Coord = record {
    x : integer;
    y : integer;
};
```

```
var sharedCoord : Coord;
```

```
process Helicopter
var nextCoord : Coord;
begin
loop
    computeNextCoord(nextCoord) ;
    sharedCoord := nextCoord;
end;
end;
```

```
process PoliceCar
begin
loop
    moveTo(sharedCoord) ;
end;
end;
```

# Međusobno isključenje

Proces Helicopter upisuje u sharedCoord:	Vrednost u sharedCoord:	Proces PoliceCar čita iz sharedCoord:
	0,0	
x:=1	1,0	
y:=1	1,1	
	1,1	x=1
	1,1	y=1
x:=2	2,1	
y:=2	2,2	
	2,2	x=2
	2,2	y=2
x:=3	3,2	
	3,2	x=3
	3,2	y=2
y:=3	3,3	

Rezultat: kriminalac beži, iako ga helikopter dobro prati!

# Međusobno isključenje

- ◆ Primer 3: Ograničeni bafer. Šta se može desiti ako dva procesa-proizvođača uporedo pozivaju `append`?
- ◆ Deo koda (sekvenca naredbi) procesa koji se mora izvršavati nedeljivo (*invisible*) u odnosu na druge takve delove koda drugih procesa naziva se *kritična sekcija* (*critical section*)
- ◆ Sinhronizacija koja je neophodna da bi se obezbedila atomičnost izvršavanja kritičnih sekcija naziva se *međusobno isključenje* (*mutual exclusion*, Dijkstra 1965.)
- ◆ Prepostavlja se da je atomičnost operacije dodele vrednosti skalarnoj promenljivoj obezbeđena na nivou upisa u memoriju (instrukcija upisa vrednosti u skalarnu promenljivu je atomična)

# Uslovna sinhronizacija

- ◆ *Uslovna sinhronizacija (condition synchronization):* jedan proces želi da izvrši akciju koja ima smisla ili je sigurna samo ako je neki drugi proces preuzeo neku svoju akciju ili se nalazi u nekom definisanom stanju
- ◆ Uslovna sinhronizacija kod ograničenog bafera:
  - proizvođač ne sme da stavi podatak u bafer ukoliko je bafer pun
  - potrošač ne može da uzme podatak iz bafera ukoliko je bafer prazan
- ◆ Zaključak: implementacija ograničenog bafera treba da obezbedi i međusobno isključenje i uslovnu sinhronizaciju

# Uposleno čekanje

- ◆ Uslovna sinhronizacija – opšti slučaj:

```
process P1; (*Waiting*)
begin
  ...
  while flag = false do
    null
  end;
  ...
end P1;
```

```
process P2; (*Signalling*)
begin
  ...
  flag:=true;
  ...
end P2;
```

- ◆ Uslovna sinhronizacija – ograničeni bafer:

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

# Uposleno čekanje

- ◆ Ovakva realizacija, gde proces koji čeka izvršava praznu petlju dok uslov nastavka ne bude zadovoljen, naziva se *uposleno čekanje (busy waiting)*
- ◆ Očigledan veliki nedostatak: proces koji uposleno čeka troši procesorsko vreme na "jalove" instrukcije, ne radeći ništa korisno
- ◆ Uslovna sinhronizacija je jednostavna, ali šta je sa međusobnim isključenjem?

# Uposleno čekanje

## ◆ Pokušaj rešenja 1:

```
process P1
begin
    loop
        flag1 := true;          (* Announce intent to enter *)
        while flag2 = true do null end; (* Busy wait *)
        <critical section>      (* Critical section *)
        flag1 := false;          (* Exit protocol *)
        <non-critical section>
    end
end P1;
```

Problem: živo blokiranje (*livelock*)

```
process P2
begin
    loop
        flag2 := true;          (* Announce intent to enter *)
        while flag1 = true do null end; (* Busy wait *)
        <critical section>      (* Critical section *)
        flag2 := false;          (* Exit protocol *)
        <non-critical section>
    end
end P2.
```

# Uposleno čekanje

## ◆ Pokušaj rešenja 2:

```
process P1
begin
    loop
        while flag2 = true do null end; (* Busy wait *)
        flag1 := true;
        <critical section>          (* Critical section *)
        flag1 := false;              (* Exit protocol *)
        <non-critical section>
    end
end P1;

process P2
begin
    loop
        while flag1 = true do null end; (* Busy wait *)
        flag2 := true;
        <critical section>          (* Critical section *)
        flag2 := false;              (* Exit protocol *)
        <non-critical section>
    end
end P2;
```

Problem: ne obezbeđuje međusobno  
isključenje zbog *utrkivanja*  
*(race condition)*

# Uposleno čekanje

## ◆ Pokušaj rešenja 3:

```
process P1
begin
    loop
        while turn = 2 do null end; (* Busy wait *)
        <critical section>          (* Critical section *)
        turn := 2;                   (* Exit protocol *)
        <non-critical section>
    end
end P1;

process P2
begin
    loop
        while turn = 1 do null end; (* Busy wait *)
        <critical section>          (* Critical section *)
        turn := 1;                   (* Exit protocol *)
        <non-critical section>
    end
end P2;
```

Problem: suvišna sekvencijalizacija  
(naizmeničnost) – slaba konkurentnost

# Uposleno čekanje

- ◆ Rešenje (Peterson 1981.):

```
process P1
begin
    loop
        flag1:=true; turn:=2;          (* Announce intent to entry *)
        while flag2 and turn=2 do null end; (* Busy wait *)
        <critical section>           (* Critical section *)
        flag1 := false;              (* Exit protocol *)
        <non-critical section>
    end
end P1;

process P2
begin
    loop
        flag2:=true; turn:=1;          (* Announce intent to entry *)
        while flag1 and turn=1 do null end; (* Busy wait *)
        <critical section>           (* Critical section *)
        flag2 := false;              (* Exit protocol *)
        <non-critical section>
    end
end P2;
```

Problemi:

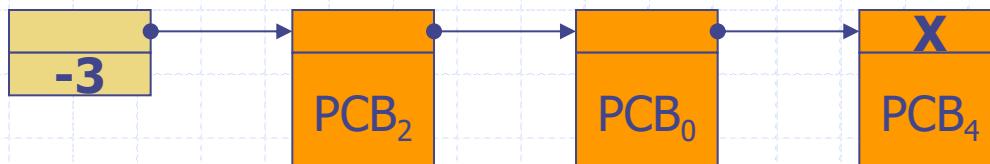
- ◆ neefikasnost
- ◆ složenost, posebno za opšti slučaj više procesa

# Semafori

- ◆ Jednostavan koncept i mehanizam za programiranje međusobnog isključenja i uslovne sinhronizacije (Dijkstra 1968.)
- ◆ Semafor je celobrojna nenegativna promenljiva nad kojom se, osim inicijalizacije, mogu vršiti samo dve operacije:
  - **wait (S)** : (Dijkstra je originalno zvao P) Ako je vrednost semafora S veća od nule, ta vrednost se umanjuje za jedan; u suprotnom, proces mora da čeka sve dok S ne postane veće od nule, a tada se vrednost takođe umanjuje za jedan
  - **signal (S)** : (Dijkstra je originalno zvao V) Vrednost semafora se uvećava za jedan
- ◆ Operacije **wait** i **signal** su atomične – atomičnost implicitno obezbeđuje implementacija => procesi koji izvršavaju ove operacije ne interaguju

# Implementacija semafora

- ◆ Kada je vrednost semafora nula, proces koji je izvršio operaciju `wait()` treba da čeka da neki drugi proces izvrši operaciju `signal()`. Kako ovo realizovati?
  - Uposlenim čekanjem? – Može, ali je neefikasno
  - *Blokiranjem* (ili *suspenzijom*, engl. *blocking, suspension*): PCB procesa za koji nije zadovoljen uslov nastavka izvršavanja ne vraća se u red spremnih, već u red čekanja na semaforu – ne troši procesorsko vreme na izvršavanje sve dok uslov ne bude zadovoljen
- ◆ Moguća implementacija semafora:
  - 1)  $\text{val} > 0$ : još val procesa može da izvrši operaciju `wait` a da se ne blokira, nema procesa blokiranih na semaforu;
  - 2)  $\text{val} = 0$ : nema blokiranih na semaforu, ali će se proces koji naredni izvrši `wait` blokirati;
  - 3)  $\text{val} < 0$ : ima  $-\text{val}$  blokiranih procesa, a `wait` izaziva blokiranje



# Implementacija semafora

```
procedure wait(S)
```

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

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

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

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

```
    end
```

```
end;
```

```
procedure signal(S)
```

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

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

*take one process from the suspended queue of S*

*and deblock it by putting it into the ready queue*

```
    end
```

```
end;
```

# Implementacija semafora

1: Proces gubi procesor, ali ostaje spreman:

- ◆ Sinhrono (neblokirajući sistemski poziv):
  - ◆ eksplicitno sam zatražio preuzimanje (*dispatch, yield*)
  - ◆ drugi neblokirajući poziv (npr. *signal*)
- ◆ Asinhrono (prekid):
  - ◆ isteklo dodeljeno CPU vreme (*time exceeded*)
  - ◆ preuzimanje jer je postao spreman neki drugi proces (*preemption*)

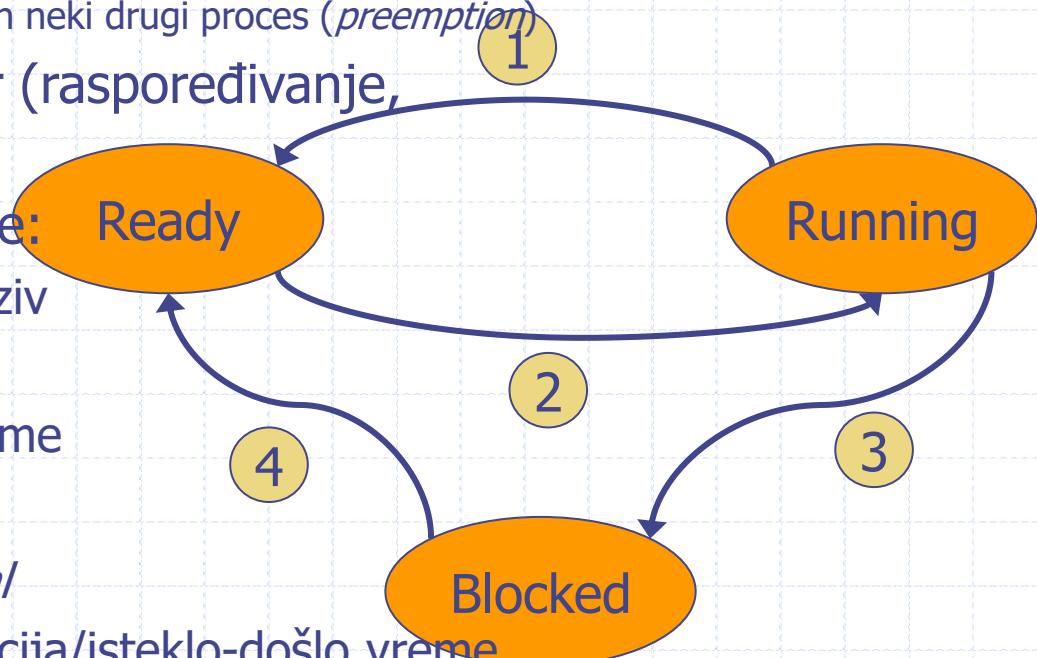
2: Izabrani proces dobija procesor (raspoređivanje, *scheduling*)

3: Proces gubi procesor i blokira se:

- ◆ izvršio blokirajući sistemski poziv
- ◆ zatražio I/O operaciju
- ◆ suspendovao se na zadato vreme

4: Proces se deblokira:

- ◆ *running* proces je izvršio *signal*
- ◆ ispunjen uslov/završena operacija/isteklo-došlo vreme



# Implementacija semafora

◆ Operacije *wait* i *signal* treba da budu atomične – *kritične sekcije*. Kako obezbediti *međusobno isključenje*?

- Upotrebiti prikazano rešenje sa uposlenim čekanjem? – Moguće, ali kako realizovati algoritam za nepoznat broj procesa?
- Atomičnost (međusobno isključenje) na višem nivou mora da se realizuje konceptom nižeg nivoa koji obezbeđuje atomičnost – atomičnost se ne može stvoriti “ni iz čega”
- Mora postojati podrška hardvera!

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

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



Kako implementirati  
lock() i  
unlock()?

# Implementacija semafora

## ◆ Implementacija lock/unlock na jednoprocesorskom sistemu:

### 1. Zabraniti (maskirati) prekide:

- Prednost: radi, i to jednostavno i efikasno
- Nedostatak (u opštem slučaju): prekide treba što manje maskirati, jer je sistem "neosetljiv, mrtav", dok su prekidi maskirani – ne reaguje na spoljašnje pobude; u slučaju otkaza ili beskonačnog izvršavanja, sistem ostaje "hiberniran"; ovde je pogodnost to što je kod operacija *wait* i *signal*:
  - pod kontrolom OS (nema grešaka, nema otkaza)
  - konačnog, predvidivog i kratkog trajanja

### 2. Zabraniti preuzimanje: prekidi su dozvoljeni, ali nije dozvoljeno preuzimanje; ako se dogodi prekid, samo zabeležiti to za kasniju obradu, ne započinjati preuzimanje, već se samo vratiti

# Implementacija semafora

- ◆ Implementacija lock/unlock na multiprocesorskom sistemu: HW mora da obezbedi instrukciju za podršku, npr.:

1. *Test-And-Set*: atomična instrukcija koja postavlja vrednost bita u zajedničkoj memoriji na 1, a vraća prethodnu vrednost; atomičnost se obezbeđuje hardverski, npr. zauzimanjem magistrale tokom cele instrukcije:

```
lock (L) :
```

```
    while (test_and_set(L)) do null;
```

```
unlock (L) :
```

```
    L:=0;
```

Uposleno čekanje? – Da, ali:

- kod je pod kontrolom OS (nema grešaka, nema otkaza)
- konačnog, predvidivog i kratkog trajanja

# Implementacija semafora

2. *Swap*: atomična zamena vrednosti registra i memorijske lokacije; sličan mehanizam kao i *Test-And-Set*

Varijanta 1:

**lock (L) :**

```
for (int acquired = 0; !acquired; )
    swap(acquired, L);
```

*Swap* instrukcije su obično skupe.

Varijanta 2:

**lock (L) :**

```
for (int acquired = 0; !acquired; ) {
    while (!L);
    swap(acquired, L);
}
```

Mogu li se ove instrukcije iskoristiti za implementaciju lock/unlock i na jednoprocесорском систему? – Zašto da ne!  
U čemu je razlika?

# Implementacija semafora

```
class Semaphore {  
public:  
    Semaphore (int initialValue=1) : val(initialValue), lck(0) {}  
  
    void wait ();  
    void signal ();  
  
    int value () { return val; };  
  
protected:  
  
    void block ();  
    void deblock ();  
  
private:  
  
    int val;  
    Queue blocked;  
    int lck; // lock  
};
```

# Implementacija semafora

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

void Semaphore::deblock () {
    // Deblocking:
    Thread* t = blocked.get();
    Scheduler::put(t);
}
```

# Implementacija semafora

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

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

# Implementacija semafora

- ◆ Mogu li se primitive lock/unlock koristiti i za međusobno isključenje na nivou korisničkog programa? – Obezbeđuju međusobno isključenje, ali:
  - Zabрана prekida ili preuzimanja:
    - ◆ sistem je neosetljiv na spoljašnje pobude, slaba reaktivnost
    - ◆ korisnički kod kritične sekcije ničim nije ograničen niti se garantovano završava u konačnom roku, pa može predugo ili večno zadržati sistem u "hibernaciji" (npr. beskonačna petlja)
  - Uposleno čekanje: neefikasno, troši procesorsko vreme na jalove instrukcije. – Da li je uvek tako?  
I blokiranje nešto "košta" – procesorsko vreme se opet troši na režijske (beskorisne) operacije (baratanje redovima, promena konteksta), što nekad nije zanemarljivo. Nije li uposleno čekanje ponekad efikasnije? Kada? Vrteti se ili blokirati se (*spin or block*)?

# Implementacija semafora

- ◆ Ideja: "vrteti se" (uposleno čekati) sve dok je to kraće nego vreme blokiranja, inače se blokirati?
- ◆ Kako znati koliko uposleno čekati, kada se blokirati?
- ◆ Neka je režijsko vreme blokiranja (uključujući i promenu konteksta) jednako  $b$ :
  - ako bi se uposleno čekalo za vreme  $t < b$  i onda dobio ključ, to je isplativije nego odmah se blokirati (uvek  $b$ )
  - ako bi se uposleno čekalo  $b$  vremena a onda blokiralo ako se ne dobije ključ, ukupno vreme je  $2b$
  - ako bi se samo uposleno čekalo, to može da potraje neodređeno ( $> 2b$ )
- ◆ Zaključak: sledeći postupak uvek garantuje režijsko vreme do  $2b$ :
  - uposleno čekati dok se ne dobije ključ ili dok ne istekne  $b$  vremena
  - potom se blokirati

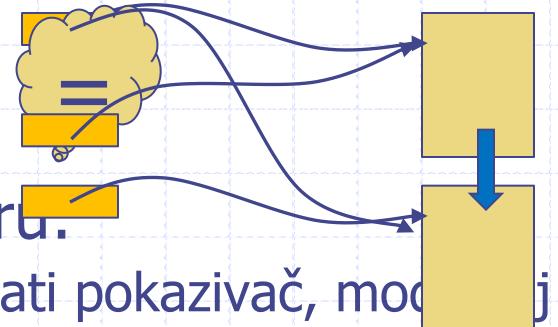
# Implementacija semafora

- ◆ Da li je zaključavanje neophodno?
- ◆ Zašto biti "pesimista" u pogledu međusobnog isključenja? Kolika je stvarna šansa da više procesa uleti u konflikt u kritičnoj sekciji, posebno ako ih je malo? Zašto trošiti toliko režijskog vremena na nešto što se možda retko ili nikad ne dešava?
- ◆ Drugi pristup – *optimistički pristup bez čekanja i zaključavanja (optimistic concurrency control, wait-free synchronization, lock-free synchronization)*:

**$x := x + 1$**

1. Učitaj  $x$  u registar R1
2. Dodaj 1 na R1 i smesti u registar R2
3. Atomično uradi: ako je i dalje  $x = R1$ , smesti R2 u  $x$ , inače idи на 1

# Implementacija semafora



- ◆ Moguće uopštiti na proizvoljnu strukturu.
  - Napravi kopiju strukture na koju ukazuje dati pokazivač, modificiši kopiju strukture
  - Atomično uradi: uporedi izvorni pokazivač sa pročitanim, ako su isti zameni pokazivač da ukazuje na kopiju
  - Ako je neki drugi proces izmenio polaznu strukturu, ponovi sve
- ◆ Potrebna je podrška procesora – instrukcija za atomično poređenje i upisivanje. Moderni procesori to imaju (x86: CMPXCHG, CMPXCHG8B)
- ◆ Prednosti:
  - nema režiskog vremena zbog zaključavanja i blokiranja
  - bolje se ponaša u slučaju otkaza (procesa koji drži nešto zaključano)
- ◆ Nedostatak: mnogo ponovnih pokušaja u slučaju velikog opterećenja
- ◆ Neko je napisao ceo OS bez zaključavanja!

# Upotreba semafora

- ◆ Međusobno isključenje pomoću semafora:

```
var mutex : Semaphore = 1; // Initially equal to 1
process P1;
loop
    wait(mutex);
    <critical section>
    signal(mutex);
    <non-critical section>
end
end P1;

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

Prednost: jednostavan kod, jednak za proizvoljan broj procesa

Nedostatak: *wait* i *signal* moraju biti upareni. Šta ako se greškom ne upare?

# Upotreba semafora

- ◆ Primer deljenog semafora: (pojednostavljeni) POSIX API

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
// Initialization:
const char* mutexName = "/myprogram_mutex"
sem_t* mutex = sem_open(mutexName,O_CREAT,1);
// O_CREAT|O_EXCL creates an exclusive semaphore
...
// Use for mutual exclusion:
sem_wait(mutex);
// Critical section
sem_post(mutex);
...
// Release the semaphore when it is no longer needed:
sem_close(mutex);
```

# Upotreba semafora

- ◆ Opšti slučaj sekcije u koju može da uđe N procesa:

```
var mutex : Semaphore = N; // Initially equal to N

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

# Upotreba semafora

- ◆ Uslovna sinhronizacija pomoću semafora:

```
var sync : Semaphore = 0; // Initially equal to 0

process P1; // Waiting process
...
wait(sync);
...
end P1;

process P2; // Signalling process
...
signal(sync);
...
end P2;
```

# Upotreba semafora

- ◆ Opšti slučaj  $n$  procesa P koji se sinhronizuju pomoću  $n-1$  semafora S:
  - P(1) se izvršava i signalizira S(1)
  - P( $k$ ) čeka na S( $k-1$ ), izvršava se i signalizira S( $k$ )
- ◆ (Veštački) primer: sinhronizovati procese koji zajedno obavljaju operaciju **print(f(x,y))**

```
float x,y,z;
```

```
Semaphore Sx(0) , Sy(0) , Sz(0) ;
```

```
T1:
```

```
x=...;
```

```
Sx.signal();
```

```
y=...;
```

```
Sy.signal();
```

```
...
```

```
T2:
```

```
Sx.wait();
```

```
Sy.wait();
```

```
z=f(x,y);
```

```
Sz.signal();
```

```
...
```

```
T3:
```

```
Sz.wait();
```

```
print(z);
```

```
...
```

# Upotreba semafora

- ◆ Ograničeni bafer pomoću semafora:

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

class BoundedBuffer {
public:

    BoundedBuffer ();

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

private:
    Semaphore mutex;
    Semaphore spaceAvailable, itemAvailable;

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

# Upotreba semafora

```
BoundedBuffer::BoundedBuffer () :  
    mutex(1), spaceAvailable(N), itemAvailable(0),  
    head(0), tail(0) {}  
  
void BoundedBuffer::append (Data* d) {  
    spaceAvailable.wait();  
    mutex.wait();  
    buffer[tail] = d;  
    tail = (tail+1)%N;  
    mutex.signal();  
    itemAvailable.signal();  
}  
  
Data* BoundedBuffer::take () {  
    itemAvailable.wait();  
    mutex.wait();  
    Data* d = buffer[head];  
    head = (head+1)%N;  
    mutex.signal();  
    spaceAvailable.signal();  
    return d;  
}
```

# Upotreba semafora

- ◆ Prikazani semafori nazivaju se još i *n*-arni ili *brojački* semafori
- ◆ Za mnoge primene (npr. međusobno isključenje) dovoljni su *binarni* semafori: najveća vrednost mu je 1
- ◆ Operacija *wait* blokira proces ako je semafor 0, odnosno postavlja semafor na 0 ako je bio 1
- ◆ Operacija *signal* deblokira proces ako čeka, a postavlja semafor na 1 ako ga nema
- ◆ Ponegde se binarni semafor naziva i *događaj (event)* – samo signalizira da se neki događaj desio (logička, Bulova vrednost, desio se ili ne)
- ◆ U mnogim sistemima na događaj može čekati samo jedan proces ("vlasnik" događaja) i samo on može vršiti *wait*, *signal* može uraditi bilo koji proces
- ◆ Neki sistemi podržavaju kombinovano čekanje na više događaja po uslovu "i" i "ili"
- ◆ Posebno pogodni za čekanje na spoljašnje događaje koji se signaliziraju iz prekidne rutine: završena I/O operacija, isteklo vreme suspenzije procesa. Zato su veoma osetljivi za implementaciju

# Upotreba semafora

## ◆ Pogodnosti semafora:

- jednostavan i efikasan koncept
- generalan koncept niskog nivoa – pomoću njega se mogu implementirati mnogi drugi, apstraktniji koncepti za sinhronizaciju

## ◆ 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*

# Upotreba semafora

- ◆ Šta je problem sa sledećim kodom (S1 i S2 su inicijalno 1)?

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

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

Moguća sekvenca:

```
P1 - wait(S1); // S1 := 0  
P2 - wait(S2); // S2 := 0  
P1 - wait(S2); // P1 blocks on S2  
P2 - wait(S1); // P2 blocks on S1
```

Problem - *mrtvo* (ili *kružno*) *blokiranje* (*deadlock*):  
stanje sistema u kome je nekoliko procesa  
suspendovano (blokirano) međusobnim  
uslovljavanjem (čekanjem) - detalji u OS2

# IV Upravljanje memorijom

Vezivanje adresa

Deljenje memorije

Organizacija i alokacija memorije

Virtuelna memorija

# Glava 10: Vezivanje adresa

Problem vezivanja adresa (*address binding*)

Prevođenje (*compilation*)

Povezivanje (*linking*)

Učitavanje (*loading*)

Dinamičko preslikavanje adresa

# Problem vezivanja adresa

- ◆ Izvorni program na jeziku C/C++ podeljen na fajlove:

```
// A.cpp:  
int a = 3;  
void f() { ... }
```

```
// B.cpp:  
extern int a;  
extern void f();  
void g() {  
    ...a++; ...f()...;  
}
```

- ◆ Pitanja:

- Kako preslikati obraćanje promenljivoj **a** ili funkciji **f** u mašinski "razumljivo" adresiranje – prevodenje (*compilation*)?
- Kako povezati prevedene fajlove u jedinstven program i rešiti obraćanje promenljivoj **a** ili funkciji **f** u drugom fajlu – povezivanje (*linking*)?
- Kako odrediti adrese promenljive **a** i funkcije **f** kada se sazna gde će proces biti smešten u memoriji – učitavanje (*loading*)?
- Postoji li potreba za preslikavanjem logičkih u fizičke adrese u vreme izvršavanja, uz pomoć hardvera – dinamičko preslikavanje adresa?

# Prevodenje (*compilation*)

A.cpp

```
int a = 3;  
  
void f() {  
    ...  
}
```

A.obj

```
↑a: 0  
↑f: 1  
...  
a: 3  
f: ....  
....  
...
```

```
extern int a;  
void f();  
  
void g()  
{  
    ...f()  
    ...a...  
}
```

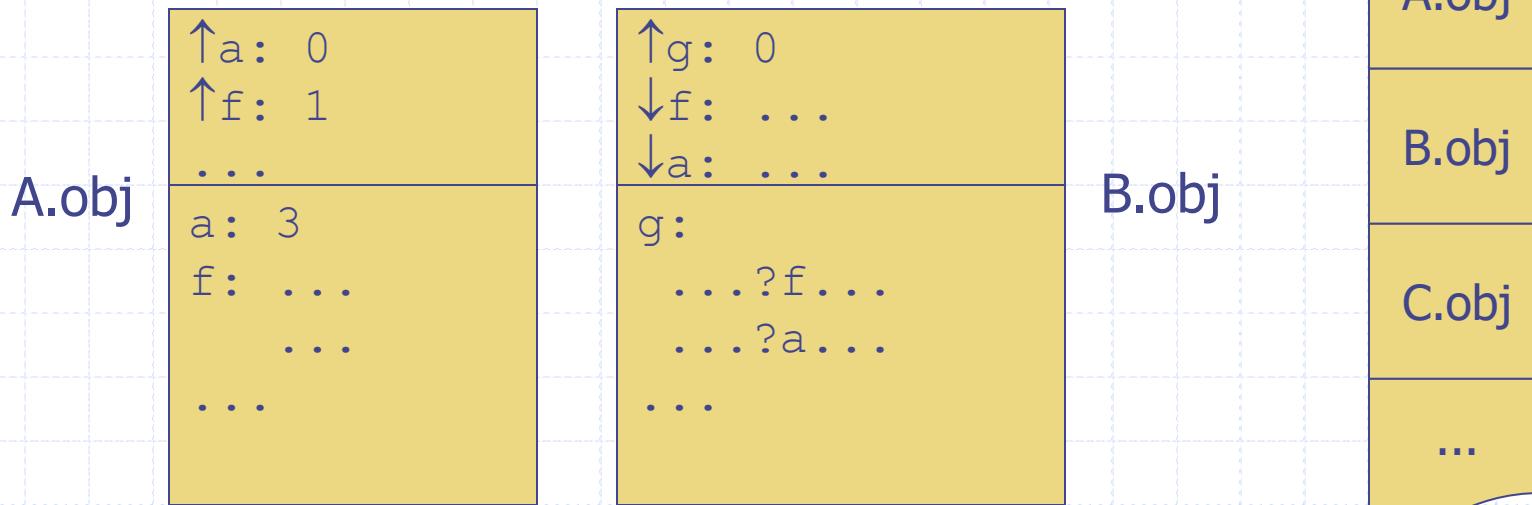
Ovo nije definicija  
koja brižljivo uključuje  
alokaciju prostora

B.cpp

```
↑g: 0  
↓f: ....  
↓a: ....  
g: ?  
....?f...  
....?a...  
...
```

B.obj

# Povezivanje (*linking*)



- ◆ Linker ima zadatak da sastavi niz .obj fajlova (A.obj, B.obj, C.obj itd.) i napravi jedan izvršni fajl (P.exe)
- ◆ Linker ovo tipično radi u dva prolaza:
  - pravi globalnu mapu .obj fajlova i tabelu definisanih izveženih simbola i njihovih globalnih adresa (relativno u odnosu na početak .exe fajla)
  - razrešava referenciranja uveženih simbola koristeći adrese definisanih simbola izračunate u tabeli simbola

# Povezivanje (*linking*)

- ◆ *Biblioteke (libraries, .lib)* imaju isti oblik i značenje kao i obični .obj fajlovi, osim što su pripremljeni prevodenjem i povezivanjem skupa izvornih fajlova
- ◆ Linker tretira biblioteke na isti način kao i druge .obj fajlove
- ◆ Moguće greške tokom povezivanja:
  - simbol nije definisan: linker saopštava samo ime .obj fajla koji uvozi (referiše) nedefinisani simbol; često zbunjuje, pošto korisnički kod možda uopšte ne koristi taj simbol; verovatan uzrok: simbol se koristi u nekoj povezanoj biblioteci, ali druga biblioteka u kojoj je taj simbol definisan nije uključena u listu za povezivanje
  - višestruke definicije simbola: linker saopštava imena .obj fajlova koji definišu isti simbol; verovatan uzrok: definicije u izvornim fajlovima i/ili biblioteci ("sukob imena", *name clashing*)
- ◆ Povezivanje sa ciljem pravljenja biblioteke (.lib) razlikuje se od povezivanja sa ciljem pravljenja programa (.exe)! Zašto?

# Učitavanje (*loading*)

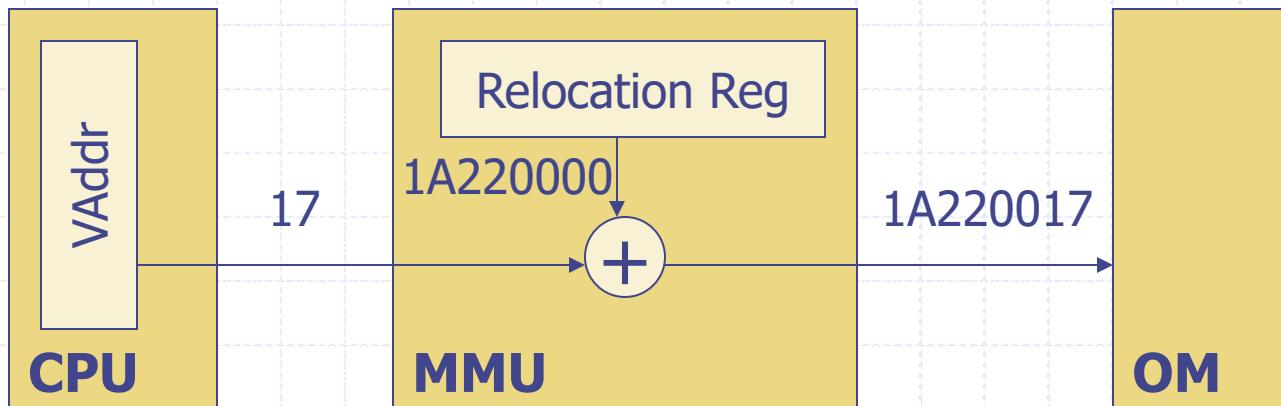
- ◆ Procesi čije je započinjanje zahtevano (kreirani su) nalaze se u redu za započinjanje (*job queue, input queue*) – na spisku su svih procesa, ali još nisu spremni za izvršavanje
- ◆ Da bi ovakav proces bio spreman za izvršavanje i prešao u *ready queue*, OS treba sa njim da uradi sledeće:
  1. pronađe slobodan prostor u memoriji za smeštanje programa i podataka procesa
  2. učita program i statički alocirane podatke u memoriju sa diska
  3. (ako je potrebno) razreši adrese referisanih simbola u programu – relativne adrese u odnosu na početak programa pretvori u absolutne memorijske fizičke adrese, prostim dodavanjem absolutne fizičke adrese smeštanja procesa u memoriju – statičko vezivanje adresa u vreme učitavanja
  4. kreira početni kontekst
- ◆ Ove poslove obavlja poseban deo OS, sistemski program pod nazivom *loader* ("utovarivač")

# Dinamičko preslikavanje adresa

- ◆ Ako OS omogućava postojanje više procesa u memoriji u jednom trenutku i izbacivanje nekih (delova) procesa a ubacivanje drugih, onda staticko vezivanje adresa u vreme učitavanja nije dovoljno
- ◆ Potrebno je da proces bude *relokabilan*: da se može jednostavno, u vreme izvršavanja programa, promeniti lokacija procesa (programa i njegovih podataka) na drugo mesto u memoriji, a da to ni na koji način ne utiče na program i njegovo adresiranje podataka (pokazivači!)
- ◆ Potrebno je zato da proces poseduje *logički (virtuelni) adresni prostor*, tako da program može da adresira sve adrese u opsegu 0 do Max
- ◆ Logičke adrese se preslikavaju u *fizičke adrese* koje se upućuju memorijskim modulima
- ◆ Preslikavanje logičkih u fizičke adrese obavlja se *dinamički, u vreme izvršavanja*
- ◆ Ovo preslikavanje vrši *hardver* – obavezna je hardverska podrška
- ◆ Ovo preslikavanje vrši se uvek i za svaku logičku adresu – program “vidi” isključivo logički adresni prostor, nikako fizički

# Dinamičko preslikavanje adresa

- ◆ Najjednostavnija tehnika dinamičkog preslikavanja adresa: bazni register za relokaciju (*base relocation register*):



- ◆ Zaduženja OS-a:
  - Loader treba da definiše vrednost relokacionog registra na osnovu mesta učitavanja procesa u memoriji i smesti tu vrednost u PCB
  - prilikom svake promene konteksta, OS treba da upiše vrednost iz PCB u relokacioni register
- ◆ Složenije tehnike dinamičkog preslikavanja: stranična, segmentna i stranično-segmentna organizacija

# Glava 11: Deljenje memorije

Problem deljenja memorije

Dinamičko učitavanje (*dynamic loading*)

Preklopi (*overlays*)

Biblioteke sa dinamičkim vezivanjem (DLL)

Zamena (*swapping*)

# Problem deljenja memorije

- ◆ Više procesa konkuriše za isti resurs. Kako ih sve zadovoljiti? Deljenjem resursa!
- ◆ Kako se može deliti resurs:
  - vremenski (*time sharing*): jedno vreme resurs koristi jedan proces, pa ga onda preuzima (*preempt*) i malo koristi drugi itd.; primer: CPU
  - prostorno (*space sharing*): jedan deo resursa koristi jedan proces, drugi deo drugi proces itd.; primer: operativna memorija
- ◆ Operativna memorija se može deliti i vremenski i prostorno:
  - samo vremenski: celu memoriju koristi samo jedan proces neko vreme, pa onda celu memoriju koristi neki drugi; veoma stari OS
  - samo prostorno: ceo proces je u OM, ali je u datom trenutku više procesa u OM; uglavnom stariji i slabiji OS, bez izbacivanja delova ili celih procesa
  - kombinacija: delovi procesa ili celi procesi se izbacuju i učitavaju se drugi, u jednom trenutku je u OM više delova ili celih procesa; moderni OS

# Problem deljenja memorije

Ukoliko više procesa treba da koristi memoriju, kako rešiti problem nedostatka (ograničenja) memorije? Tehnike:

- smanjiti potrebu za memorijom od strane programa bez učešća OS-a - prevodilac obezbeđuje da program smanji svoje potrebe:
  - ◆ dinamičko učitavanje (*dynamic loading*)
  - ◆ preklopi (*overlays*)
- prostorno deliti memoriju:
  - ◆ deljene biblioteke (*shared libraries*) i dinamičko vezivanje (*dynamic linking*)
  - ◆ učitavati više procesa u memoriju; neophodan uslov: relokabilnost
- vremenski deliti memoriju:
  - ◆ zamena (*swapping*)
  - ◆ virtualna memorija (*virtual memory*)

# Dinamičko učitavanje

## ◆ Ideja:

- složeni program često nikada ne izvršava neke svoje delove ili ne koristi neke svoje podatke; neke procedure se možda nikada ili jako retko pozivaju, npr. kod za obradu grešaka i izuzetnih situacija; neki podaci se nikada ili retko koriste
- zašto bezuslovno unapred učitavati ove delove programa, ako oni nikada neće biti korišćeni?
- ovakve delove učitavati *dinamički*, tek u vreme izvršavanja, po potrebi – kada im se prvi put pristupi

# Dinamičko učitavanje

## ◆ Realizacija:

- prevodilac obezbeđuje potrebnu podršku i rastavlja program na potprograme kao odvojene celine za učitavanje; može zahtevati sugestiju i pomoć programera
- potprogrami se smeštaju na disk kao relokabilne celine
- pri pokretanju procesa, učitava se glavni program i programska tabela adresa potprograma
- kada se pozove potprogram, pozivajući kod proverava u tabeli da li je potprogram učitan
- ako nije, pokreće se loader (kao deo programa, ne OS-a) da učita potprogram na određenu lokaciju i postavi njegovu adresu u tabelu potprograma
- kada se potprogram ponovo pozove, njegova adresa je u tabeli spremna za poziv (indirektno adresiranje potprograma)

◆ Dinamičko učitavanje ne zahteva nikakvu podršku OS-a – sve radi prevodilac i generisani kod. OS samo obezbeđuje bibliotečne rutine za dinamičko učitavanje

# Preklopi

## ◆ Ideja:

- zašto ne omogućiti da program alocira manje prostora nego što u celini zauzima, tako što se delovi koji se ne koriste u isto vreme zamenjuju u memoriji, zauzimajući isto mesto - *preklapajući se (overlaping)*
- kada zatreba neki deo koji nije u memoriji, izbaciti deo sa kojim se preklapa i na njegovo mesto učitati potrebni

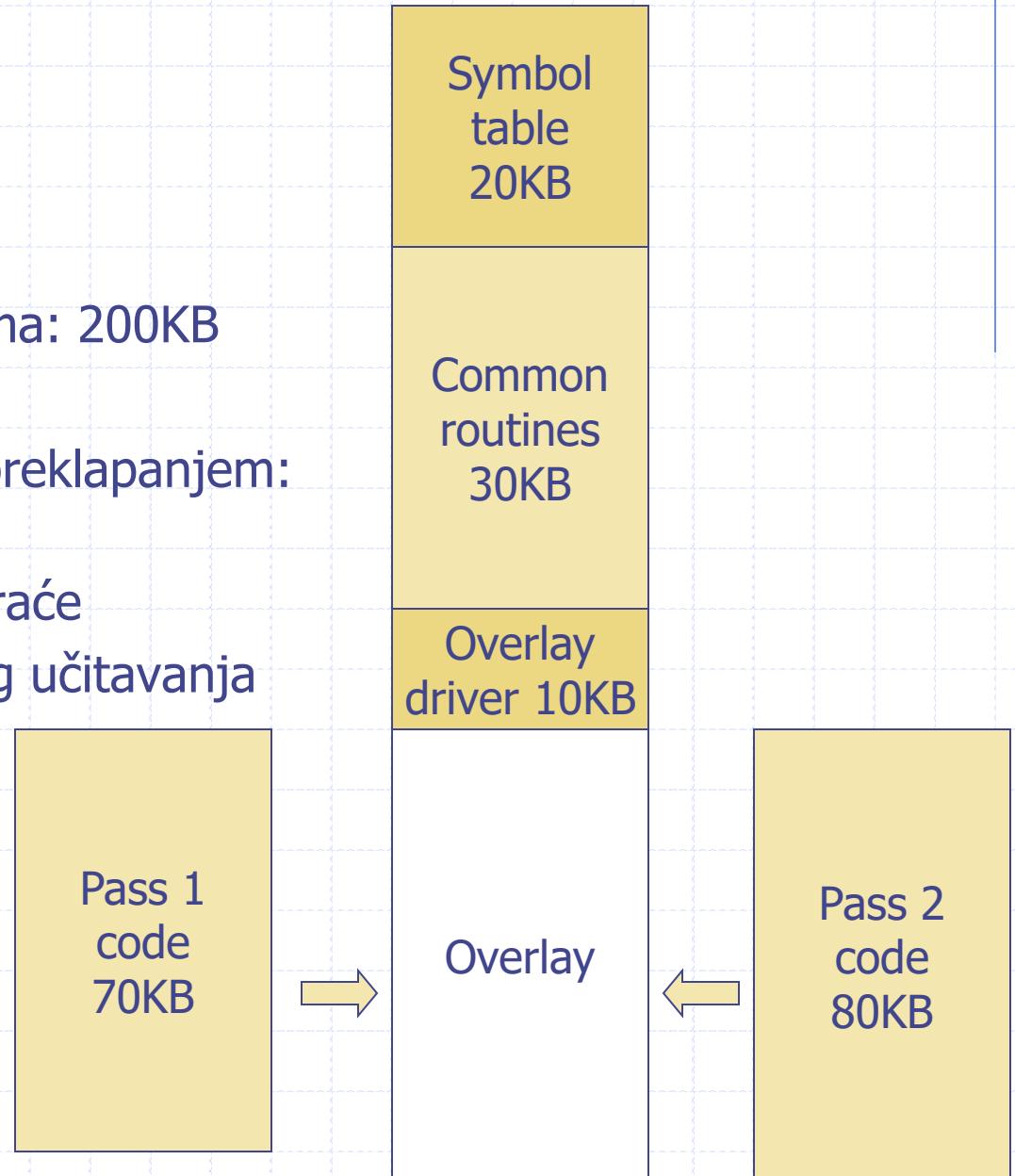
## ◆ Realizacija:

- prevodilac obezebeđuje potrebnu podršku i rastavlja program na delove kao odvojene celine za učitavanje; može zahtevati sugestiju i pomoć programera
- delovi se smeštaju na disk kao relokabilne celine
- pri pokretanju procesa, učitavaju se određeni delovi
- kada se zahteva neki deo, pozivajući kod poziva *overlay driver* koji proverava da li je deo učitan
- ako nije, pokreće se loader (kao deo programa, ne OS-a) da učita deo na određenu lokaciju, zamenjujući neki drugi deo

# Preklopi

## ◆ Primer:

- linker sa dva prolaza
- ukupna veličina programa: 200KB
- *overlay driver*: 10KB
- potrebna memorija sa preklapanjem: 140KB
- početno učitavanje je kraće
- izvršavanje je duže zbog učitavanja



# Preklopi

- ◆ Preklapanje ne zahteva nikakvu podršku OS-a – sve radi prevodilac (uz pomoć programera) i generisani kod. OS samo obezbeđuje bibliotečne rutine za dinamičko učitavanje
- ◆ Ali – da li preklapanje i niti (*overlays+threads*) mogu zajedno?
- ◆ Problem: za velike programe (za male preklapanje uglavnom nije potrebno) konstruisanje preklopa zahteva razumevanje značenja programa, što može da bude složeno
- ◆ Zbog toga je preklapanje ograničeno na mikroračunare sa manjom količinom fizičke memorije i bez hardverske podrške za složenije i bolje mehanizme upravljanja memorijom

# Biblioteke sa dinamičkim vezivanjem

## ◆ Ideja:

- mnogi programi često koriste iste sistemske biblioteke, npr. za datoteke, GUI, mrežnu komunikaciju itd.
- ako bi se te biblioteke statički povezivale, svaki program koji se izvršava zahtevao bi prostor unutar sebe za te biblioteke, a isti kod bi se ponavljao u svim programima
- zašto ne obezbediti *deljenje biblioteka (shared libraries)*, tako da u memoriji postoji samo jedna kopija koda biblioteka
- potrebno je vršiti *dinamičko povezivanje (dynamic linking)* u vreme izvršavanja umesto statičkog povezivanja pre izvršavanja
- ideja slična dinamičkom učitavanju, osim što se umesto učitavanja povezivanje odlaže za vreme izvršavanja
- deljene biblioteke sa dinamičkim vezivanjem (*dynamic linking libraries, DLL*)

# Biblioteke sa dinamičkim vezivanjem

## ◆ Realizacija:

- u programu koji koristi DLL, za svaki potprogram iz DLL-a uvodi se *stub* – “kaobajagi” potprogram, njegov zamenik
- pozivi datog potprograma prevode se uobičajeno, ali oni pozivaju *stub* umesto pravog potprograma P
- *stub* u sebi sadrži referencu (adresu) stvarnog potprograma P iz biblioteke, inicialno postavljenu na *null*
- *stub* je mali deo koda koji radi sledeće:
  - ◆ ako je referenca na potprogram *null*, poziva se sistemska usluga koja pronalazi traženi DLL i njegov potprogram P i vraća njegovu adresu
  - ◆ vraćenu vrednost smešta u svoju referencu, tako da se svaki sledeći poziv odmah razrešava adresiranjem preko reference, bez sistemskog poziva

# Biblioteke sa dinamičkim vezivanjem

```
X proc (A a, B b, C c) { // stub for proc
    static X (*ref)(A,B,C) = 0;
    if (ref==0) {
        ref = sys_call(MapDLL, dllName, "proc");
        if (ref==0) ... // Exception!
    }
    return (*ref)(a,b,c);
}
```

- sistemska usluga treba da uradi sledeće:
  - ◆ u sistemskom registru DLL-ova pronađe traženi DLL i traženi potprogram
  - ◆ učita DLL u memoriju ako već nije učitan na zahtev nekog drugog procesa
  - ◆ (ako je potrebno) preslika fizički prostor koji zauzima DLL u virtuelni prostor pozivajućeg procesa i vrati virtuelnu adresu potprograma – neophodna podrška OS-a

# Biblioteke sa dinamičkim vezivanjem

- ◆ Pogodnost: prilikom izdavanja nove verzije biblioteke (npr. ispravka greške), programi koji je koriste ne moraju ponovo da se staticki povezuju
- ◆ Problem: šta ako nova verzija biblioteke nije više kompatibilna sa starim programima?
- ◆ Neophodno označavanje verzija biblioteka i provera kompatibilnosti biblioteke koja postoji u sistemu sa verzijom koju program zahteva
- ◆ Posledica: moguće je da u datom trenutku u sistemu postoji učitano više verzija iste biblioteke

# Zamena

- ◆ Ideja: vremenski deliti memoriju tako što se jedan proces izbaci iz nje snimanjem na disk da bi drugi bio učitan sa diska – *zamena (swapping)*
- ◆ Jednostavna varijanta: kada dođe do preuzimanja, OS započinje snimanje sadržaja memorije procesa koji je izgubio procesor na disk i učitavanje drugog na njegovo mesto, dok u međuvremenu procesor izvršava neki treći proces koji je spremam
- ◆ Da li se proces koji se učitava u memoriju vraća na isto mesto iz koga je izbačen? Zavisi! Od čega?
- ◆ Disk za podršku zameni mora biti brz i velikog kapaciteta da prihvati cele adresne prostore više procesa
- ◆ Prostor na disku za podršku zameni je van fajl sistema OSa da bi bio brz

# Zamena

- ◆ Posledica: promena konteksta može biti veoma duga
- ◆ Na primer, neka je brzina transfera diska 5MB/s; za zamenu procesa koji zauzima 1MB memorije potrebno je 2 puta po oko 0,2s, dakle oko 0,4s; šta ako proces zauzima 128MB? Zato je potrebno da OS zna koliko proces stvarno zauzima, ne koliko može da zauzima
- ◆ Da bi sistem bio efikasan, vremenski kvant dodeljen procesu za izvršavanje mora biti značajno duži od promene konteksta
- ◆ Problem: šta ako proces koji se izbacuje čeka na završetak I/O operacije koja koristi deo memorije tog procesa za svoje bafere? Rešenja:
  - zabraniti zamenu procesa koji čeka na završetak I/O
  - koristiti memorijski prostor OS-a za I/O bafere

# Zamena

- ◆ Malo sistema koristi jednostavnu varijantu stalne zamene celog procesa prilikom svake promene konteksta; mnogo više se koriste modifikovane varijante
- ◆ Unix modifikacija: zamena je normalno onemogućena, ali se aktivira kada sistem postane opterećen sa mnogo procesa
- ◆ MS Windows 3.1 varijanta: kada se pokrene novi proces a nema dovoljno memorije, postojeći proces se zamenjuje; ovde korisnik, a ne OS inicira preuzimanje i zamenu: kada korisnik pređe da radi sa nekim procesom, on se aktivira, po potrebi zamenjujući neki drugi proces; izbačeni proces ostaje van memorije sve dok korisnik ne pređe na njega

# Glava 12: Organizacija i alokacija memorije

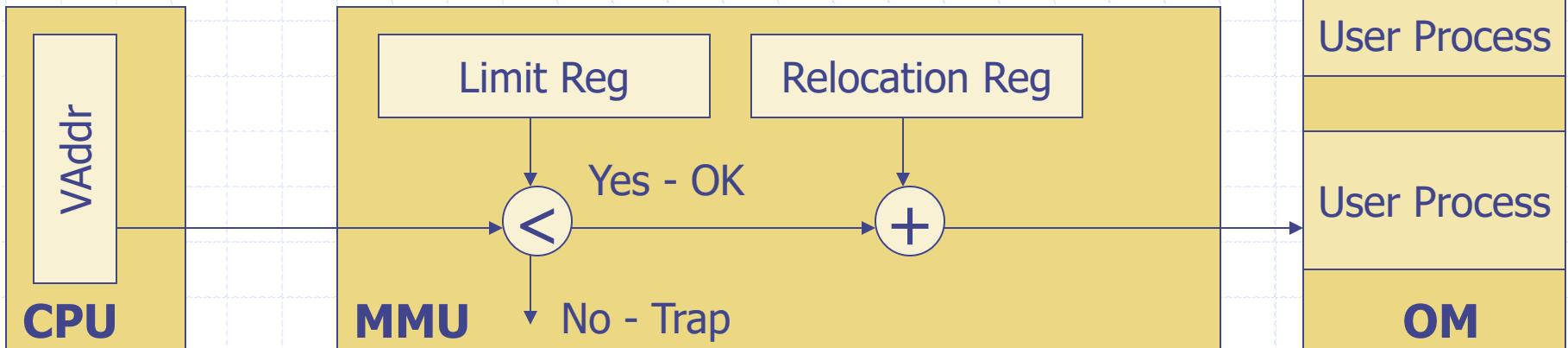
Kontinualna alokacija

Stranična organizacija

Segmentna organizacija

# Kontinualna alokacija

- ◆ Za svaki proces alocira se kontinualan memorijski prostor
- ◆ Relokatibilnost se obezbeđuje dinamičkim preslikavanjem sa registrom za relokaciju (*relocation register*)
- ◆ Ovakvo preslikavanje obezbeđuje i zaštitu OS-a od korisničkih procesa i procesa između sebe – registar granice (*limit register*)
- ◆ Ova dva regista sastavni su deo konteksta procesa – OS ih učitava iz PCB pri promeni konteksta



# Kontinualna alokacija

- ◆ Osnovno pitanje: kako pronaći slobodan prostor za smeštanje procesa – problem *alokacije (allocation)*?
- ◆ Jedno jednostavno rešenje – particije (*partition*): podeliti OM za korisničke procese na particije jednakе veličine, maksimalno dozvoljene procesu. Svaka particija može da sadrži samo jedan proces – stepen multiprogramiranja je ograničen brojem particija
- ◆ OS vodi tabelu particija – za svaku particiju po jedan ulaz. Ulaz sadrži informaciju da li je particija slobodna, odnosno kom procesu je dodeljena. PCB sadrži oznaku particije procesa
- ◆ Kada ima slobodne particije, proces se učitava i pokreće. Kada proces završi, njegova particija se označava slobodnom
- ◆ Zastareo metod, ne primenjuje se više

# Kontinualna alokacija

## ◆ Opštiji pristup - dinamička alokacija memorije (*dynamic storage allocation*):

- OS vodi evidenciju o zauzetim i slobodnim delovima memorije
- inicijalno je ceo prostor za korisničke procese slobodan
- kada se kreira novi proces, stavlja se u ulazni red (*input queue*)
- OS analizira memorijske zahteve procesa u ulaznom redu i trenutno stanje slobodne memorije i bira proces za učitavanje (na osnovu algoritma raspoređivanja ili memorijskih zahteva)
- OS pronalazi slobodan deo memorije dovoljno veliki da smesti izabrani proces
- OS učitava proces u izabrani deo slobodne memorije; deo memorije koji proces zauzima označava zauzetim, a ostatak ostaje slobodan
- kada se proces završi, deo memorije koji je zauzimao proglašava se slobodnim, uz eventualno spajanje u kontinualnu celinu sa slobodnim delovima ispred i iza oslobođenog dela
- OS može da ispita postoji li proces u ulaznom redu koji sada može da se učita

# Kontinualna alokacija

- ◆ Osnovno pitanje: ukoliko postoji više slobodnih delova u koje može da se smesti proces, kako izabrati jedan?
- ◆ Pristupi:
  - izabrati prvi koji odgovara (*first fit*): jednostavan i efikasan
  - izabrati onaj koji najbolje odgovara (*best fit*), u smislu da ostaje najmanje "otpatka", sa idejom najboljeg iskorišćenja prostora
  - izabrati onaj koji najlošije odgovara (*worst fit*), u smislu da najviše preostaje, sa idejom da se taj ostatak najlakše može iskoristiti
- ◆ Eksperimenti pokazuju da su *first fit* i *best fit* bolji u smislu efikasnosti i iskorišćenja memorije; ni jedan od ta dva nije generalno bolji u smislu iskorišćenja, a *first fit* je generalno efikasniji

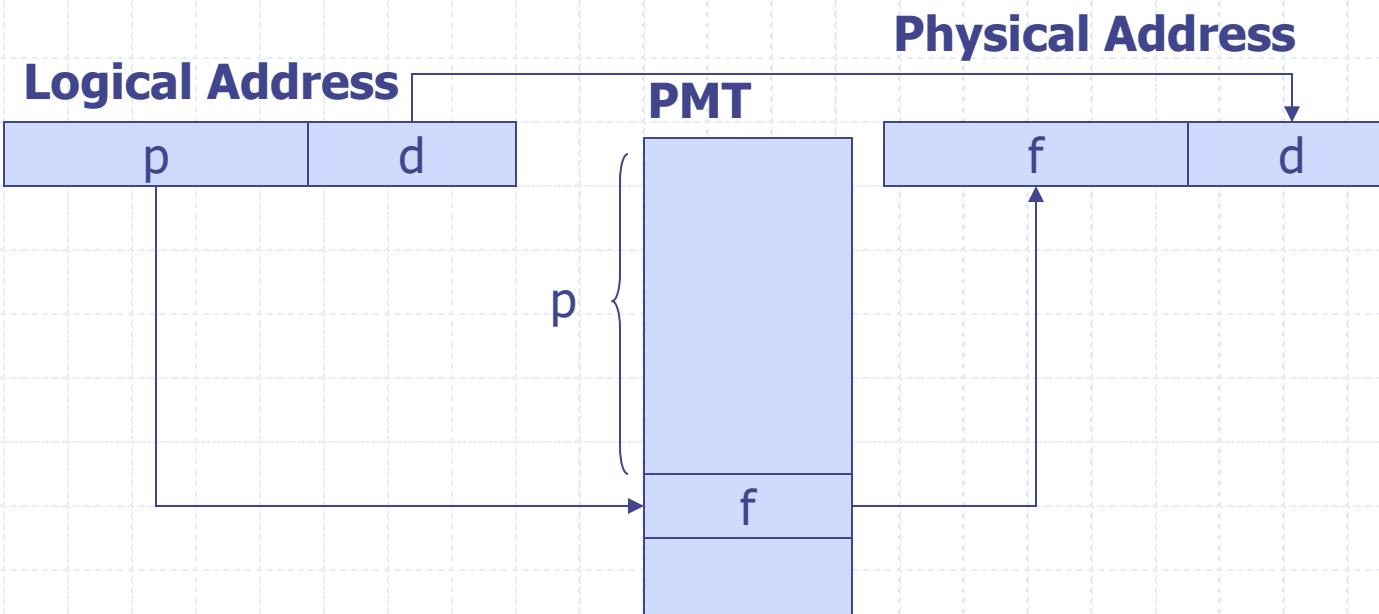
# Kontinualna alokacija

- ◆ Osnovni problem - eksterna fragmentacija (*external fragmentation*): ukupan slobodni prostor jest dovoljan, ali nije kontinualan za alokaciju
- ◆ Statistike: na  $N$  alociranih blokova, još oko  $\frac{1}{2}N$  blokova se gubi zbog fragmentacije
- ◆ Uzrok: alokacija delova različitih veličina u slobodne celine različitih veličina
- ◆ Još jedan problem: šta ako se alocira prostor od 18462 bajta u slobodan deo od 18464 bajta – preostaju samo 2 slobodna bajta. Troškovi vođenja evidencije o tako malom slobodnom prostoru nisu opravdani!

# Kontinualna alokacija

- ◆ Pristupi rešavanju eksterne fragmentacije:
  - podeliti memoriju na blokove fiksne veličine i alocirati memoriju samo u celom broju blokova; razlika između stvarno zahtevane memorije i alocirane memorije – neiskorišćen ostatak – *interna fragmentacija*
  - kompakcija (*compaction*): relocirati procese tako da se spoje zauzeti i slobodni delovi memorije – “skupiti” slobodnu memoriju u jedan veliki blok; moguće samo ako je relokacija dinamička, nije moguće za statičku relokaciju (u vreme učitavanja); problem: skupo rešenje
  - stranična organizacija (*paging*): omogućiti da logički adresni prostor procesa ne bude kontinualan

# Stranična organizacija



- ◆ Tipične veličine stranice: od 512 do 16M adresibilnih jedinica
- ◆ Inherentno obezbeđuje dinamičku relokabilnost
- ◆ Nema eksterne fragmentacije
- ◆ Ima interne fragmentacije. Ako je veličina procesa nezavisna od veličine stranice, očekivana veličina internog fragmenta je  $\frac{1}{2}$  stranice
- ◆ Taj razlog favorizuje male stranice. Međutim, male stranice unose više rezija (veća PMT, manje efikasan I/O) – pogododnije su veće stranice

# Stranična organizacija

- ◆ Posledica: svaki proces "vidi" kontinualan logički (virtuelni) adresni prostor sa samo jednim programom, dok je u fizičkoj memoriji taj prostor rasut po okvirima (*frame*, blokovima)
- ◆ Ova razlika je transparentna za korisnički proces – sakriva ga hardver za preslikavanje adresa (MMU) i OS
- ◆ Kada se proces veličine  $N$  stranica izabere za učitavanje u memoriju, OS pronalazi  $N$  slobodnih okvira za smeštanje procesa
- ◆ Zbog toga OS mora da vodi tabelu zauzetosti okvira fizičke memorije (*frame table*): za svaki okvir jedan ulaz – da li je okvir slobodan ili je zauzet i koji proces ga zauzima
- ◆ OS mora da vodi računa o tome da su adrese koje proces prenosi kao argumente sistemskog poziva logičke. OS mora da ih preslika u fizičke adrese korišćenjem PMT za dati proces

# Stranična organizacija

- ◆ Ako je PMT u fizičkoj memoriji, svaki pristup jednoj logičkoj lokaciji zahteva dva pristupa fizičkoj memoriji – pristup memoriji se dvostruko produžava
- ◆ Da li PMT može da se smesti u HW za preslikavanje adresa (MMU)?
- ◆ Teorijski - da. Posledice:
  - cela PMT je deo konteksta procesa koji mora da se učita u MMU pri promeni konteksta
  - registarski fajl za smeštanje PMT treba da bude brz
- ◆ Međutim, takav registarski fajl za novije sisteme bio bi neizvodljivo veliki – reda 1M registara (ulaza u PMT)
- ◆ Zbog toga noviji sistemi čuvaju PMT u operativnoj memoriji, a na PMT ukazuje PMTP registar – deo konteksta
- ◆ Kako rešiti problem neefikasnog pristupa OM? TLB!

# Stranična organizacija

## ◆ Moderni računari:

- veoma veliki logički adresni prostor:  $2^{32}$  do  $2^{64}$  lokacija
- neka je veličina stranice 4K lokacija ( $2^{12}$ ), sledi:
- PMT ima  $2^{20}$  do  $2^{52}$  ulaza!
- ako je ulaz 4B (32 bita), što omogućava  $2^{32}$  fizičkih okvira, odnosno  $2^{44}$  fizičkih lokacija, sledi:
- PMT je veličine  $2^{22}$  (4M) do  $2^{54}$  bajtova!
- Ovo može biti neizvodljivo!
- Ali na sreću, često nije ni potrebno jer proces praktično nikada ne koristi ceo svoj logički adresni prostor

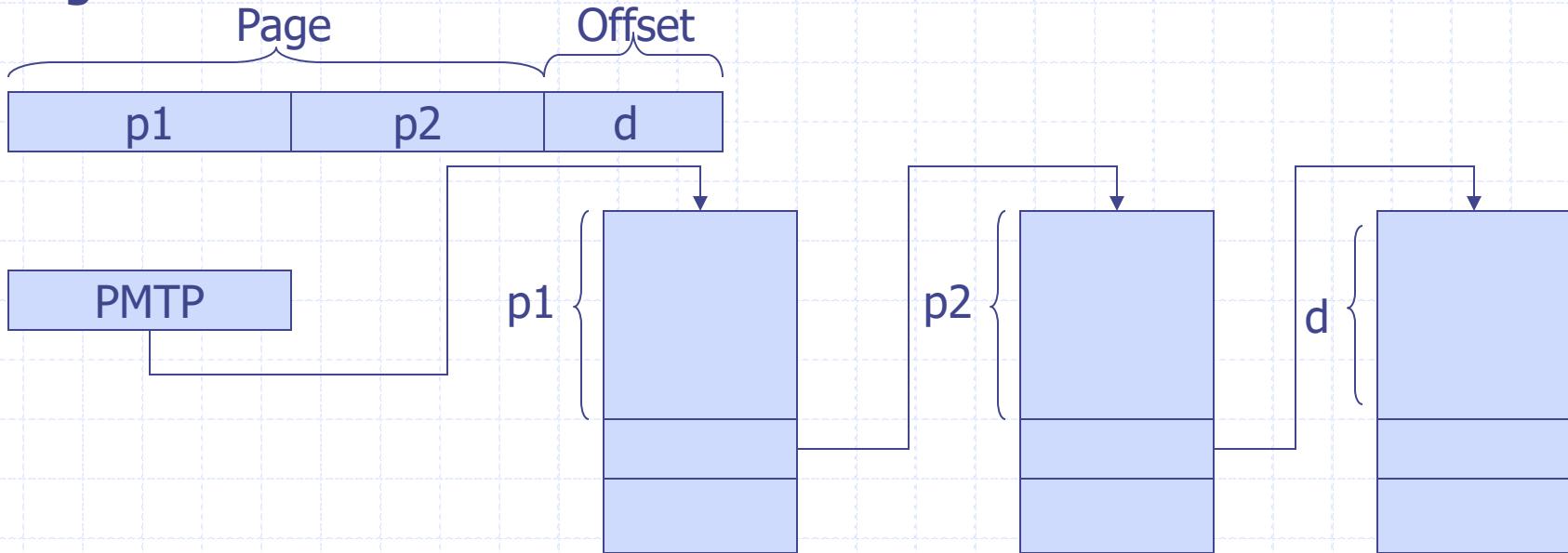
## ◆ Jedno rešenje: PMT implementirana kao *hash* tabela

◆ Ideja: veliki prostor ključeva (*page*) sa malo njih koji su iskorišćeni - treba imati preslikavanje ključeva u ulaze *hash* tabele koje "rasipa" ključeve koji se pojavljuju po tabeli, uz rešavanje konflikata (više ključeva se preslikava u isti ulaz) asocijativnim traženjem ključa u istom ulazu

# Stranična organizacija

- ◆ Drugo rešenje: *straničenje u više nivoa (multilevel paging)*
- ◆ Ideja: samu PMT podeliti na stranice i te stranice (tj. samo one potrebne) rasuti po fizičkoj memoriji
- ◆ Straničenje u dva nivoa:

**Logical Address**

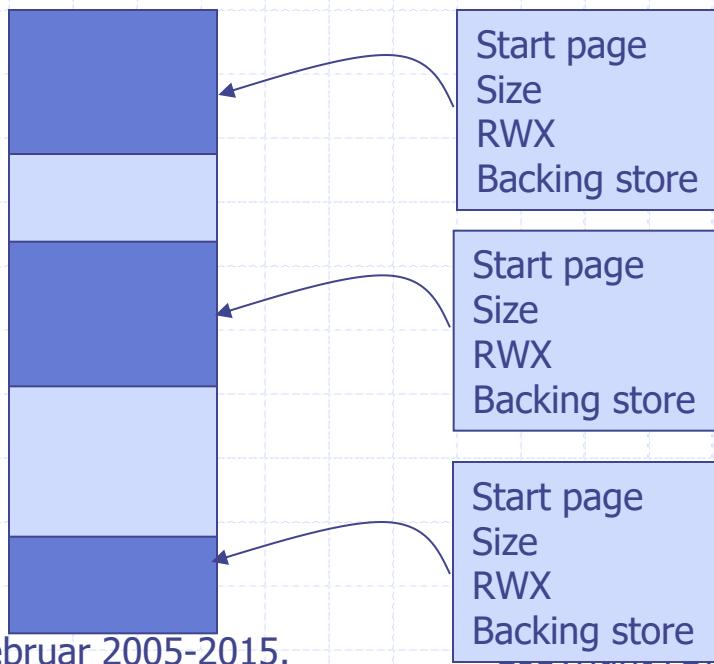


# Stranična organizacija

- ◆ Pojam *logičkog segmenta* virtuelnog adresnog prostora – deo virtuelnog adresnog prostora koji proces koristi na isti način (ista namena, npr. kod, stek, statički podaci, *heap*)
- ◆ Proces mora da *allocira* segment (deklariše OS-u da koristi određeni segment) na jedan od dva načina:
  - Statički: segmenti su definisani zapisima u .exe fajlu; *loader* ih tumači prilikom pokretanja procesa
  - Dinamički: tokom svog izvršavanja, proces allocira novi segment sistemskim pozivom (opseg adresa, tj. stranica, način korišćenja)
- ◆ Za svaki alocirani segment, OS vodi evidenciju u *deskriptoru* segmenta
- ◆ Loader kreira ove strukture prilikom kreiranja procesa (kreiranje memorijskog konteksta)

# Stranična organizacija

- ◆ Deskriptori segmenata – struktura isključivo pod kontrolom OSa i odvojena od PMT, na osnovu koje OS održava PMT:
  - Opseg adresa (stranica)
  - Način korišćenja (prava pristupa, RWX – određuju prava u deskriptoru svake stranice tog segmenta u PMT)
  - Odakle potiče i gde se snima (*Backing store*: exe fajl/swap space)

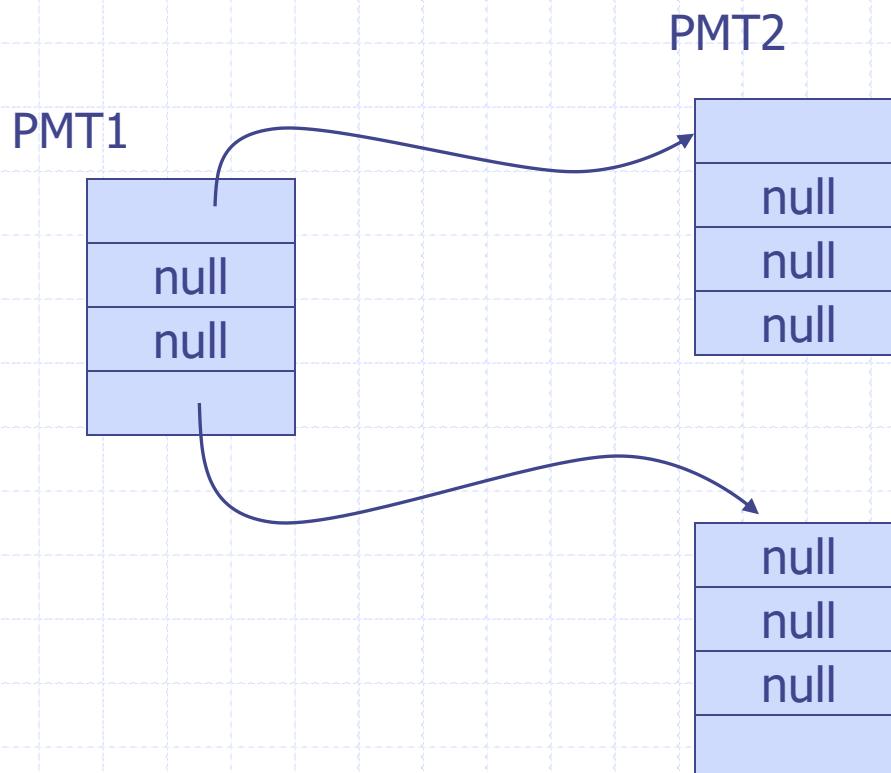


# Stranična organizacija

- ◆ Deskriptor stranice u PMT sadrži samo informacije potrebne hardveru:
  - Prava pristupa (RWX)
  - Da li je preslikavanje moguće (npr. *null* vrednost ako nije), bez obzira zašto nije – zato što stranica nije validna, jer ne pripada alociranom segmentu, ili zato što nije u memoriji
  - Ako jeste, u koji okvir se preslikava (*frame*)
- ◆ Ako preslikavanje nije moguće, hardver generiše *page fault* ne ulazeći u razlog. Kada obrađuje ovaj izuzetak, OS utvrđuje razlog na osnovu virtualne adrese i podataka u deskriptoru segmenta kome pripada i reaguje na jedan od dva načina:
  - Gasi proces, ako adresa pripada nedeklarisanom segmentu (*memory access violation*)
  - Učitava stranicu, ako adresa pripada deklarisanom segmentu

# Stranična organizacija

- ◆ Učinak straničenja u više nivoa: proces po pravilu koristi samo mali deo svog virtuelnog adresnog prostora (posebno za 64-bitne arhitekture)



# Stranična organizacija

- ◆ Kako straničenje u više nivoa utiče na performanse?
- ◆ Prepostavke:
  - vreme pristupa fizičkoj memoriji 100 ns
  - vreme pretrage i čitanja TLB-a 20 ns
  - procenat pogotka u TLB-u 98%
- ◆ Vreme pristupa bez TLB-a:
  - straničenje u jednom nivou:  $2 \times 100 \text{ ns} = 200 \text{ ns}$
  - straničenje u tri nivoa:  $4 \times 100 \text{ ns} = 400 \text{ ns}$
- ◆ Vreme pristupa sa TLB-om:
  - straničenje u jednom nivou:  $0,98 \times 120 \text{ ns} + 0,02 \times 220 \text{ ns} = 122 \text{ ns}$  (22%)
  - straničenje u tri nivoa:  $0,98 \times 120 \text{ ns} + 0,02 \times 420 \text{ ns} = 126 \text{ ns}$  (26%)

# Stranična organizacija

## ◆ Zaštita u straničnoj organizaciji:

- OS-a od korisničkih procesa i procesa između sebe inherentno (fizički im se ne može pristupiti)
- od prekoračenja opsega dozvoljenih adresa procesa ili dozvole upisa - proširenjem ulaza PMT bitima zaštite:
  - ◆ *valid/invalid* bit: da li je data stranica uopšte dozvoljena za pristup – da li je u opsegu dozvoljenih adresa ili nije
  - ◆ *read/write/execute* biti: biti dozvole čitanja, upisa ili izvršavanja sadržaja iz date stranice

prilikom preslikavanja adrese, HW (MMU) dovlači deskriptor stranice i najpre proverava pravo pristupa do te stranice; ako pristup nije dozvoljen, generiše se interni prekid (*exception*)

## ◆ Kako uštedeti na PMT ako nisu sve stranice potrebne?

Registrar dužine tabele stranica (*page-table length register*, PTLR) kao deo konteksta procesa

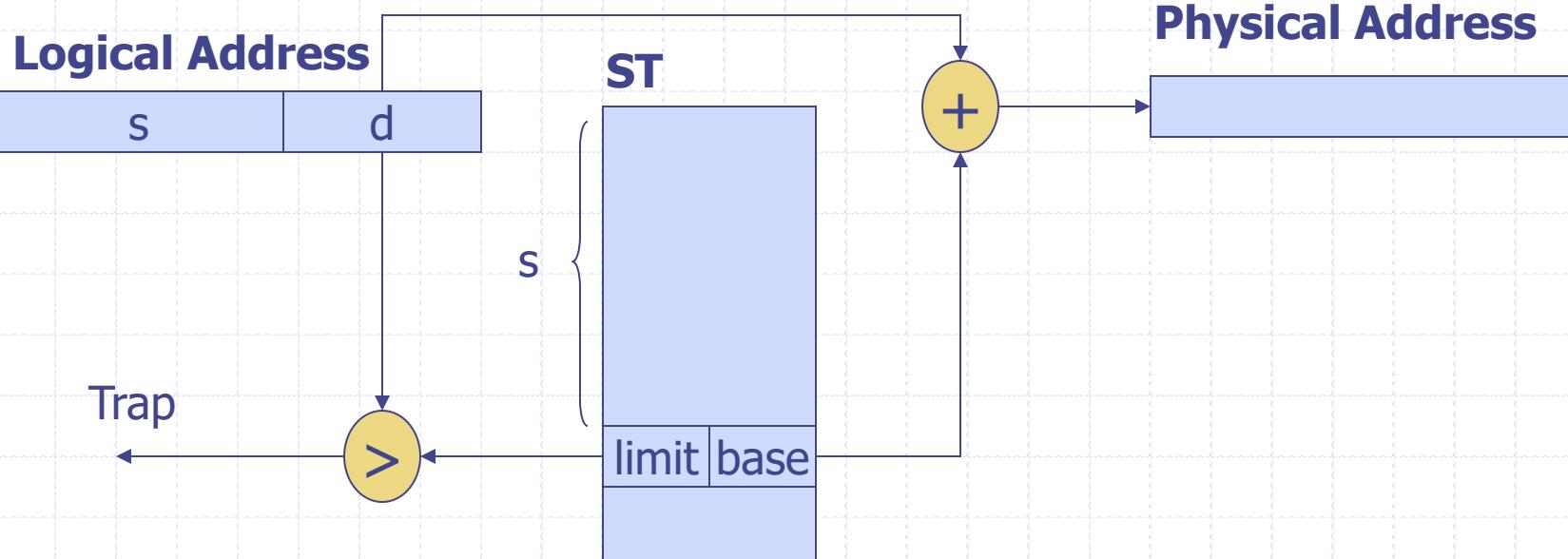
# Stranična organizacija

- ◆ U višekorisničkim sistemima dešava se da mnogo procesa izvršava isti program. Ako svaki proces učitava isti program u svoj fizički memorijski prostor, ovaj prostor se nepotrebno troši dupliranjem istog sadržaja (programskog koda)
- ◆ Rešenje - deljenje stranica (*shared pages*): PMT za različite procese preslikavaju stranice sa istim programskim kodom u isti fizičke okvire, dok se stranice sa podacima preslikavaju različito (sopstveno za svaki proces)
- ◆ Preduslov: da se kod ne menja tokom izvršavanja!

# Stranična organizacija

- ◆ Jedna tehnika za smanjenje režijskih troškova pri kreiranju procesa sa `fork()` - kopiranje na upis (*copy on write*):
  - pri pozivu `fork()`, proces-dete deli sve stranice sa roditeljem – nema kopiranja stranica (PMT se prosto iskopiraju – „plitko kopiranje“)
  - sve stranice logički dozvoljene za upis se u deskriptorima u PMT oba procesa označe kao zabranjene za upis, ali u deskriptorima segmenata kojima pripadaju označene su kao “copy on write” (logički dozvoljene za upis, ali samo ako nisu deljene)
  - sve dok proces pristupa stranici samo za čitanje, ona može da ostane deljena
  - kada proces prvi put upisuje u neku ovakvu deljenu stranicu:
    - ◆ HW generiše izuzetak, jer stranica nije dozvoljena za upis
    - ◆ OS zaključuje da je upis logički dozvoljen, ali da je stranica deljena; stranica se tada kopira i prestaje da bude deljena, već privatna (označava se kao dozvoljena za upis u PMT) i instrukcija ponavlja

# Segmentna organizacija



- ◆ Segmentna organizacija podržava korisnički pogled na program i njegove podatke: adresni prostor je podeljen na logičke celine – *segmente*
- ◆ Podelu na segmente vrši prevodilac prema logičkoj strukturi programa (npr. programski moduli, globalni podaci, stek, dinamička memorija – *heap* itd.)

# Segmentna organizacija

- ◆ Zaštita se prirodno uklapa u segment kao logičku celinu čiji se sadržaj tipično koristi na isti način:
  - deskriptor segmenta (ulaz u ST) sadrži graničnu vrednost (*limit*) koja štiti od prekoračenja veličine segmenta; npr. može se iskoristiti za HW proveru granica niza ako se niz smesti u zaseban segment
  - deskriptor segmenta se može proširiti bitima za prava pristupa; npr. segment sa instrukcijama ima samo pravo izvršavanja, segment sa konstantnim podacima samo pravo čitanja itd.
  - ove provere vrši HW (MMU) pri svakom preslikavanju; ukoliko provera nije prošla, generiše se izuzetak
- ◆ Deljenje segmenata se takođe lako implementira - ST različitih procesa preslikavaju određeni logički segment u istu fizičku početnu lokaciju (*base*); mogu se deliti i samo delovi programa, npr. statičke biblioteke

# Segmentna organizacija

- ◆ Problem: šta je sa adresiranjem kod skokova (instrukcija skoka absolutno adresira odredište skoka u istom segmentu), ako se taj kod deli kao različit logički segment u različitim procesima?
- ◆ Rešenje: koristiti samo relativne skokove u segmentima koji se dele
- ◆ Segmenti su različite veličine, a moraju se alocirati u memoriji. Posledice - kao i kod kontinualne alokacije:
  - potrebna je dinamička alokacija memorije
  - kako izabrati slobodan prostor za smeštanje (*first fit, best fit*)
  - moguća je eksterna fragmentacija; rešenja ista kao i kod kontinualne alokacije:
    - ◆ kompakcija (skupo) ili
    - ◆ straničenje – *segmentno-stranična organizacija*

# Glava 13: Virtuelna memorija

Pojam virtuelne memorije

Učitavanje stranica na zahtev

# Pojam virtuelne memorije

- ◆ Do sada razmatrano: rukovanje memorijom u cilju omogućavanja multiprogramiranja – više procesa može biti u memoriji uz efikasno korišćenje memorije (neophodna relokabilnost)
- ◆ Do sada prepostavljano: ceo proces mora biti u memoriji da bi se izvršavao
- ◆ Motivacija: često se delovi koda koriste jako retko (npr. za obradu greške ili procedure koje se pozivaju po retkoj potrebi) ili se za podatke alocira mnogo veći prostor nego što je potrebno
- ◆ Sledeći korak: proces ne mora biti ceo u memoriji da bi se izvršavao – *virtuelna memorija (virtual memory)*



# Pojam virtuelne memorije

## ◆ Posledice:

- logički adresni prostor može biti mnogo veći od fizičke memorije
- programer je rasterećen brige o ograničenjima fizičke memorije
- više procesa može da se učita u memoriju, povećavajući iskorišćenje
- procesi mogu lako da dele fajlove i segmente memorije (ali samo za čitanje); deljena memorija može da se koristi za međuprocesnu komunikaciju
- usložnjava implementaciju i povećava režijske troškove
- može da bude veoma neefikasno ako se nepažljivo koristi

# Učitavanje stranica na zahtev

◆ Kako učitavati delove programa po potrebi? Učitavanje na zahtev (*demand paging*):

- Stranice procesa nalaze se na sekundarnoj memoriji (brzi disk)
- Inicijalno učitati samo nekoliko stranica procesa koje će biti potrebne; u graničnom slučaju, proces se može pokrenuti bez ijedne učitane stranice (samo se kreira memoriski kontekst – deskriptori segmenata i PMT prvog nivoa sa svim ulazima *null*!)
- Kada se izvršava instrukcija, za pristup nekoj adresi (za instrukciju ili podatak), procesor pronalazi u PMT podatak da stranica nije u memoriji – *page fault*
- Deo OS (*pager*) preuzima kontrolu:
  - ◆ proverava da li je pristup traženoj stranici uopšte dozvoljen procesu; ako nije, proces se može ugasiti
  - ◆ pronalazi slobodan okvir u memoriji; ako takvog nema, bira stranicu istog ili drugog procesa koju će izbaciti iz OM po *algoritmu zamene (page replacement)* - detalji u OS2
  - ◆ pokreće se operacija sa diskom za učitavanje tražene stranice u odabrani okvir; kada se operacija završi, ažurira se PMT procesa

# Učitavanje stranica na zahtev

- ◆ Ključni HW zahtev: omogućiti da se prekinuta instrukcija može izvršiti ispočetka, sa istim rezultatom kao da nije izvršena donekle
- ◆ Ukoliko instrukcija menja samo jednu vrednost (ima samo jedan rezultat), sve je u redu – ponavlja se samo dohvatanje instrukcija i operanada, što nema sporedne efekte
- ◆ Ali šta ako instrukcija vrši više upisa? Mogući pristupi:
  - CPU pokušava da pristupi svim adresama koje će biti korišćene u instrukciji tako da generiše *page fault* unapred, pre nego što instrukcija promeni neku vrednost u memoriji
  - upotrebljavaju se posebni registri za smeštanje starih vrednosti lokacija koje se menjaju; ukoliko nastane *page fault*, CPU vraća stare vrednosti u promenjene lokacije
  - RISC procesori imaju instrukcije sa najviše jednim upisom

# Učitavanje stranica na zahtev

- ◆ Ključni uslovi za postizanje zadovoljavajućih performansi kod dohvatanja stranica:
  - postići što je moguće manju učestanost pojave *page fault*
  - omogućiti što brže učitavanje stranice sa diska:
    - ◆ poseban prostor na disku (*swap space*), izvan fajl sistema
    - ◆ direktni pristup (nema pretrage i indirekcije), adresa na disku neposredno u PMT
- ◆ Procena: da bi se efektivno vreme pristupa memoriji produžilo za ne više od 10%, potrebno je da učestanost pojave *page fault* bude reda  $10^{-6}$

# V Ulazno/izlazni sistemi

Sistemske I/O usluge  
I/O podsistem

# Glava 14: Sistemske I/O usluge

Arhitektura I/O podsistema

Klasifikacija I/O uređaja

Aplikativni programski interfejs prema I/O

# Arhitektura I/O podsistema

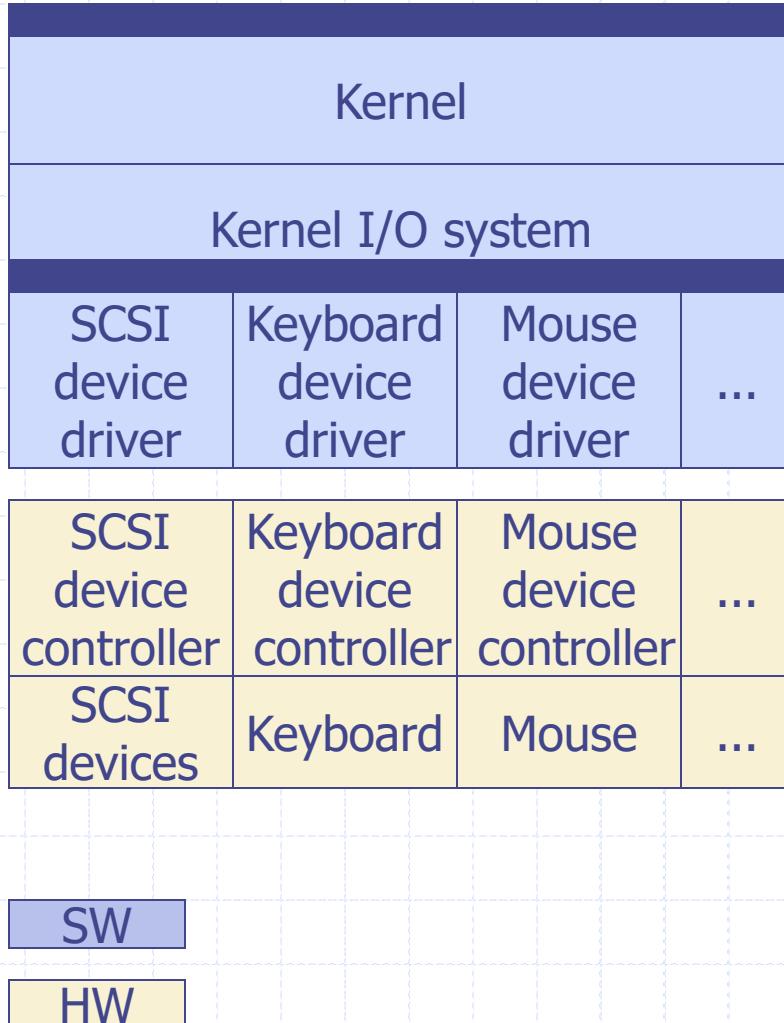
## ◆ Problemi:

- I/O uređaji su izuzetno raznoliki po funkciji, načinu korišćenja, kapacitetu, brzini pristupa i transfera itd. Kako ove razlike učiniti što transparentnijim za korisnike i njihove programe?
- Proizvođači HW stalno izmišljaju i proizvode nove vrste raznolikih uređaja. Kako omogućiti ugradnju novog HW bez stalne promene OS-a?

## ◆ Rešenje - kao i kod svih ostalih složenih i fleksibilnih SW sistema:

- apstrakcija (*abstraction*): zanemariti detalje različitosti zarad isticanja zajedničkih osobina različitih entiteta (klasifikacija, generalizacija)
- enkapsulacija (*encapsulation*): pristupati entitetu samo preko jasno definisanog *interfejsa*, sakriti *implementaciju*
- slojevitost (*layering*): softverske komponente realizovati slojevito, tako da jedan sloj (komponenta) može da interaguje samo sa susednim slojevima; slojevi se redaju po nivoima apstrakcije

# Arhitektura I/O podsistema



*Drajver uređaja (device driver):*

- ◆ sakriva specifičnosti uređaja iza standardnog interfejsa prema Kernel I/O sistemu
- ◆ omogućava proizvođačima HW uređaja da svoje uređaje prilagode datom OSu bez potrebe za promenom OS-a

Nezgoda: svaki OS ima svoj standard za interfejs prema drajverima

# Klasifikacija I/O uređaja

Apstrakcija I/O uređaja – klasifikacija na kategorije sličnih prema određenim karakteristikama:

◆ Prema jedinici prenosa:

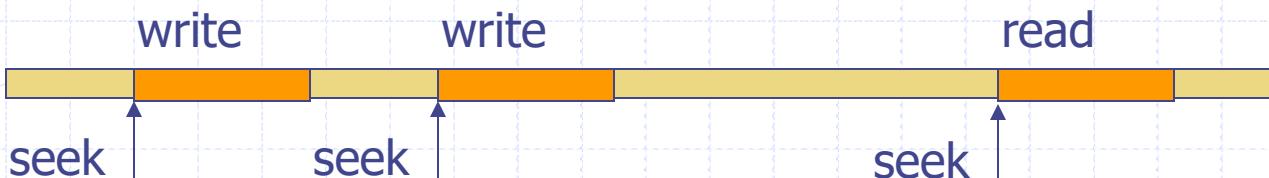
- znakovni (*character-stream*): prenose bajt po bajt kao jedinicu
- blokovski (*block*): prenose blok podataka kao jedinicu

◆ Prema načinu pristupa:

- sekvencijalni (*sequential*): prenos samo po redu (tastatura, štampač, modem, mreža, ...)



- sa direktnim (proizvoljnim) pristupom (*direct, random-access*): može se pristupiti proizvoljnom delu ukupnog prostora (diskovi, *flash*, ...)



# Klasifikacija I/O uređaja

## ◆ Prema vremenskim karakteristikama transfera:

- sinhroni (*synchronous*): obavlja operaciju u predvidivim trenucima/intervalima ili ima predvidivo vreme odziva (disk, monitor)
- asinhroni (*asynchronous*): obavlja operaciju u nepredvidivim trenucima/intervalima ili ima nepredvidivo vreme odziva (tastatura)

## ◆ Prema deljivosti:

- deljivi (*shareable*): više procesa mogu uporedo koristiti uređaj i preplitati zahteve za operacijom (disk, tastatura)
- nedeljivi, "posvećeni" (*dedicated*): samo jedan proces može vršiti operaciju sa uređajem i mora je završiti u potpunosti pre nego što počne drugi (štampač)

## ◆ Prema brzini izvršenja operacije: od nekoliko bajtova do nekoliko gigabajta u sekundi

## ◆ Prema vrsti moguće operacije: samo čitanje, samo upis, i čitanje i upis

# I/O API

## ◆ Najčešće podržane kategorije uređaja:

- blokovski (*block I/O*)
- znakovni tokovi (*character-stream I/O*)
- memorijski preslikani fajl (*memory-mapped file*)
- mrežne priključnice (*network sockets*)
- pristup časovnicima (*clock*) i vremenskim brojačima (*timer*)

## ◆ Neki OS omogućuju i direktni pristup drijveru generičkom operacijom koja se direktno prosleđuje drijveru; npr. UNIX `ioctl()`

## ◆ Blokovski uređaji – osnovne operacije interfejsa:

- *read* – čita blok podataka sa tekuće pozicije na uređaju i smešta na zadato mesto u memoriji
- *write* – upisuje blok podataka sa zadatog mesta u memoriji na tekuću poziciju na uređaju
- *seek* – ako je uređaj sa direktnim pristupom – pozicionira tekuću poziciju na uređaju na zadato mesto

# I/O API

- ◆ Znakovni tokovi – osnovne operacije interfejsa:
  - *get character* – čita i vraća jedan znak sa uređaja
  - *put character* – upisuje dati znak na uređaj
- ◆ Memorijski preslikani fajl – pristup fajlu na isti način kao i segmentu operativne memorije; OS vrši transfer iz fajla i u fajl po potrebi – detalji u OS2
- ◆ Mrežna priključnica (*socket*) - apstrakcija koja dozvoljava komunikaciju između dva procesa na udaljenim računarima; osnovne operacije (detalji u OS2):
  - kreiraj i vrati jedan *socket*
  - poveži *socket* sa drugim na datoj IP adresi
  - pošalji paket (poruku) na dati *socket*
  - primi (sa ili bez čekanja) poruku sa datog *socket-a*

# I/O API

## ◆ Usluge vezane za realno vreme:

- očitaj i vrati tekući datum i vreme
- vrati vreme proteklo između dva događaja
- pokreni datu operaciju u zadato vreme – uspavaj proces do zadatog vremena/na zadato vreme

## ◆ Hardverska podrška:

- sat realnog vremena
- periodični prekid od vremenskog brojača
- programabilni vremenski brojač: pokreće se zadavanjem vremena za koje će generisati prekid

## ◆ Korisnički programi i OS zahtevaju merenje (proizvoljno) mnogo vremenskih intervala. Kako to ostvariti sa konačnim (malim) brojem hardverskih vremenskih brojača?

## ◆ Rešenje: koncept logičkog (softverskog) vremenskog brojača. Realizacija: "Programiranje u realnom vremenu - Skripta", poglavlje "Realno vreme" i zadatak 9.1

# I/O API

## ◆ Vrste poziva I/O usluga:

- blokirajući: proces koji je zadao operaciju se blokira sve dok operacija nije završena u potpunosti
- neblokirajući sinhroni: proces zadaje operaciju i ona mu odmah vraća kontrolu sa vraćenom informacijom o tome koliki deo operacije je uspešno završen (ništa, delimično ili sve)
- neblokirajući asinhroni: proces samo zadaje operaciju i nastavlja izvršavanje; ako je potrebno, završetak operacije se procesu dojavljuje asinhronim signalom (npr. postavljanje nekog indikatora ili *call-back* rutinom)

# Glava 15: I/O podsistem

---

Rasporedjivanje

Baferisanje

Keširanje

*Spooling*

Zaštita

Obrada I/O zahteva

Performanse

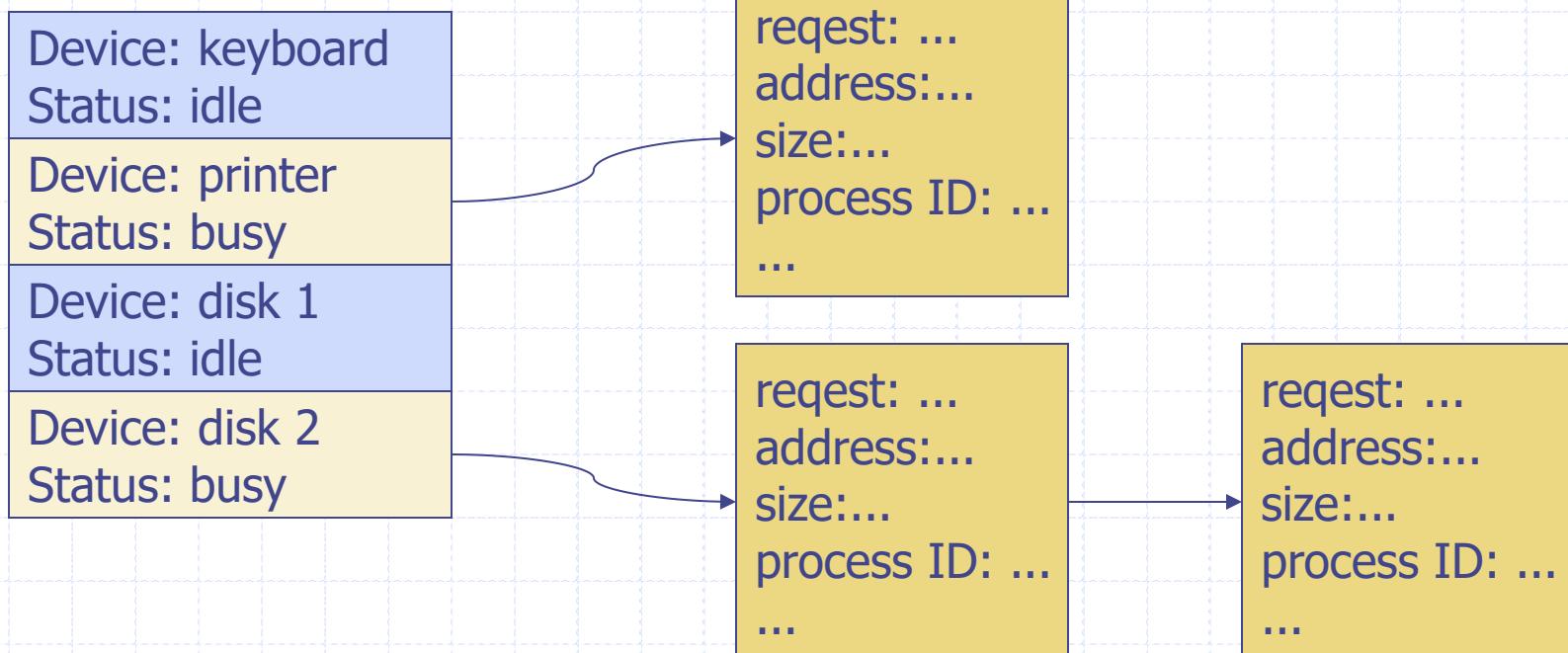
---

# Raspoređivanje

- ◆ Zahtevi za I/O operacijama na istom uređaju smeštaju se u red čekanja
- ◆ I/O podsistem kernela uzima jedan po jedan zahtev i zadaje operaciju drajveru uređaja, ali kojim redom?
- ◆ Primer: u redu operacija na disku nalaze se tri zahteva, prvi koji pristupa bloku blizu unutrašnjeg cilindra, drugi koji pristupa spoljašnjem cilindru, treći koji pristupa cilindru blizu sredine; glava diska je trenutno na spoljašnjem cilindru; šta ako se zahtevi opslužuju redom kojim su zadati?
- ◆ U opštem slučaju, raspoređivanje zahteva za operacijama na uređaju (*I/O scheduling*) treba da pronađe povoljan redosled izvršavanja da bi se smanjilo vreme čekanja procesa na kraj operacije

# Raspoređivanje

- ◆ Kada se izda blokirajući zahtev za I/O operacijom, zahtev se smešta u red zahteva za dati uređaj, a pozivajući proces se blokira
- ◆ Raspoređivač I/O zahteva (*I/O scheduler*) bira jedan po jedan po odgovarajućem algoritmu – detalji u OS2

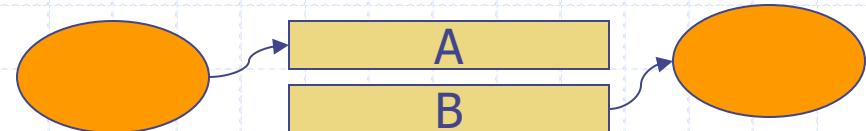


# Baferisanje

- ◆ Bafer je memorijski prostor za smeštanje podataka koji se prenose između dva učesnika u prenosu (proces i uređaj)
- ◆ Razlozi uvođenja bafera u I/O operacije:
  - amortizacija razlike u brzinama "proizvodnje" i "potrošnje" podataka učesnika u prenosu
  - adaptacija različitih veličina blokova prenosa učesnika
  - semantika kopiranja (*copy semantics*): proizvođač kopira originalne podatke u bafer, odakle se dalje prenose, pa nakon toga može da menja originalne podatke bez uticaja na prenos

## Dvostruki bafer:

- proizvođač upisuje u bafer A, dok potrošač čita iz bafera B
- kada su oba završila, baferi zamenjuju uloge
- primer: prenos podataka sa modema na disk



# Keširanje

- ◆ Keš (*cache*) je deo brže memorije koji sadrži kopiju dela podataka sa sporije memorije – pristup podatku u kešu je mnogo brži nego pristup originalnom podatku
- ◆ Primeri keša:
  - procesorski keš
  - keš podataka sa diska – u operativnoj memoriji, pod kontrolom OS-a
  - keš disk kontrolera – na samom kontroleru, pod njegovom kontrolom, transparentan za OS
- ◆ Razlika između bafera i keša: bafer može sadržati jedinstvenu kopiju podataka; keš po definiciji sadrži kopiju originala
- ◆ Keš i bafer se mogu koristiti zajedno – isti deo memorije za obe namene

# Spooling

- ◆ Problem: više procesa zadaje operacije nedeljivom uređaju uporedo, tako da se operacije prepliću? Npr., više procesa šalje blokove podataka na štampač
- ◆ Jedan pristup - OS obezbeđuje sistemske usluge za rezervaciju uređaja:
  - kada mu je potreban, proces rezerviše uređaj i čeka dok ovaj nije raspoloživ
  - uređaj se oslobađa tek kada proces završi ceo posao sa uređajem, opet eksplicitnom operacijom
- ◆ Drugi pristup – *spooling*:
  - operacije zadate uređaju od strane procesa smeštaju se u poseban fajl; kada završi, proces zatvara fajl i "predaje ga" OSu
  - poseban proces ili nit OSa, *spooler*, uzima jedan po jedan fajl iz reda i prenosi njegov sadržaj na uređaj

# Zaštita

- ◆ Korisnički proces može greškom ili zlonamerno da ugrozi rad sistema zahtevajući neregularne I/O operacije. Kako sprečiti?
- ◆ Tehnike:
  - I/O operacije se mogu zadati isključivo sistemskim pozivom; I/O instrukcije se uređaju zadaju iključivo u zaštićenom procesorskom režimu
  - mehanizam zaštite memorije štiti memorijski preslikani I/O prostor
  - mehanizam zaštite memorije ograničava pristup samo do segmenata memorije dozvoljenih za I/O; primer: igrice i grafički programi zahtevaju direktni pristup memorijski preslikanom prostoru grafičkih kartica zbog bržeg pristupa
  - mehanizam virtuelne memorije štiti bafere i keševe OS-a

# Obrada I/O zahteva

1. Proces izvršava blokirajući sistemski I/O poziv
2. Kod sistemske usluge proverava da li su zahtev i njegovi parametri ispravni; ako nisu, vraća se greška
3. Ako se radi o ulazu, proverava se da li su traženi podaci već u kešu; ako jesu, podaci se prenose pozivaocu i sistemski poziv se završava
4. Određuje se uređaj i drajver na koga se zahtev odnosi
5. Proces se uklanja iz reda spremnih i smešta u red onih koji čekaju na dati uređaj
6. Zahtev za I/O operacijom se smešta u red za raspoređivanje na datom uređaju
7. U nekom trenutku, dati zahtev dolazi na red i zadaje se drajveru uređaja (npr. interna nit jezgra obrađuje zahteve i poziva operacije drajvera)

# Obrada I/O zahteva

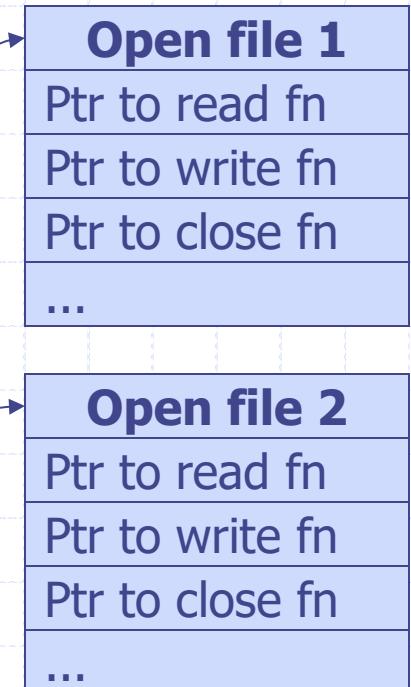
8. Drajver alocira bafer za transfer podataka i zadaje operaciju kontroleru uređaja upisom u njegov kontrolni registar; drajver može da koristi programirani I/O ili DMA
9. Kontroler uređaja obavlja operaciju i prenosi podatke
10. Da bi bio obavešten o završetku operacije, drajver uređaja može da koristi prozivanje ili prekid
11. Po završetku operacije, drajver uređaja označava zahtev izvršenim i obaveštava I/O podsistem o tome
12. Kernel prenosi podatke i status operacije u prostor korisničkog procesa
13. Kernel deblokira proces vraćajući ga u red spremnih
14. Proces nastavlja sa izvršavanjem i završava sistemski poziv

# Obrada I/O zahteva

- ◆ Kako pronaći drajver uređaja za dati zahtev?
- ◆ Veoma raznoliki pristupi u raznim OS,  
uglavnom se svode na preslikavanje  
simboličkog imena uređaja ili fajla u drajver;  
odgovarajuće strukture OS-a moraju da vode  
računa o tom preslikavanju
- ◆ Kako odrediti i pronaći implementaciju date  
operacije za dati drajver/uređaj?
- ◆ Objektno orijentisani pristup - polimorfizam i  
dinamičko vezivanje (*dynamic binding*)

Process' open files

file 1
file 2
...



# Performanse

## ◆ I/O operacije bitno utiču na performanse celog sistema:

- povećavaju broj promena konteksta
- uključuju mnogo kopiranja podataka između raznih bafera
- povećavaju opterećenje magistrale zbog prenosa sa DMA
- povećavaju učestanost prekida

## ◆ Načini poboljšanja performansi:

- smanjiti broj promena konteksta i njegove troškove (npr. korišćenjem kernel niti)
- smanjiti broj kopiranja podataka
- smanjiti učestanost prekida: povećanjem blokova ili prozivanjem – ako je operacija brza, prozivanje je efikasnije nego obrada prekida
- određene operacije implementirati u hardveru – DMA ili specijalizovani I/O procesori
- balansirati opterećenje CPU, memorije, magistrale i uređaja

# VI Fajl sistemi

Interfejs fajl sistema

Implementacija fajl sistema

# Glava 16: Interfejs fajl sistema

---

Pojam fajla

Struktura direktorijuma

Montiranje fajl sistema

Deljenje fajlova

Zaštita

# Pojam fajla

- ◆ Fajl (*file*) je logička jedinica smeštanja informacija
- ◆ Apstrahuje fizička svojstva uređaja za smeštanje podataka.  
OS preslikava fajl na fizički uređaj
- ◆ Uniformiše pogled na skup informacija smešten na uređaju
- ◆ Za korisnika, fajl je najmanja jedinica alokacije logičkog prostora na sekundarnoj memoriji: podaci se ne mogu smeštati izvan fajla
- ◆ Fajlovi: programi ili podaci
- ◆ OS ne ulazi u tumačenje sadržaja i strukture fajla, osim za neke posebne vrste fajlova (npr. fajl sa programom)
- ◆ Strukturu fajla tumači onaj program koji ga je kreirao ili koji je u stanju da je prepozna
- ◆ Neki OS zahtevaju određenu strukturu svakog fajla koju u celini ili delimično tumače. Minimum – izvršivi fajl

# Pojam fajla

- ◆ Fajl je *apstraktni tip podataka (abstract data type)*: atributi+operacije
- ◆ Atributi fajla (tipično):
  - simboličko ime (naziv) fajla: identifikator u ljudski čitljivoj formi
  - identifikator: jedinstveni interni identifikator fajla u sistemu
  - tip
  - lokacija: informacija o uređaju i mestu gde se fajl nalazi na njemu
  - veličina: trenutna veličina (u bajtovima, rečima ili blokovima)
  - zaštita: informacije o pravima pristupa fajlu (ko sme da ga čita, piše, briše, izvršava itd.)
  - datum, vreme i korisnik koji je kreirao, poslednji modifikovao ili poslednji pristupao fajlu
- ◆ Informacije o fajlovima čuvaju se u strukturi direktorijuma (katalog, imenik)

# Pojam fajla

- ◆ Osnovne operacije sa fajlom – usluge OS (sistemske pozive):
  - Kreiranje (*create*) fajla: alociranje prostora za smeštanje fajla i formiranje ulaza u direktorijumu
  - Upis (*write*) u fajl; parametri: ime fajla i podaci za upis; traži se fajl sa datim imenom; na poziciju *pokazivača za upis* datog fajla upisuju se dati podaci i pokazivač pomera na novo mesto
  - Čitanje (*read*) iz fajla; parametri: ime fajla i mesto u OM za smeštanje pročitanih podataka; sa pozicije *pokazivača za čitanje* datog fajla čitaju se podaci i pokazivač pomera na novo mesto; obično su pokazivači za čitanje i upis isti, tj. postoji samo jedan *pokazivač trenutne pozicije za fajl (current-file-position pointer)*
  - Pozicioniranje (*reposition*) pokazivača na novo mesto (*seek*)
  - Brisanje fajla (*delete*): prostor koji je zauzimao fajl se proglašava slobodnim i briše se ulaz u direktorijumu

# Pojam fajla

- ◆ Ostale operacije sa fajlom – usluge OS (sistemske pozive):
  - Odsecanje (*truncate*): brisanje sadržaja fajla, ali zadržavanje atributa (osim veličine koja se postavlja na 0)
  - Preimenovanje (*rename*) fajla
  - Dodavanje informacija na kraj fajla (*append*): upis podataka na kraj fajla
  - Kopiranje (*copy*) fajla
  - Promena atributa fajla
- ◆ Većina ovih operacija zahteva pretragu direktorijuma za ulazom za fajl sa datim imenom i dobijanje lokacije i ostalih atributa fajla – neefikasno!

# Pojam fajla

- ◆ Rešenje: kada program želi da pristupa fajlu, prvo mora da pozove sistemsku uslugu *otvaranja fajla* (*file open*):
  - OS vodi *tabelu otvorenih fajlova*; za svaki otvoreni fajl, ulaz u ovoj tabeli čuva sve potrebne informacije o fajlu (atribute, posebno lokaciju i pokazivač trenutne pozicije)
  - sistemska usluga *otvaranja fajla* radi:
    - ◆ pronalazi ulaz za dati fajl u strukturi direktorijuma
    - ◆ otvara novi ulaz u tabeli otvorenih fajlova
    - ◆ učitava atribut fajla u ulaz u tabeli otvorenih fajlova
    - ◆ opcionalno proverava prava pristupa do fajla u odnosu na pristup zahtevan pri otvaranju fajla
    - ◆ vraća indeks ili pokazivač na ulaz u tabeli otvorenih fajlova
  - svako sledeće obraćanje fajlu iz programa, tj. sve ostale operacije nad otvorenim fajlom koriste ID ulaza u tabeli otvorenih fajlova, a ne ime fajla – nema pretrage direktorijuma, efikasan pristup do atributa fajla
  - nakon upotrebe, fajl se mora zatvoriti, da bi se ulaz uklonio iz tabele otvorenih fajlova

# Pojam fajla

- ◆ Podrška mogućnosti otvaranja istog fajla od strane više procesa istovremeno – dva nivoa tabele otvorenih fajlova, za svaki proces pojedinačno i globalno za sve proceze na nivou OS

Process X Open Files Table

Cur File Ptr	Access Rights	Ptr

Global Open Files Table

1	Location	Name	...

Process Y Open Files Table


Y: FILE fh = file\_open("...",read\_only)  
Y: file\_close(fh)

# Pojam fajla

## ◆ Koncept zaključavanja fajla (*file locking*):

- jedan proces traži da zaključa fajl; ako nijedan drugi proces nije zaključao fajl, proces će dobiti ključ i nijedan drugi proces neće moći da dobije ključ nad fajлом
- modaliteti zaključavanja:
  - ◆ dve vrste ključa: deljeni (*shared*) i ekskluzivni (*exclusive*); više procesa može imati deljeni ključ; samo jedan može imati ekskluzivni ključ
  - ◆ obavezni ključ (fajl se implicitno zaključava pri otvaranju) ili neobavezni ključ (fajl se zaključava na eksplisitan zahtev)

# Pojam fajla

- ◆ Pojam tipa fajla: OS može, ali ne mora da poznaje koncept tipa fajla; ako ga poznaje, može da kontroliše upotrebu fajla
- ◆ Način smeštanja informacije o tipu:
  - u ekstenziji imena fajla (Windows)
  - kao poseban atribut fajla (postavlja se prilikom kreiranja; Mac OS)
  - u samom sadržaju fajla (UNIX: opcioni *magic number* na početku fajla)
- ◆ Interno strukturiranje fajla:
  - smeštanje informacija na disku je uvek u jedinicama fiksne veličine – blokovima (veličine zavisne od veličine sektora) => interna fragmentacija
  - logički, fajl se može posmatrati kao:
    - ◆ nestrukturiran: sekvenca bajtova sa mogućnošću direktnog pristupa svakom bajtu
    - ◆ strukturiran u logičke zapise fiksne ili promenljive veličine

Neophodno preslikavanje u blokove – jednostavan problem za OS

# Struktura direktorijuma

- ◆ Disk se može podeliti na *particije (partition, slices, minidisks)*
- ◆ Svaka particija može da sadrži poseban fajl sistem ili da bude rezervisana za zamenu (*swap space*)
- ◆ Particija može da se proteže i na više fizičkih diskova
- ◆ Više particija može da se kombinuje u veću celinu – *volumen (volume)*; fajl sistem se može kreirati na jednom volumenu
- ◆ Svaki volumen ili particija mora da sadrži informacije o fajlovima na sebi – *direktorijum (directory)*
- ◆ Direktorijum je imenik fajlova – preslikava ime fajla u atributе fajla (lokacija i drugi), poput tabele simbola

# Struktura direktorijuma

## ◆ Tipične operacije nad direktorijumom:

- pronađi fajl sa datim imenom ili sa imenima koja zadovoljavaju dati kriterijum (*search*)
- kreiraj fajl i dodaj u direktorijum
- obriši fajl i izbac i iz direktorijuma
- vrati spisak svih fajlova
- promeni ime fajla

## ◆ Najjednostavnija struktura direktorijuma – u jednom nivou:

- svi fajlovi su u istom (jedinom) direktorijumu
- fajl mora imati jedinstveno ime u direktorijumu, pa i u celom sistemu
- mnogo fajlova smanjuje preglednost
- fajlovi se ne mogu organizovati u logičke celine
- krajnje nepraktično, posebno za više korisnika

# Struktura direktorijuma

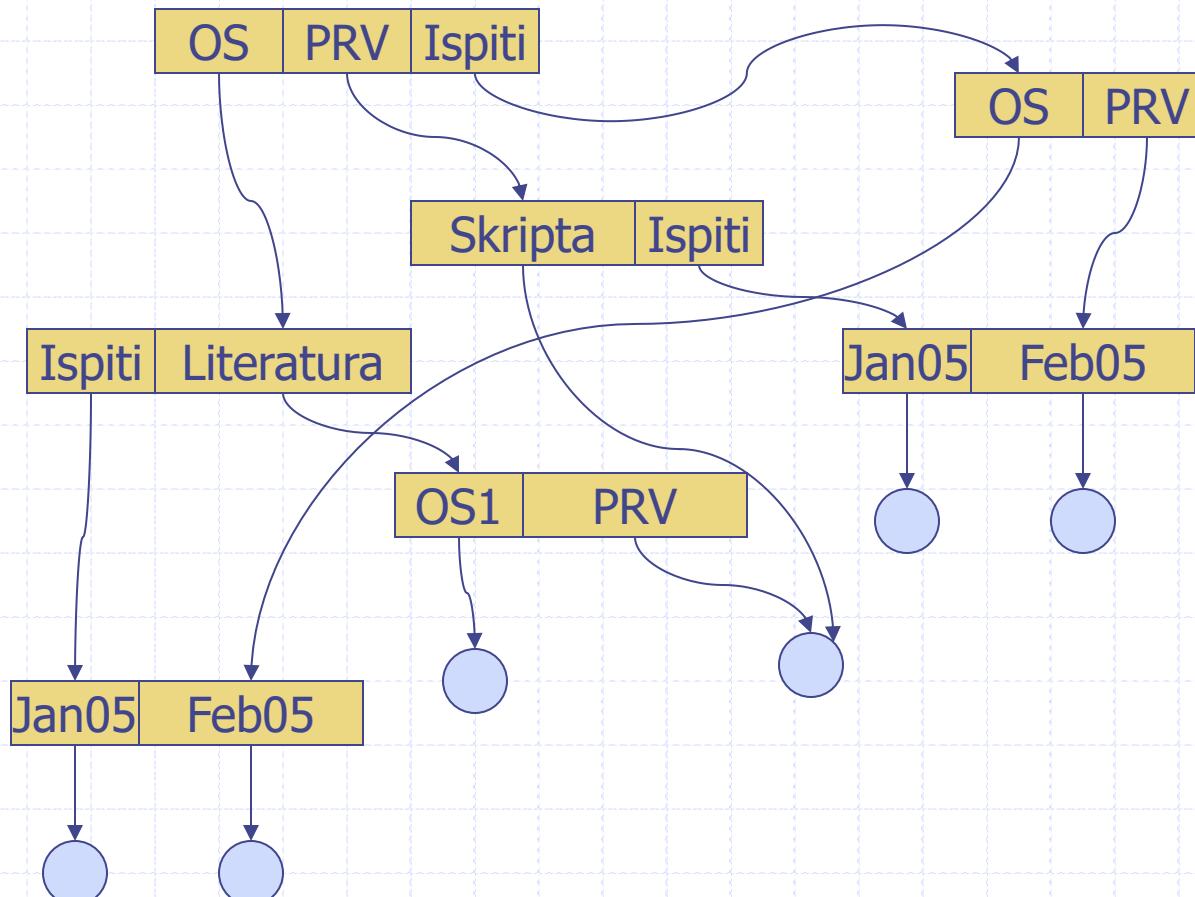
- ◆ Direktorijum u dva nivoa: za svakog korisnika zaseban direktorijum u jednom nivou; i dalje veoma problematično:
  - fajlovi jednog korisnika su nestrukturirani i nepregledni
  - kako deliti fajlove? kako pristupati fajlovima drugog korisnika? – imenovanje punom stazom (npr. volume/user/file)
  - kako pristupati sistemskim fajlovima? – složenijom pretragom: ako se fajl ne nađe u direktorijumu tekućeg korisnika, traži se u sistemskom ili u spisku staza za pretragu (*search path*)
- ◆ Uopštenje: dozvoliti hijerarhijsku strukturu poddirektorijuma oblika stabla proizvoljne dubine:
  - korisnik može manipulisati strukturom proizvoljno
  - svaki fajl je jedinstveno određen punom stazom od korena
  - direktorijum je takođe fajl, samo što se posebno tretira – ima strogo određenu internu strukturu koju tumači fajl sistem OS-a

# Struktura direktorijuma

- ◆ Staze do fajla:
  - svaki proces ima svoj *tekući direktorijum (current directory)* koji se može promeniti sistemskim pozivom
  - otvaranje fajla samo po imenu traži fajl u tekućem direktorijumu
  - fajl se može zadati i zadavanjem:
    - ◆ pune staze od korena stabla direktorijuma
    - ◆ relativne staze u odnosu na tekući direktorijum
- ◆ Struktura stabla ne dozvoljava deljenje direktorijuma ili fajlova između korisnika. Opštiji pristup – struktura direktorijuma i fajlova je usmereni graf bez petlji (*directed acyclic graph, DAG*)
- ◆ Deljeni direktorijum ili fajl se može nalaziti u više direktorijuma, ali nije kopiran, već oni njega samo referišu

# Struktura direktorijuma

## DAG struktura direktorijuma:



# Struktura direktorijuma

## ◆ Implementacija:

- svaki ulaz u svakom direktorijumu je podjednak i predstavlja referencu na fizički fajl ili direktorijum (kao na prethodnoj slici)
- neki ulazi predstavljaju fizičke fajlove ili direktorijume, a neki veze ili prečice (*link, shortcut*) koje referišu na fizički fajl ili direktorijum (punim simboličkim imenom ili fizičkom referencom)

## ◆ Pitanje: kako se odnositi prema brisanju fajla? Mogući pristupi:

- obrisati fajl kad god neko to zatraži preko bilo koje veze; šta je sa ostalim vezama?
- obrisati fajl tek kada nestane poslednja referenca

## ◆ Konačno uopštenje: dozvoliti petlje u grafu – dozvoliti da poddirektorijum referiše na sopstveni naddirektorijum.

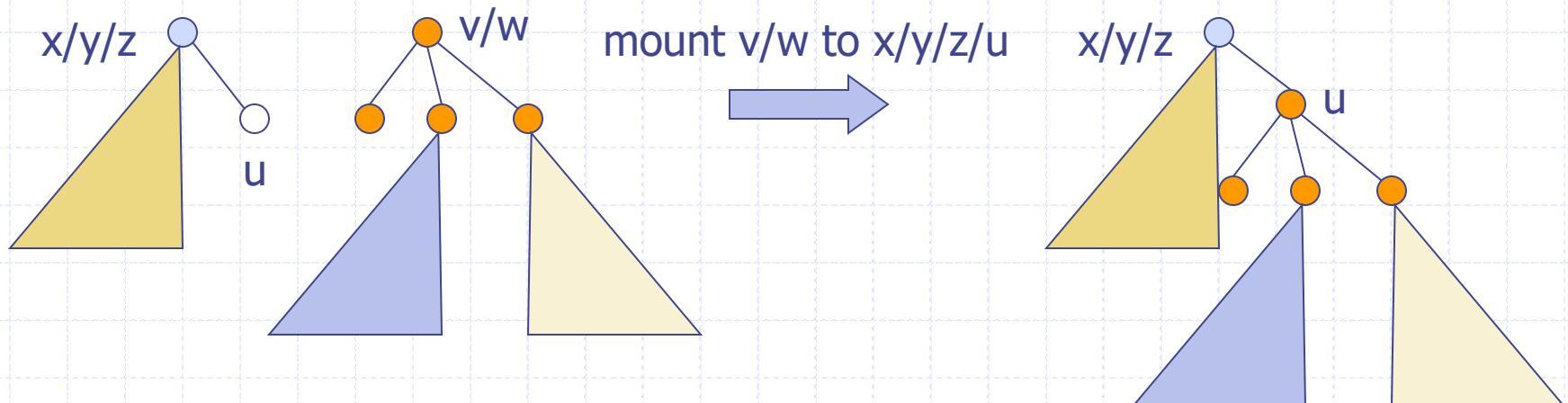
### Pitanja:

- kako implementirati obilazak fajl sistema npr. zbog pretrage?
- kako zaključiti da fajl više niko ne referiše pa se može obrisati?

# Montiranje fajl sistema

## ◆ Montiranje fajl sistema sa volumena (*file-system mounting*):

- OSu je zadato ime volumena na kome je neki fajl sistem i mesto (tačka montiranja, *mounting point*), tipično prazan direktorijum u fajl strukturi gde će “montirati” fajl sistem sa tog uređaja
- OS proverava da li je dati fajl sistem koji se montira prihvativ i ima očekivani format
- OS menja strukturu direktorijuma svog fajl sistema tako da montirani fajl sistem postaje podstruktura (poddirektorijum) od tačke montiranja



# Montiranje fajl sistema

- ◆ Fajl sistem sa nekog volumena mora se najpre montirati da bi bio dostupan u jedinstvenom prostoru imena fajl sistema OS-a:
  - OS montira fajl sisteme sa uređaja koje pronađe, pri pokretanju (fiksni uređaji) ili kada se uređaj priključi ili medijum ubaci (flopi, flash, CD)
  - montiranje može biti i na eksplicitan zahtev (UNIX), i to prilikom startovanja sistema (čitanjem spiska iz sistemskog konfiguracionog fajla) ili ručno (sistemskim pozivom)
- ◆ Montiranje datog fajl sistema može biti ograničeno na određeno ciljno mesto (samo u koren, samo u prazan direktorijum i sl.), na samo jedno mesto itd.
- ◆ Demontiranje (*unmounting*): suprotna operacija, fajl sistem sa montiranog volumena više nije dostupan

# Deljenje fajlova

- ◆ Udaljeni (*remote*) fajl sistemi – pristup fajlovima preko mreže:
  - File Transfer Protocol (FTP): protokol koji obezbeđuje operacije za prenos fajlova između udaljenih fajl sistema; anonimni (*anonymous*) ili autorizovani pristup fajlovima
  - distribuirani fajl sistemi (DFS): udaljeni direktorijumi se vide kao i lokalni
  - World Wide Web
- ◆ Klijent/server (*client/server*) arhitektura
- ◆ Distribuirani direktorijumi, protokoli imenovanja i autorizacija

# Zaštita

- ◆ U višekorisničkim sistemima sa deljenim fajlovima, potrebno je ograničiti i kontrolisati pristup korisnicima do određenih fajlova
- ◆ Tipovi pristupa fajlovima:
  - čitanje (*read*)
  - upis (*write*)
  - izvršavanje (*execute*): učitavanje fajla u memoriju i izvršavanje kao programa
  - dodavanje (*append*): upis novih informacija na kraj fajla
  - brisanje (*delete*)
  - listanje (*list*): pročitati ime i atribute fajla (direktorijuma)
  - ostale operacije mogu biti izvedene iz osnovnih (atomičnih)
- ◆ Postoji mnogo mehanizama zaštite, svi imaju svoje prednosti i nedostatke, kao i situacije u kojima su pogodni ili ne

# Zaštita

- ◆ Zadatak kontrole pristupa (*access control*): održavati preslikavanje (korisnik, fajl, operacija) -> dozvoljeno/zabranjeno i pomoću tog preslikavanja proveravati pristup do fajla
- ◆ Definisanje ovog preslikavanja za svakog korisnika, fajl i operaciju pomoću lista kontrole pristupa (*access control list, ACL*) je nepraktično:
  - suviše informacija, teško ih je definisati i održavati
  - ako se svakom fajlu pridruži spisak korisnika i prava pristupa operacijama, ulaz u direktorijumu treba da bude promenljive i neograničene veličine, što komplikuje rukovanje prostorom

# Zaštita

- ◆ Zato se upotrebljavaju "kondenzovane" varijante, klasifikacijom korisnika u grupe:
  - vlasnik (*owner*): korisnik koji je kreirao fajl; podrazumevano ima pravo da vrši sve operacije sa fajlom i da daje prava drugima
  - grupa (*group*): skup korisnika koji su imenovani kao radna grupa jer dele fajl izvršavajući slične operacije; imaju pravo na podskup operacija nad fajlom
  - svi ostali korisnici, univerzum (*universe*): imaju pravo na podskup operacija sa fajlom
- ◆ UNIX: po tri bita (**rwx** – *read, write, execute*) za vlasnika, grupu i ostale određuju prava izvršavanja ove tri operacije
- ◆ Moderniji, fleksibilniji pristup: opcionalno dodavanje lista kontrole pristupa ovako fiksnom zapisu prava za vlasnika, grupu i ostale – finije podešavanje za pojedine korisnike
- ◆ Primer: MS Windows i njegov GUI za definisanje prava

# Glava 17: Implementacija fajl sistema

Struktura fajl sistema

Implementacija direktorijuma

Metode alokacije

Rukovanje slobodnim prostorom

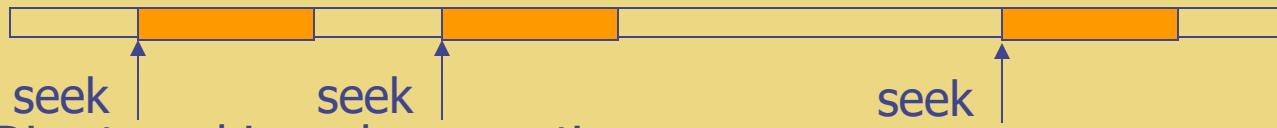
Efikasnost i performanse

Oporavak od otkaza

# Struktura fajl sistema

## File system API:

- File: fopen(...), fclose(...), fseek(...), fread(...), fwrite(...), fappend(...),...  
        write              write              read
- Directory: hierarchy, operations...
- Protection...



## File system implementation:

- Data structures (persistent and in-memory)
- Procedures...

## Block I/O (disk) API:

read(BlockNo, void\* buffer), write(BlockNo, void\* buffer)

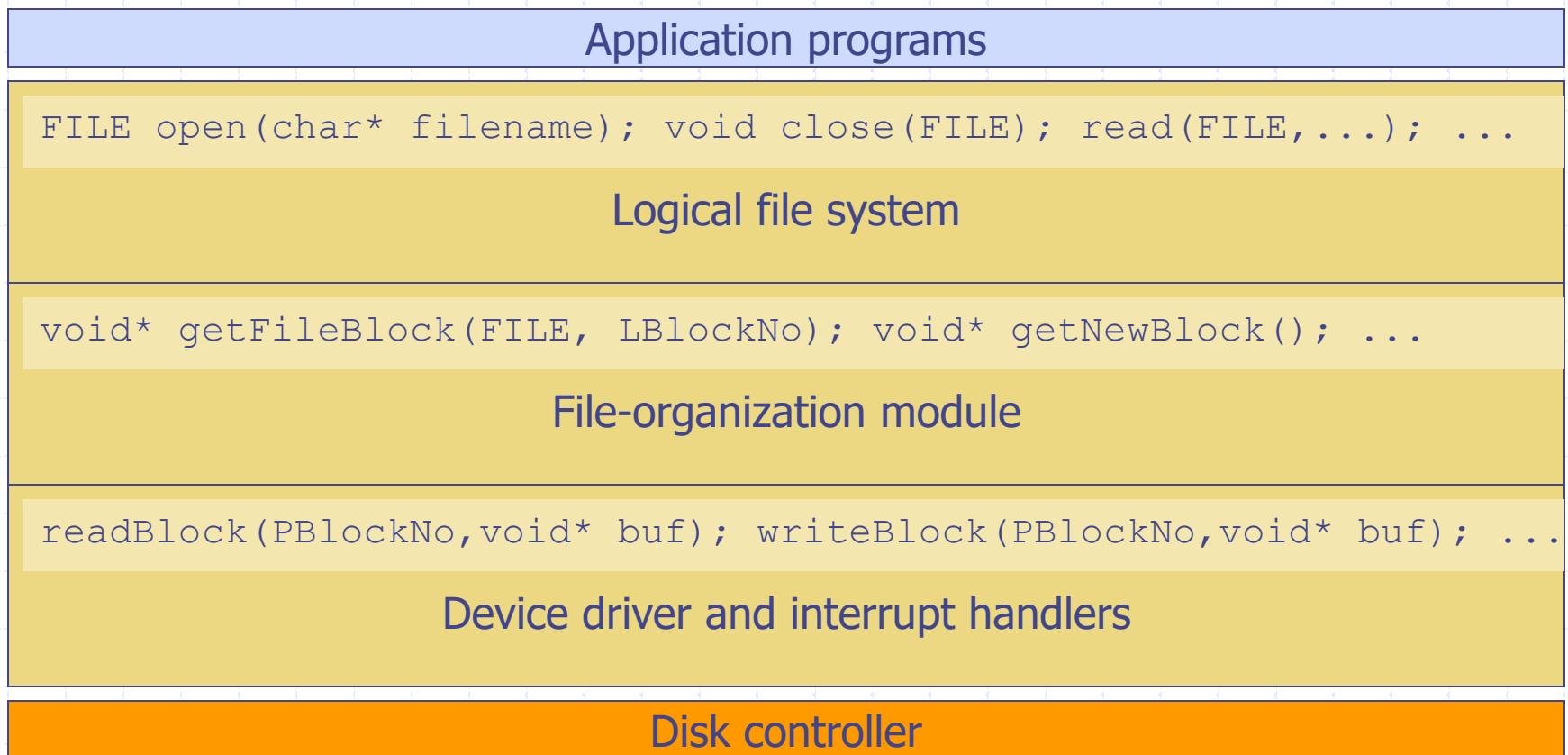


# Struktura fajl sistema

- ◆ Fajl sistemi obično počivaju na sekundarnim memorijskim medijumima koji permanentno čuvaju veliku količinu podataka
- ◆ Ova diskusija se prvenstveno odnosi na smeštanje fajl sistema na diskove kao najčešće medijume
- ◆ Primarne karakteristike diskova koje ih čine pogodnim:
  - blok-orientisani uređaj sa mogućnošću čitanja i upisa: pročitaj blok u memoriju, izmeni ga, upiši ga na disk; blok tipično sadrži nekoliko sektora, a veličine sektora su od 32B do 4KB, tipično 512B
  - uređaj sa direktnim pristupom bilo kom bloku, samo uz potrebu pomeranja glave diska i čekanja na rotaciju
- ◆ Implementacije fajl sistema jako variraju i sve imaju svoje specifičnosti. Ovde se prikazuju samo neki osnovni principi
- ◆ Postoji mnogo fajl sistema u upotrebi, a jedan isti OS može podržavati i više fajl sistema. Npr. UNIX – UFS; Windows NT, 2000 i XP – FAT, FAT32 i NTFS; Linux – preko 40 fajl sistema, bazični je *extended file system* (ext2, ext3); CD-ROM (ISO 9660), DVD, flopi disk, ...

# Struktura fajl sistema

- ◆ Slojevitost implementacije funkcionalnosti fajl sistema odražava nivoe apstrakcije u preslikavanju korisničkih zahteva (sistemske pozive) u elementarne operacije sa I/O uređajem:



# Struktura fajl sistema

## ◆ Strukture podataka fajl sistema na disku:

- *boot control block*: na svakom volumenu tipično prvi blok na disku; sadrži informacije kako podići OS sa tog volumena; ako volumen nije *bootable*, ovaj blok je prazan (UFS: *boot block*, NTFS: *partition boot sector*)
- *volume control block*: na svakom volumenu ili particiji sadrži globalne informacije o volumenu, kao što su broj blokova, veličina bloka, broj slobodnih blokova, pozakivač na prvi slobodni blok, broj slobodnih FCB (*File Control Block*), pokazivač na prvi slobodni FCB itd. (UFS: *superblock*, NTFS: *master file table*)
- struktura direktorijuma za organizaciju fajlova (UFS: preslikava ime fajla u broj FCBa, NTFS ovo smešta u *master file table*)
- za svaki fajl njegov FCB (*File Control Block*) koji sadrži sve potrebne informacije (atribute) o fajlu, npr. veličinu, lokaciju, datum kreiranja i poslednje izmene, kreator, prava pristupa itd. (UFS: *inode*, NTFS: jedan ulaz u *master file table*)

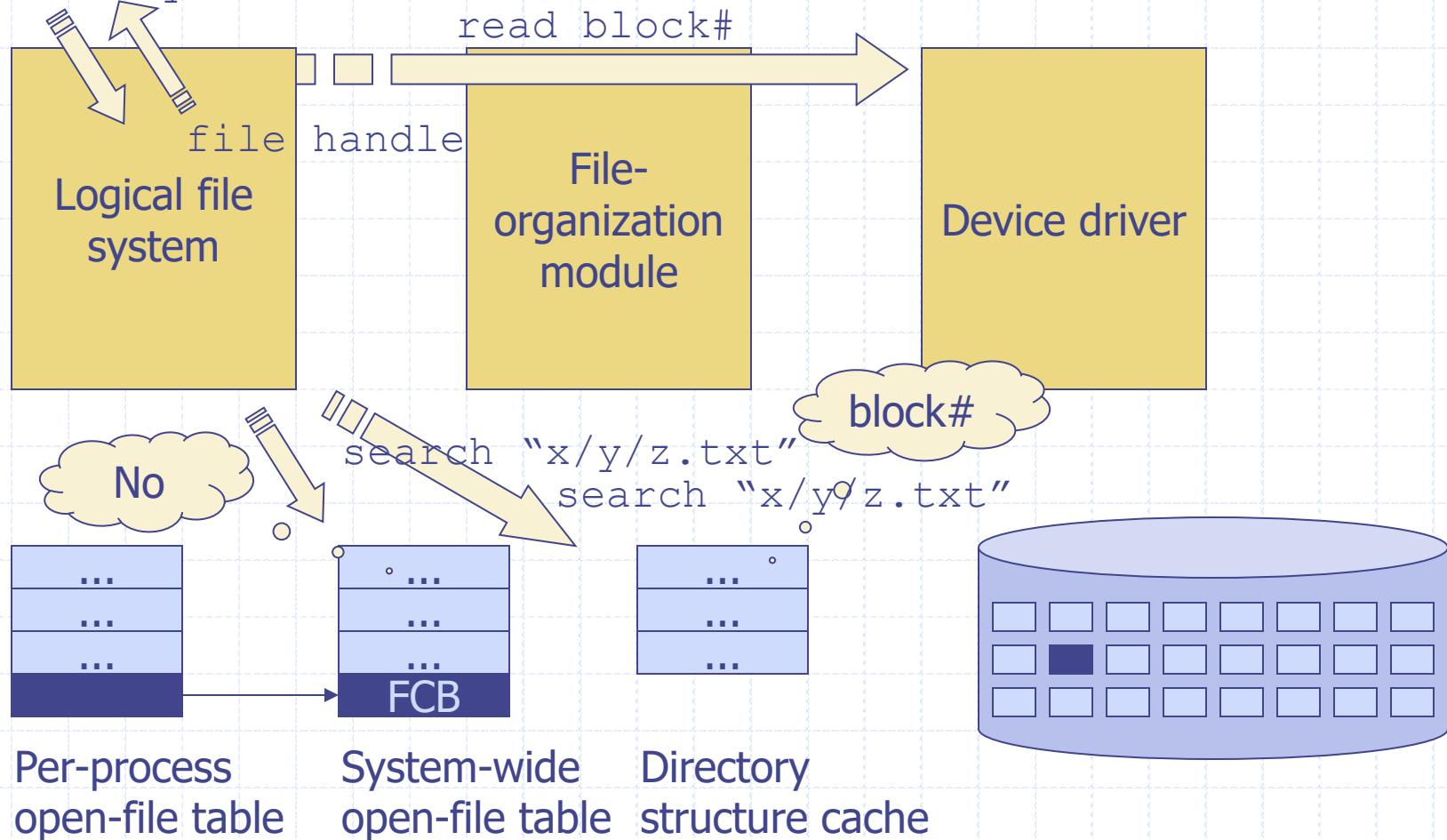
# Struktura fajl sistema

- ◆ Strukture podataka fajl sistema u operativnoj memoriji:
  - tabela montiranja: sadrži informacije o svakom montiranom volumenu
  - memorijski keš strukture direktorijuma za direktorijume kojima se skorije pristupalo
  - globalna sistemska tabela otvorenih fajlova za sve procese sadrži kopiju FCB za svaki otvoreni fajl, kao i druge potrebne informacije
  - tabela otvorenih fajlova za svaki proces čuva informacije o pristupu fajlu vezane za proces, npr. pokazivač na tekuću poziciju u pristupu, ograničenja prava pristupa, kao i pokazivač na globalnu tabelu otvorenih fajlova

# Struktura fajl sistema

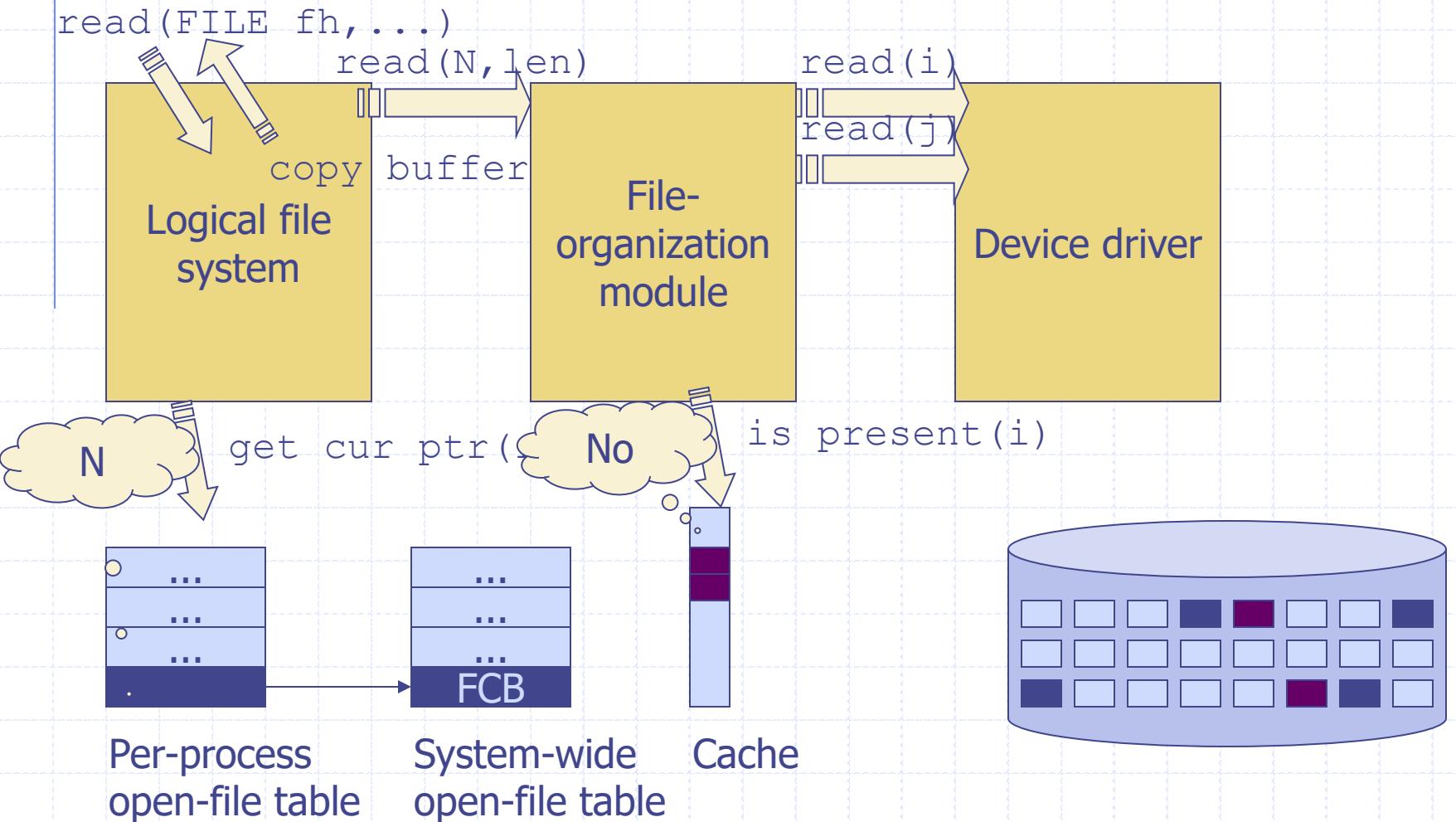
Primer implementacije sistemskih usluga: open (char\* filename)

open("x/y/z.txt")



# Struktura fajl sistema

Primer implementacije sistemskih usluga: `read(FILE filehandle, ...)`

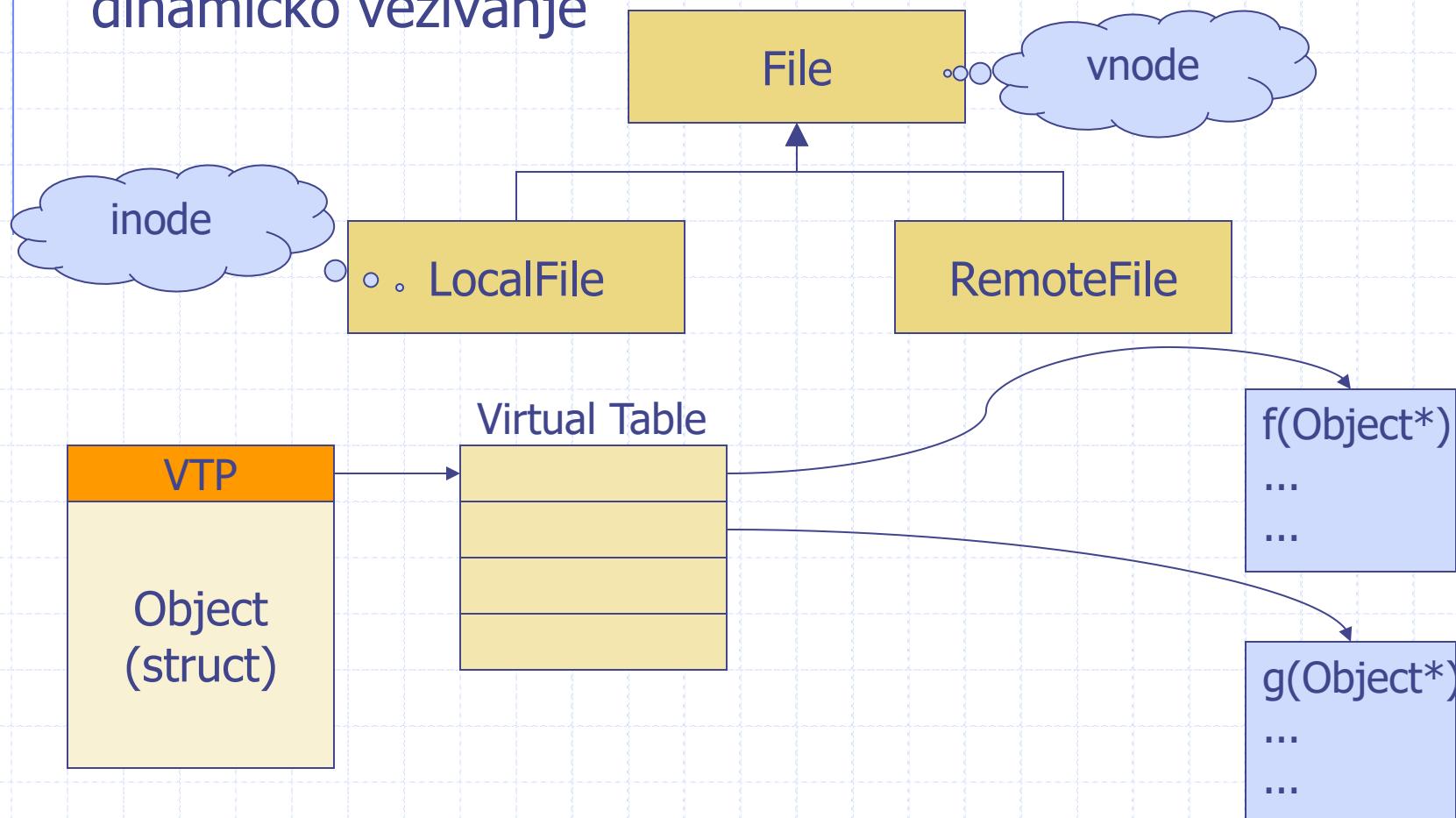


# Struktura fajl sistema

- ◆ Particija može biti:
  - "presna" ("raw") - ne sadrži fajl sistem: još nije uspostavljen fajl sistem (pre *formatizacije, formatting*) ili za posebne namene (npr. *swap space* ili za bazu podataka)
  - "pečena" ("cooked") ili formatizovana (*formatted*) - sadrži fajl sistem
  - *boot* – niz blokova u posebnom formatu koji se direktno učitavaju u memoriju i pokreće izvršavanje sa tačno određene lokacije u cilju učitavanja operativnog sistema
  - *korena (root)* – sadrži OS kernel i druge sistemske fajlove
- ◆ Mogućnost podrške više fajl sistema: *dual-boot*, svaka particija različiti fajl sistem, montiranje i udaljeni fajl sistemi
- ◆ Problem: kako efikasno omogućiti ovakvu heterogenost i transparentno ponuditi sve fajl sisteme u istoj lokalnoj strukturi direktorijuma?

# Struktura fajl sistema

- ◆ Virtuelni fajl sistem (*virtual file system*): OO pristup izolaciji različitosti fajl sistema iza istog interfejsa – polimorfizam i dinamičko vezivanje



# Implementacija direktorijuma

- ◆ *Direktorijum (directory)* je struktura podataka koja treba da obezbedi efikasno preslikavanje imena fajla (uključujući i poddirektorijum) u lokaciju tog fajla na disku
- ◆ UNIX na isti način tretira fajlove i direktorijume (jedan indikator u FCB /inode/ ukazuje na to) i ima iste operacije nad njima; na isti način se i smeštaju
- ◆ Windows razlikuje operacije nad direktorijumom i fajlovima
- ◆ Dva pristupa u realizaciji strukture i algoritama pretraživanja direktorijuma:
  - linearna lista
  - *hash* tabela

# Implementacija direktorijuma

## Linearna lista:

- ◆ Implementacija pomoću linearne strukture zapisa koji sadrže ime fajla i pokazivač na blokove sa podacima tog fajla
- ◆ Jednostavno za implementaciju
- ◆ Neefikasno pretraživanje – linearno kroz celu listu, i to za sve operacije (kreiraj novi, pronađi, obriši itd.)
- ◆ Različite fizičke implementacije:
  - ograničena veličina (statičko dimenzionisanje) – stariji OS
  - neograničena veličina (dinamička struktura) - češće
- ◆ Naprednije varijante, ali i dalje ili složene za implementaciju ili neefikasne za određene operacije: sortirana lista, B-stablo

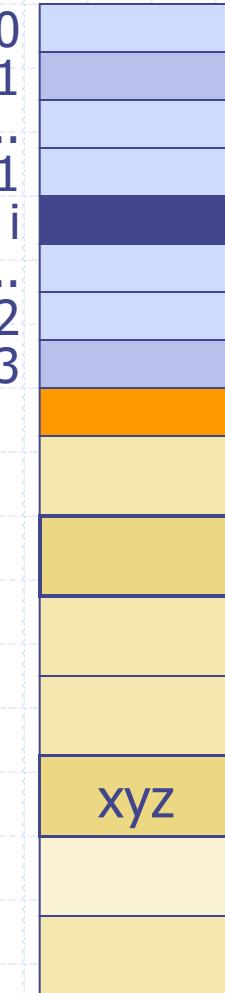
# Implementacija direktorijuma

## Hash tabela:

- ◆ Pored linearne strukture zapisa, koristi se i *hash* tabela koja preslikava ime fajla u pokazivač na strukturu u listi
- ◆ *Hash* tabela koristi neku funkciju za preslikavanje koja obezbeđuje dovoljno rasipanje ključeva (ime fajla) u skup indeksa u datom opsegu
- ◆ Rešavanje kolizija najčešće tako što svaki ulaz u *hash* tabeli pokazuje na zasebnu listu zapisa za ključeve koji se preslikavaju u taj ulaz
- ◆ Mnogo efikasnije od linearne liste

search ("xyz")

hash ("xyz") = i



# Metode alokacije

- ◆ Problem: kako alocirati prostor (blokove) i smeštati mnogobrojne fajlove na disku
- ◆ Pogodnost: direktni pristup bilo kom bloku
- ◆ Metode:
  - kontinualna alokacija
  - ulančana alokacija
  - indeksirana alokacija
- ◆ Kontinualna alokacija:
  - svaki fajl zauzima kontinualan niz blokova na disku
  - FCB sadrži samo broj prvog bloka i ukupan broj blokova koje fajl zauzima
  - direktni pristup *i*-tom bloku fajla je jednostavan: ako fajl počinje na bloku  $b$  diska, blok  $i$  fajla je u bloku  $b+i$  diska

# Metode alokacije

## ◆ Pogodnosti:

- jednostavan i efikasan direktni pristup
- jednostavan i efikasan sekvensijalni pristup

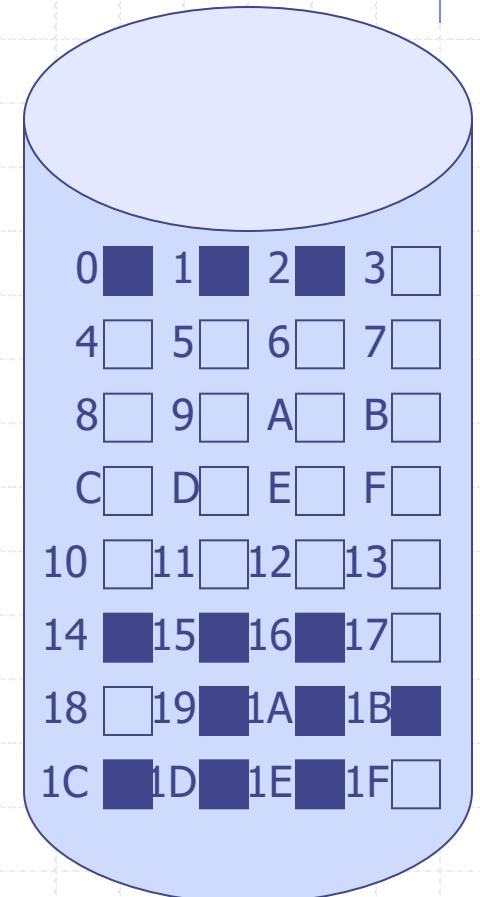
## ◆ Problemi:

- pronalaženje slobodnog prostora za smeštanje fajla – problem analogan kontinualnoj alokaciji memorije – dinamička alokacija (*first fit, best fit*)
- eksterna fragmentacija
- kako uopšte odrediti veličinu fajla u vreme kreiranja?

## ◆ Rešavanje eksterne fragmentacije:

- poseban postupak defragmentacije (kompakcije, *compaction*) relocira fajlove i sakuplja slobodan prostor; pokreće se uglavnom kada sistem nije u funkciji (*off-line*), ali moderni OS mogu ovo da rade i *on-line*

Directory		
File	Start	Size
mail	0	3
pic	14	3
addr	19	6



# Metode alokacije

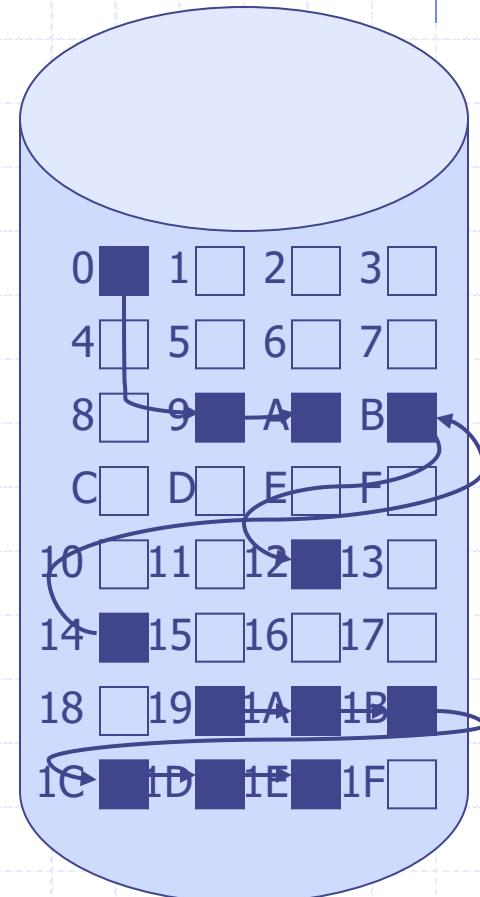
## ◆ Ulančana alokacija (*linked allocation*):

- svaki fajl je ulančana lista blokova rasutih bilo gde po disku
- svaki blok sadrži pokazivač na sledeći blok

## ◆ Pogodnosti:

- rešava probleme kontinualne alokacije (problem dinamičke alokacije i eksterne fragmentacije)
- operacije kreiranja i proširivanja fajla su jednostavne i efikasne; veličina fajla ne mora da se definiše prilikom kreiranja
- sekvensijalni pristup fajlu je jednostavan i relativno efikasan
- bilo koji slobodan blok se može koristiti za proširenje fajla
- fajl može da se proširuje sve dok ima slobodnog prostora

Directory		
File	Start	End
mail	0	A
pic	14	12
addr	19	1E



# Metode alokacije

## ◆ Nedostaci:

- izuzetno neefikasan direktni pristup fajlu: da bi se pristupilo bloku *i* fajla, potrebno je pristupiti svim prethodnim blokovima od početnog sledeći ulančane pokazivače
- pokazivači zauzimaju prostor na disku i oduzimaju ga korisnom sadržaju (može biti nezanemarljivo za velike diskove, odnosno velike pokazivače)
- osetljivost na otkaze: ako se oštete pokazivači zbog greške ili otkaza sektora, deo fajla može biti izgubljen ili potpuno pokvaren (ulančan u pogrešnu listu)

## ◆ Načini rešavanja:

- grupisati blokove u veće jedinice alokacije – *clusters*: smanjuje procenat prostora koji zauzimaju pokazivači i druge mehanizme čini efikasnijim, ali povećava internu fragmentaciju
- upisivati dodatne informacije uz pokazivač radi povećanja pouzdanosti (npr. ime fajla i redni broj bloka ili dvostrukе pokazivače)

# Metode alokacije

- ◆ Varijanta ulančane alokacije: *File-Allocation Table* (FAT, MS-DOS, IBM OS/2):
  - poseban deo svake particije zauzima FAT koja ima po jedan ulaz za svaki fizički blok na disku
  - svaki ulaz sadrži pokazivač na sledeći ulaz u lancu
  - direktorijum sadrži iste informacije kao i kod osnovne varijante
  - u suštini, pokazivači za ulančane liste se umesto u blokove smeštaju u FAT; blokovi sadrže samo podatke
- ◆ FAT – pogodnosti: jednostavni i efikasni algoritmi, uključujući i za direktni pristup ako je FAT keširan u memoriji ili je njegov veći deo u jednom bloku
- ◆ Mane:
  - ako FAT nije ceo keširan, može da uzrokuje mnogo šetanja glave diska za pristup susednim blokovima
  - veoma osetljiv na otkaze - bilo kakvo oštećenje u FAT uzrokuje velike štete

# Metode alokacije

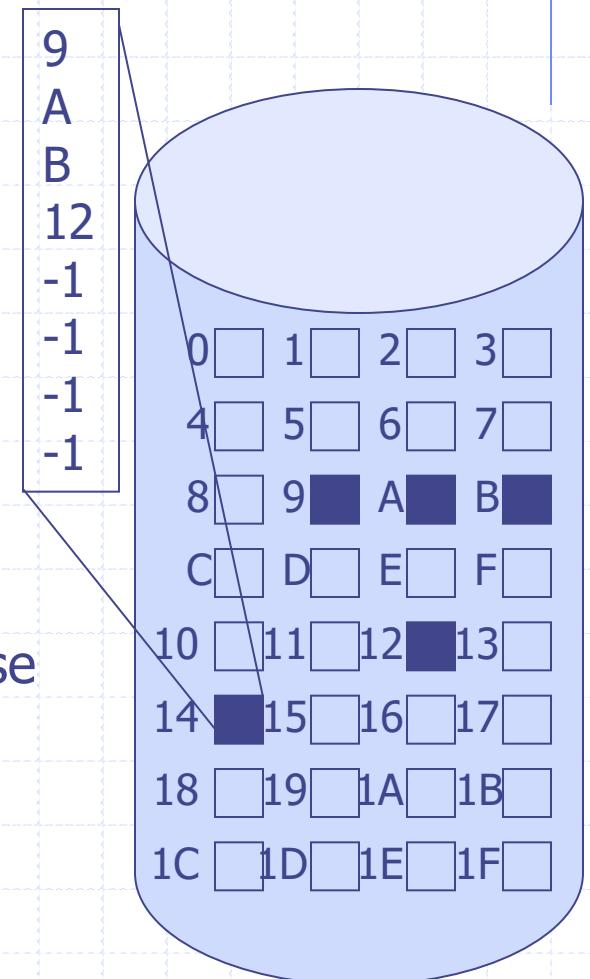
## ◆ Indeksirana alokacija (*indexed allocation*):

- svaki fajl ima svoj blok sa indeksom
- FCB sadrži broj tog indeksnog bloka
- indeksni blok sadrži spisak blokova koje fajl redom zauzima

## ◆ Pogodnosti:

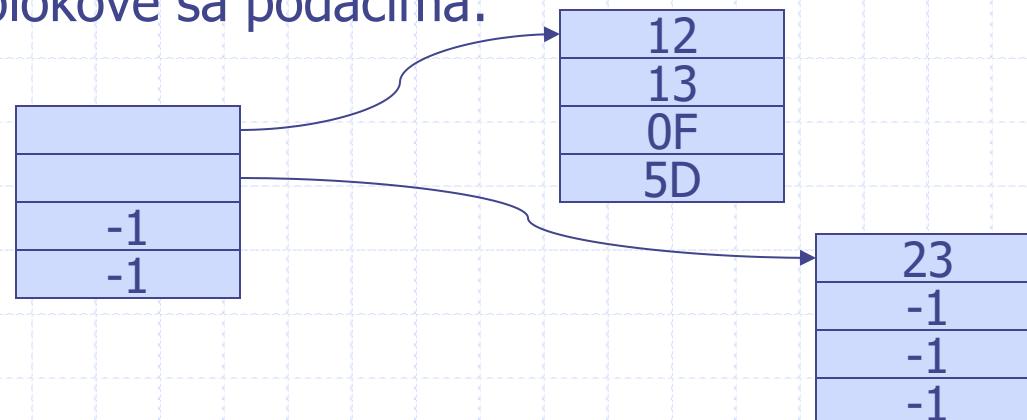
- jednostavan i efikasan pristup, uključujući i direktni (analogno straničenju)
- jednostavni i efikasni algoritmi za ostale operacije (kreiranje, proširenje)
- nije tako osetljiv na otkaze kao FAT
- nema eksterne fragmentacije, blok može da se alocira bilo gde
- sve pogodnosti ulančanog pristupa, samo što je direktni pristup efikasniji

Directory		File	Index	Size
mail		14		4



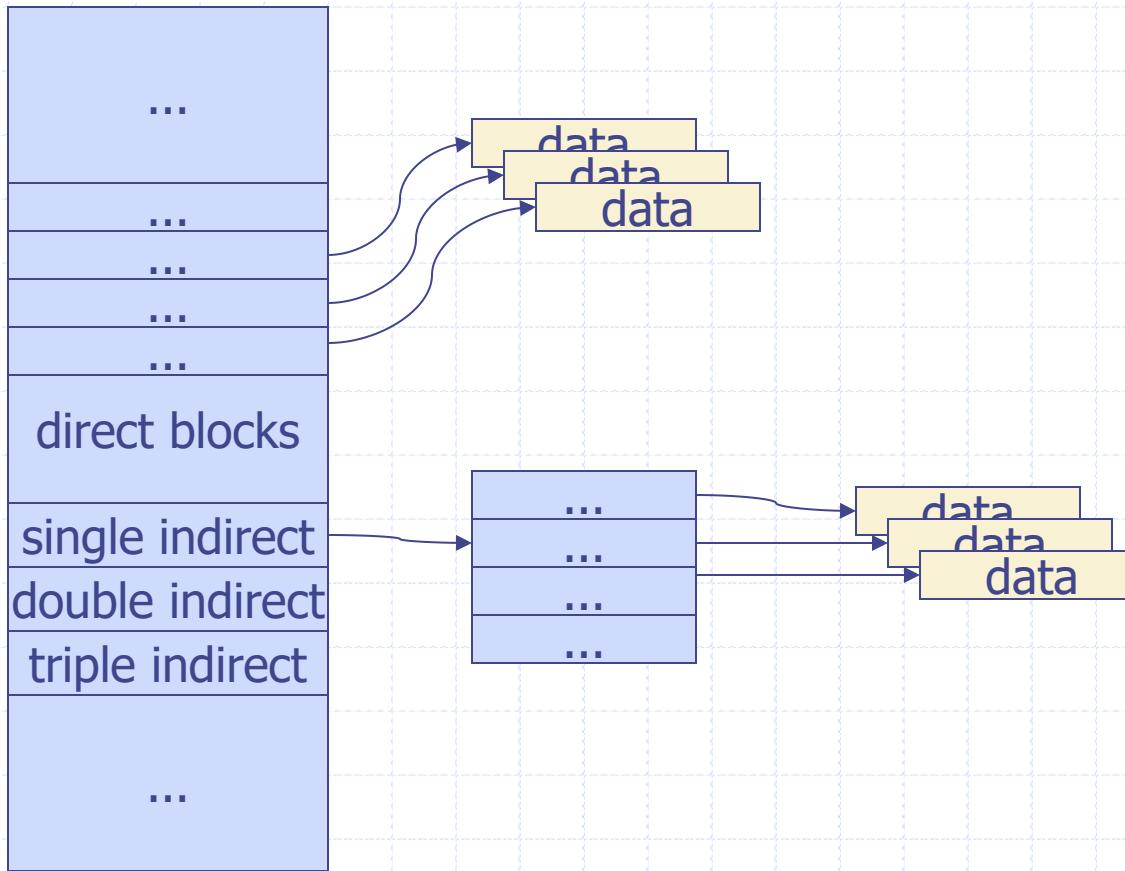
# Metode alokacije

- ◆ Nedostatak: veća potrošnja prostora na režijske informacije (indekse); da bi ta potrošnja bila manja, treba praviti manje indekse, ali manji indeksi dozvoljavaju manje fajlove
- ◆ Pristupi rešavanju:
  - ulančana šema: indeks je ulančana lista; podrazumevano zauzima jedan blok, ali po potrebi sadrži i pokazivač na naredni indeksni blok u ulančanoj listi
  - indeks u više nivoa: indeks prvog nivoa sadrži pokazivače na indeksne blokove sledećeg nivoa, a indeksi poslednjeg niva sadrže pokazivače na blokove sa podacima:



# Metode alokacije

- kombinovani pristup: primer – UNIX inode



# Metode alokacije

- ◆ Od načina alokacije jako zavisi efikasnost i performanse sistema
- ◆ Neki metod je pogodniji za neku vrstu pristupa i način korišćenja
- ◆ Zato neki sistemi upotrebljavaju različite načine alokacije za fajlove koji se unapred deklarišu sa određenom vrstom pristupa: sa ili bez ograničenja veličine, sa ili bez direktnog pristupa itd.; ako se želi promena načina upotrebe, fajl se konvertuje u drugačiji tip
- ◆ Neki sistemi upotrebljavaju kontinualnu alokaciju za male fajlove, prebacujući se na indeksirani način ako fajl poraste
- ◆ Postoje mnoge druge finije tehnike za poboljšanje performansi

# Rukovanje slobodnim prostorom

- ◆ Bez obzira na metod alokacije, fajl sistem mora da vodi evidenciju (spisak) slobodnih blokova:
  - kada traži slobodan blok, uzima se prvi blok sa ove liste (ili neki drugi, ako postoje optimizacije) i izbacuje iz spiska slobodnih
  - kada se fajl briše, svi blokovi koje je zauzimao fajl proglašavaju se slobodnim
- ◆ Bit-vektor - svaki fizički blok na disku predstavljen je jednim bitom (npr. 0-zauzet, 1-slobodan):
  - alokacija bloka podrazumeva traženje prvog bita 1 u vektoru; ako postoji podrška u instrukcijama procesora, veoma efikasno (Intel 80386 i Motorola 68020 imaju instrukcije koje vraćaju redni broj prvog bita 1 u mašinskoj reči)
  - dealokacija bloka podrazumeva resetovanje odgovarajućeg bita (izvesti računicu!)
  - pogodnost – jednostavno i efikasno
  - nedostatak: bit-vektor može da bude veoma veliki; npr. disk od 40GB sa blokom od 1KB zahteva vektor od preko 5MB

# Rukovanje slobodnim prostorom

## ◆ Ulančana lista:

- kao i kod ulančane alokacije, slobodni blokovi se ulančavaju u listu pokazivača koji se nalaze u svakom bloku
- alokacija jednog bloka je jednostavna i efikasna – uzima se prvi blok iz liste
- alokacija više blokova efikasna za ulančanu alokaciju, neefikasna za ostale – mora se prolaziti kroz ulančane blokove
- dealokacija pri brisanju fajla efikasna za ulančanu alokaciju
- FAT varijanta inherentno rukuje slobodnim prostorom – ulazi za slobodne blokove u FAT su posebno označeni

◆ Grupisanje: prvi slobodni blok sadrži spisak narednih  $n$  slobodnih blokova;  $n$ -ti iz tog spiska sadrži spisak sledećih  $n$  slobodnih blokova itd.

◆ Često se alociraju susedni blokovi, naročito kod kontinualne alokacije. U jednom zapisu liste slobodnih blokova može se čuvati samo adresa i veličina kontinualnog segmenta blokova

# Efikasnost i performanse

- ◆ Fajl sistem i I/O sa diskom su izuzetno osetljivi delovi i najčešće usko grlo u performansama sistema. Zato je potrebna njihova pažljiva konstrukcija i optimizacija
- ◆ Neki aspekti i tehnike:
  - prealokacija FCB-a na disk; unapred alocirati FCB-ove na volumenu (UNIX) kako bi kasnije operacije bile efikasnije
  - alocirati susedne blokove za isti fajl kako bi vreme pristupa bilo što kraće
  - upotreba klastera različite veličine radi smanjenja interne fragmentacije: npr. manji klaster za male fajlove i za kraj fajla
  - upotreba dinamičkih struktura neograničene dimenzije umesto ograničenih struktura sa statičkim dimenzijama
  - keširanje praktično neizostavno; za sekvencijalni pristup, *read-ahead* keširanje: učitavanje bloka unapred
  - asinhroni upis podataka (osim eventualno za metapodatke)

# Oporavak od otkaza

- ◆ Mnoge informacije o strukturi fajl sistema na disku čuvaju se u kešu u memoriji. Šta se dešava ako sistem otkaže pre nego što te podatke snimi na disk? – Korupcija fajl sistema
- ◆ Potreba za pokretanjem posebnih programa za oporavak od otkaza: analiziraju nekonzistentnu strukturu fajl sistema i pokušavaju da je restauriraju (UNIX fsck, MS-DOS chkdsk). Uspeh zavisi od fajl sistema i veličine problema
- ◆ Jedan pristup povećanju otpornosti na otkaze – sinhroni upis metapodataka na disk
- ◆ Obavezno redovno arhiviranje kopija (*backup*):
  - kompletan
  - inkrementalan – samo promene od zadatog datuma i vremena

# VII Zaključak

---

Šta je naučeno

Šta nas očekuje u Operativnim sistemima 2

Domaći zadatak

Ispit

Pitanja i diskusija