

Stranič enje

ukratko

Dio spremnika
za operacijski
sustav;
tu se nalaze i
tablice
prevođenja za
sve procese

Tablice
prevođenja

-	0	753	1
758	1	-	0
-	0	-	0
751	1	-	0
752	1	754	1
-	0	-	0
-	0	-	0
755	1	-	0
-	0	-	0
-	0	757	1
756	1	-	0

751	A4
752	A5
753	B1
754	B5
755	C3
756	C6
757	D2
758	A2
759	-

DISK

Proces A

A1
A2
A3
A4
A5

Proces B

B1
B2
B3
B4
B5
B6
B7
B8

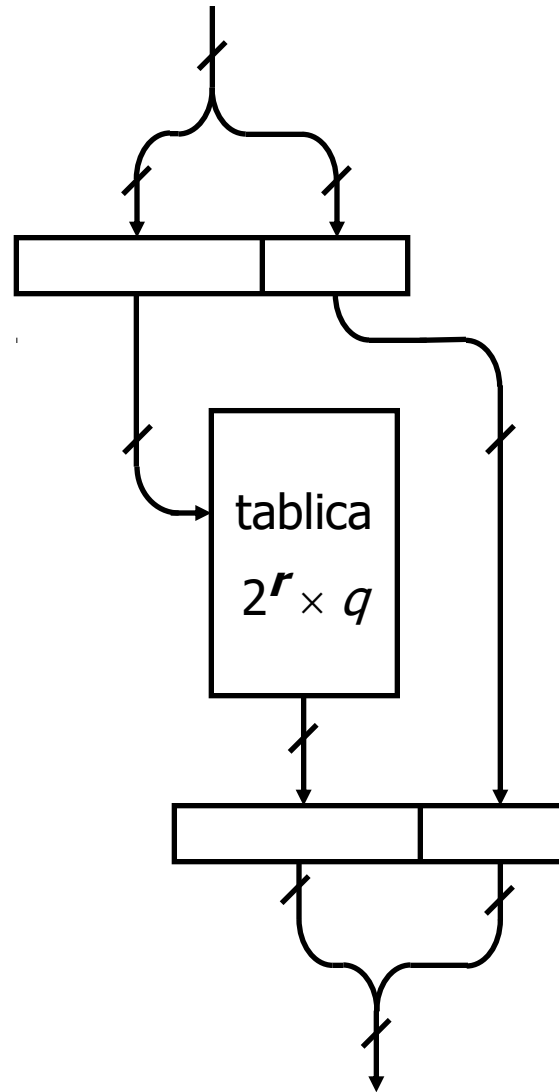
Proces C

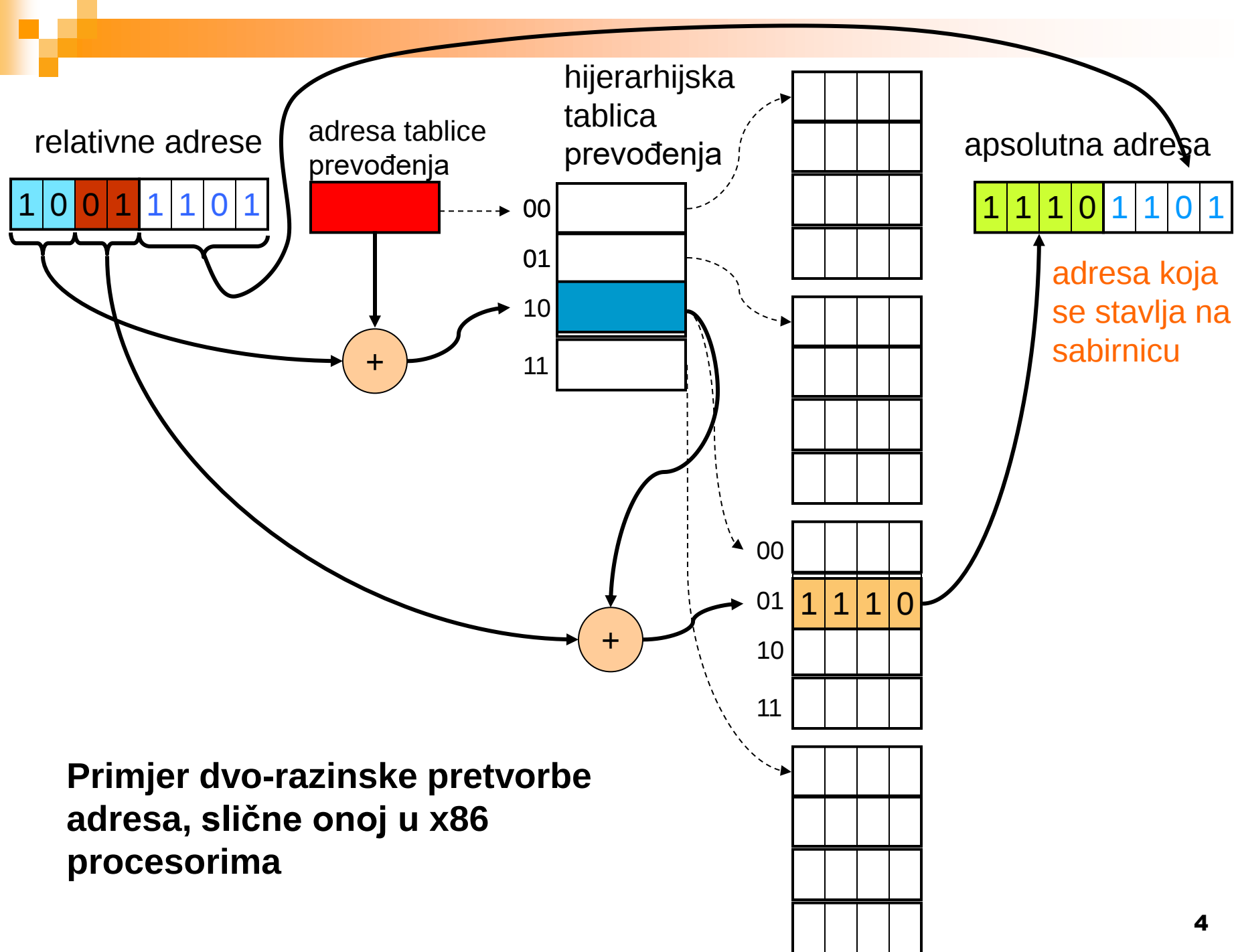
C1
C2
C3
C4
C5
C6

Proces D

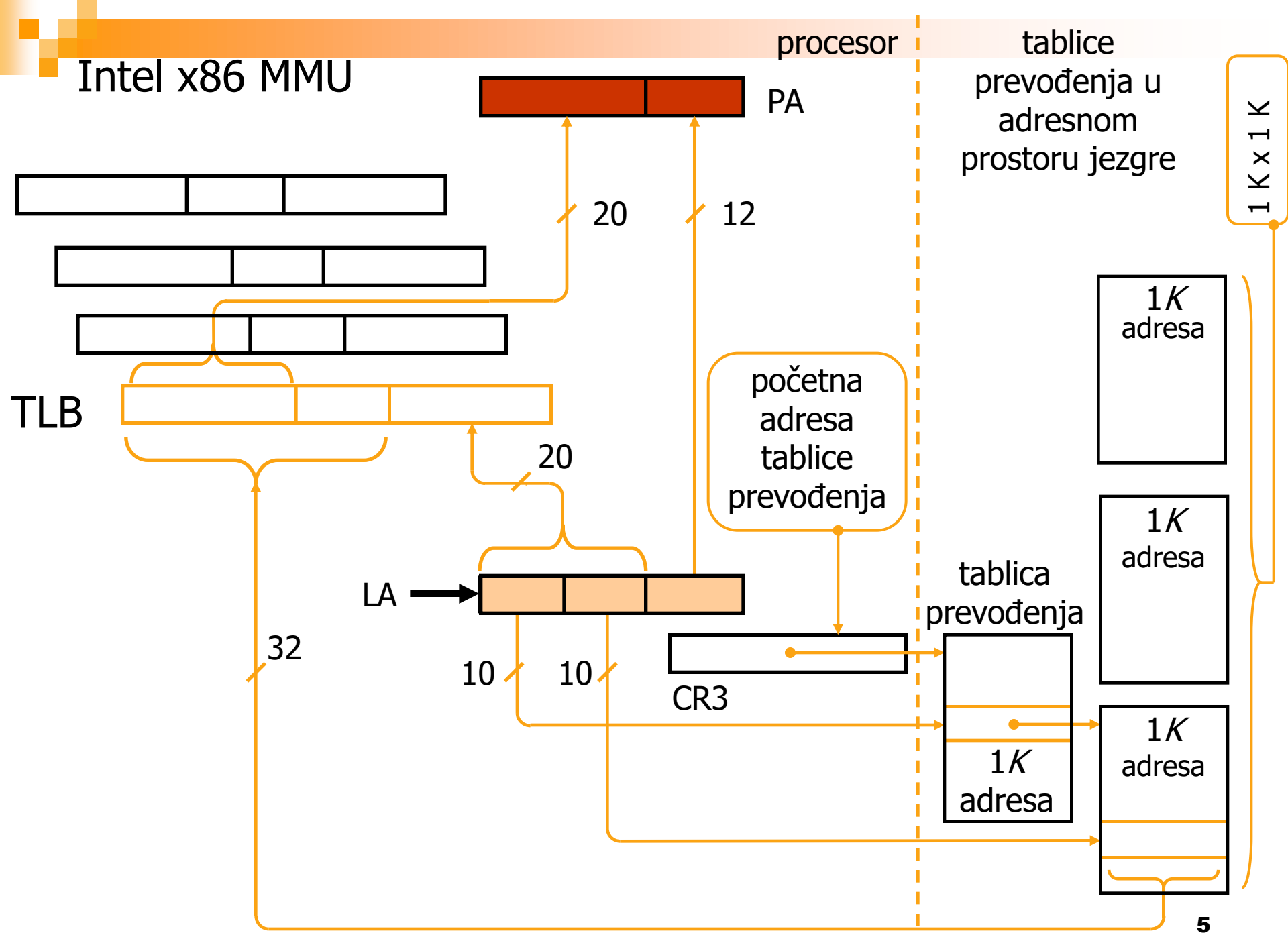
D1
D2
D3

Dio spremnika
za procese –
okviri koji npr.
započinju na
adresi 750000

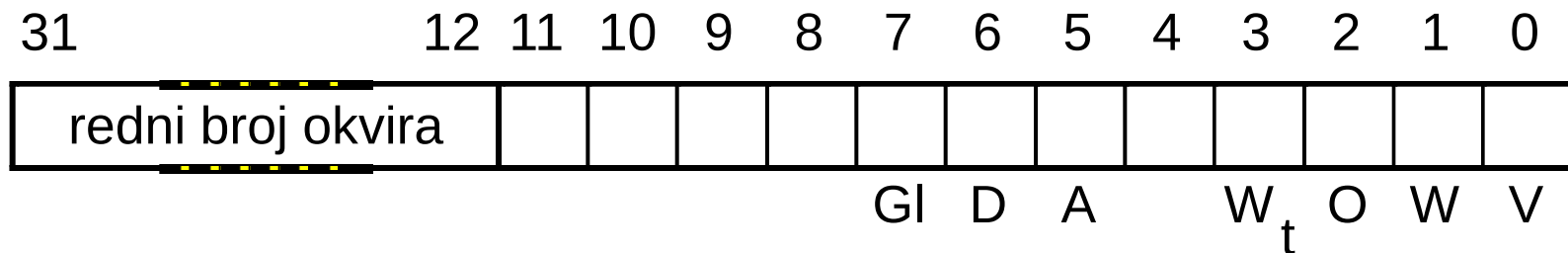




Intel x86 MMU



- Za svaku stranicu postoji zapis u tablici prevođenja:
 - redni broj okvira u kojem se stranica nalazi
 - razne zastavice, npr. za x86:



Zastavice:

V bit prisutnosti

W zaštita od promjene

0 za OS

W_t "write through"

A stranica je korištena

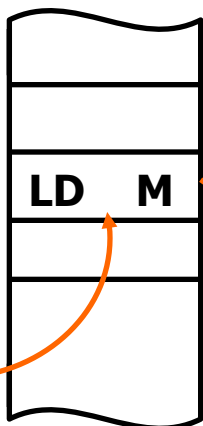
D "prljava" (dirty)

Gl globalna stranica

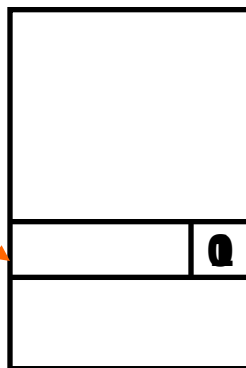
- Osim zapisa u tablici prevođenja, za svaku stranicu je potrebno zapisati i gdje se ona nalazi na pomoćnom spremniku

generirana adresa M pripada
stranici koja nije trenutno u
radnom spremniku

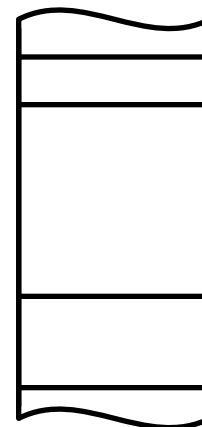
stranica u
spremniku



1



prazan
okvir

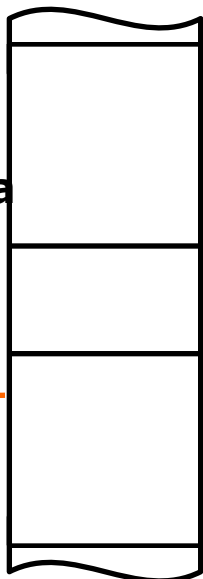


ponovi
instrukciju

6

tablica
prevođenja

jezgra



ažuriraj
tablicu
prevođenja

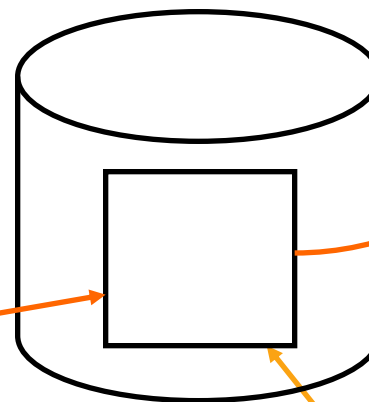
5

2

promašaj izaziva
prekid

4

učitavanje
stranice u prazni
okvir



stranica
na disku

dohvati stranicu
s diska

3

Zamjena stranica

- Kada su svi okviri popunjeni i dogodi se promašaj što napraviti?
 - očito je potrebno izbaciti neku stranicu iz nekog okvira!
 - kako odabrati stranicu za izbacivanje?
- Teorijske strategije:
 - FIFO
 - LRU
 - LFU
 - OPT

Satni algoritam

- Koristi se inačica LRU algoritma (*least recently used*)
 - izbaciti stranicu koja se već duže nije koristila
 - statistički se ta stranica niti neće još neko vrijeme koristiti
 - *satni algoritam (clock algorithm, second chance algorithm)*
 - okviri, tj. opisnici stranica koje se u njima nalaze, se kružno obilaze - promatra se **zastavica A** koja označava je li stranica korištena (engl. *accessed*)
 - ako je $A == 0$ stranica se izbacuje iz okvira i u njega se može staviti druga stranica
 - ako je $A == 1$, postavlja se $A = 0$ i pomiče se na idući okvir (trenutna stranica ostaje u okviru, daje joj se još jedna prilika obzirom da se nedavno koristila)

**Tablica
prevođenja**

okviri

0	0	2	1
1	99	0	1
2	100		0
3	199	6	1
4	200		0
5	299		0
	300		0
	399		0
	400		0
	499		0
	500		0
	599		0

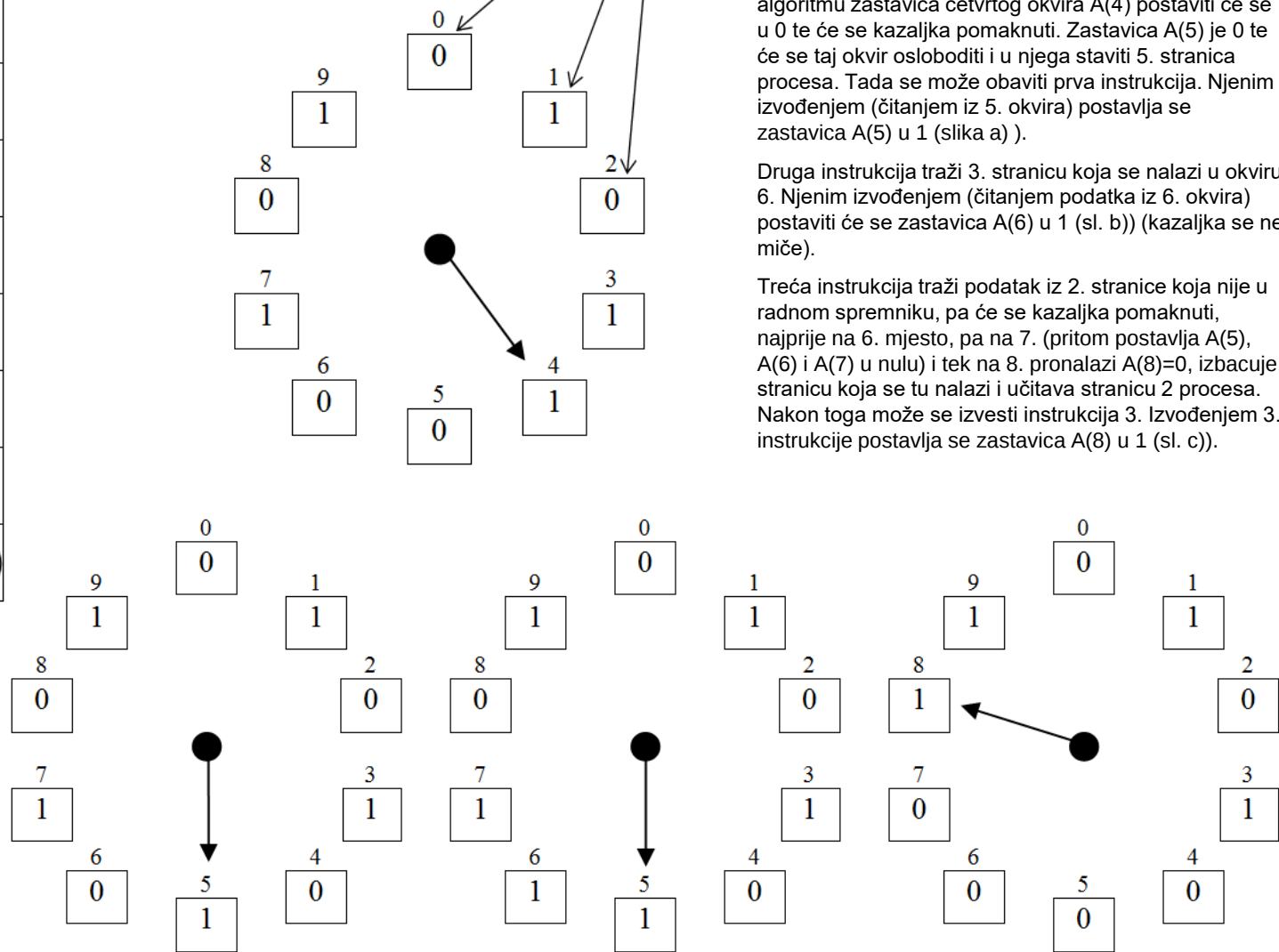
0	1
1	x
2	0
3	x
4	x
5	x
6	3
7	x
8	x
9	x

x - stranica
nekog drugog
procesa

Instrukcije (redom)

1:	LDR R1, (508)
2:	LDR R2, (332)
3:	LDR R3, (256)

zastavice A za
odgovarajuće okvire



Primjer rada satnog algoritma

Na prvu instrukciju dogoditi će se prekid zbog promašaja (traži se stranica 5 programa). Prema algoritmu zastavica četvrtog okvira A(4) postaviti će se u 0 te će se kazaljka pomaknuti. Zastavica A(5) je 0 te će se taj okvir osloboditi i u njega staviti 5. stranica procesa. Tada se može obaviti prva instrukcija. Njenim izvođenjem (čitanjem iz 5. okvira) postavlja se zastavica A(5) u 1 (slika a).

Druga instrukcija traži 3. stranicu koja se nalazi u okviru 6. Njenim izvođenjem (čitanjem podatka iz 6. okvira) postaviti će se zastavica A(6) u 1 (sl. b)) (kazaljka se ne miče).

Treća instrukcija traži podatak iz 2. stranice koja nije u radnom spremniku, pa će se kazaljka pomaknuti, najprije na 6. mjesto, pa na 7. (pritom postavlja A(5), A(6) i A(7) u nulu) i tek na 8. pronalazi A(8)=0, izbaciće stranicu koja se tu nalazi i učitava stranicu 2 procesa. Nakon toga može se izvesti instrukcija 3. Izvođenjem 3. instrukcije postavlja se zastavica A(8) u 1 (sl. c)).

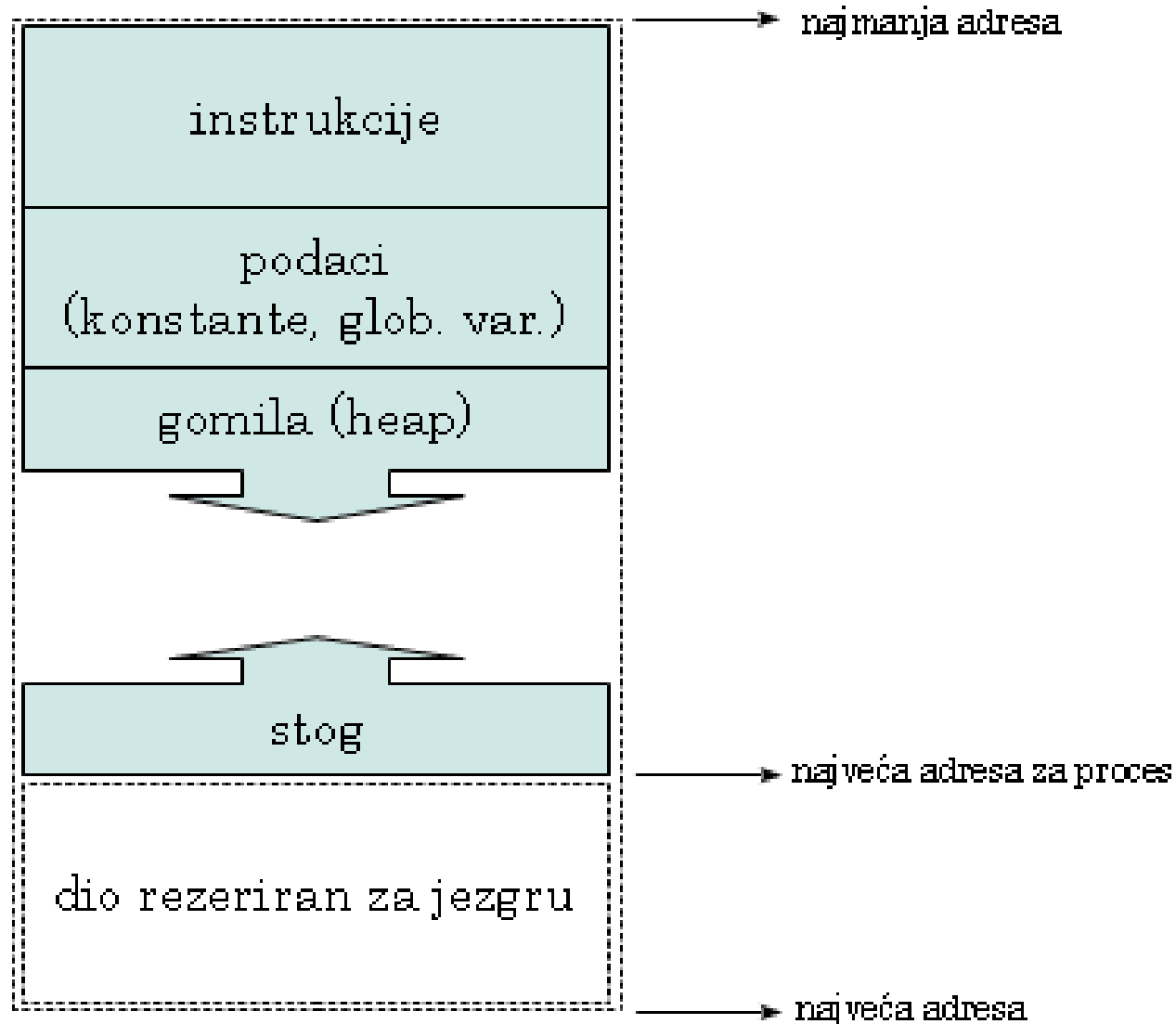
Dinamičko upravljanje spremnikom

dodatni materijali

Dinamičko upravljanje spremnikom u OS-u

- koristi se
 - za dodjelu adresnog prostora procesima
 - u sustavima koji ne koriste straničenje već “dinamičko upravljanje spremnikom”
 - za upravljanje prostorom na razini OS-a
 - koji su dijelovi za jezgru, za procese, za međuspremnike (buffere) naprava i slično
 - za upravljanje gomilom (heap) unutar procesa
 - u programima: `malloc/free`, `new/delete` i sl.
 - u nastavku razmatramo samo ovo upravljanje iako se slični postupci mogu koristiti i drugdje

Uobičajena organizacija adresnog prostora procesa



Gomila (heap)

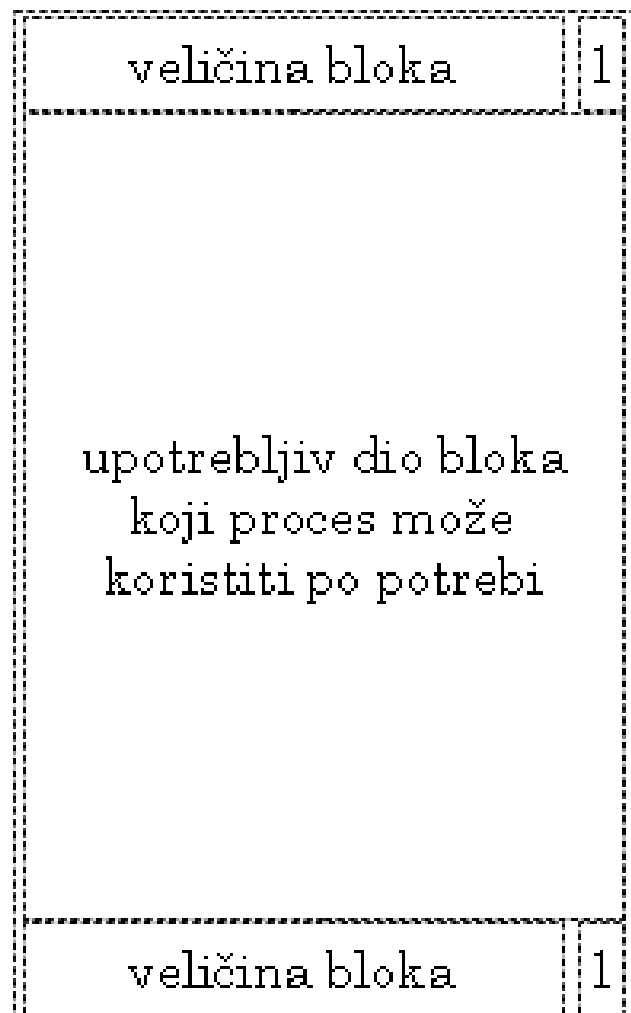
- služi za posluživanje dinamičkih zahtjeva procesa
 - pri pokretanju procesa nisu unaprijed poznati zahtjevi
 - veličina problema se zadaje/učitava naknadno
 - u C-u:
 - malloc/free (i slični)
 - u C++ (Java):
 - new/delete (i slični)
- veličina gomile raste s novim zahtjevima
 - adresa do kuda je trenutno gomila narasla naziva se *program break value* (pogledati *sbrk*)

Slobodni i zauzeti blokovi

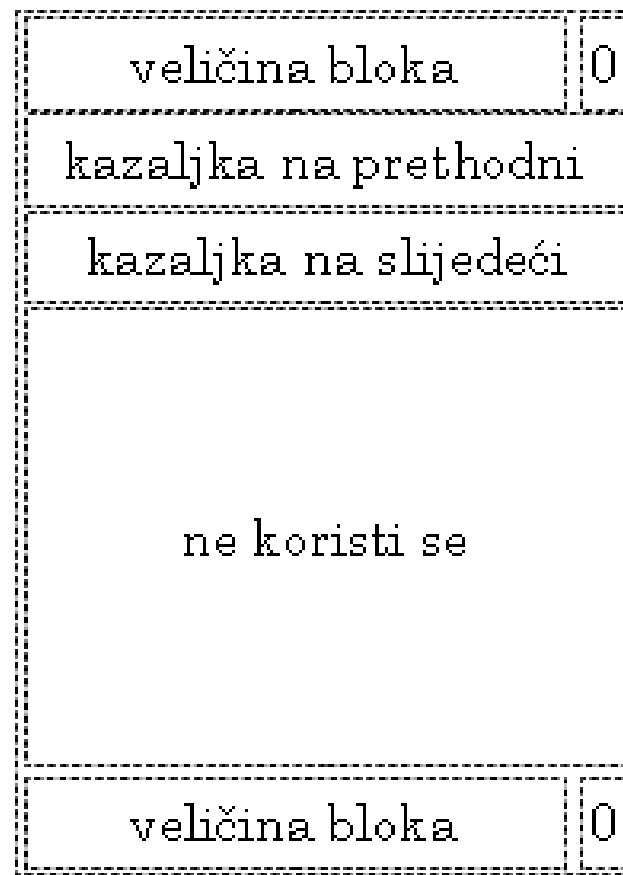
- potrebna struktura podataka sastoji se od:
 - liste slobodnih blokova
 - liste zauzetih blokova
 - nije neophodna; može služiti radi provjere rada algoritma (da se ne oslobađa nepostojeći blok)
- svaki blok mora imati zaglavlje sa:
 - veličinom bloka
 - oznakom zauzetosti
 - kazaljka na idući/prethodni blok
 - prema potrebi algoritma
- podnožje (na kraju bloka) ovisno o algoritmu
 - uglavnom su poželjna, radi jednostavnosti algoritma
 - zaglavlja što kraća da ne troše uzalud spremnički prostor

Primjer zaglavlja za zauzeti i slobodni blok

zauzeti blok



slobodni blok



Struktura podataka u C-u za prethodni primjer

```
/* veličine blokova moraju biti višekratnik od 2 da bi se  
   zadnji bit mogao iskoristiti za oznaku zauzetosti */
```

```
/* zaglavlja za zauzete blokove */
```

```
struct zag_zau {  
    int vel; /* veličina bloka uključuje zaglavlja */  
};
```

```
/* podnožje za zauzete blokove jednako je zaglavlju */
```

```
#define pod_zau zag_zau
```

```
/* zaglavlje za slobodne blokove */
```

```
struct zag_slo {  
    int vel; /* veličina bloka uključuje zaglavlja */  
    struct zag_slo *preth;  
    struct zag_slo *iduci;  
};
```

```
/* podnožje za slobodne blokove jednako je zaglavlju zauzetih */
```

```
#define pod_slo zag_zau
```

Neki makroi ...

```
/* B - adresa zaglavlja - početka bloka, A - adresa "korisnog" dijela */
#define ADR_KORISNO(B) ((void *)B)+sizeof(int) /*B-početak zauzetog bloka*/
#define ADR_BLOK(A)      ((void *)A)-sizeof(int) /* A-adr. iza zagl. u z.b.*/

#define ZB(B)      ((struct zag_zau *) (B)) /* adresa u tip zag_zau */
#define SB(B)      ((struct zag_slo *) (B)) /* adresa u tip zag_slo */
#define VEL(B)      ( ZB(B)->vel & (-1) ) /* vrati veličinu bloka */
#define ZAUZ(B)      ( ZB(B)->vel & 1 ) /* je li blok zauzet */
#define POSTAVI_VEL(B,V) do { ZB(B)->vel = (V) | ZAUZ(B); } while(0)
#define POSTAVI_ZAUZ(B) do { ZB(B)->vel = (ZB(B)->vel) | 1; } while(0)
#define POSTAVI_SLOB(B) do { ZB(B)->vel = VEL(B); } while(0)
/* idući/prethodni po adresama */
#define IDUCI_BLOK(B)      ( (void *) (B) + VEL(B) )
#define PRETH_BLOK(B)      ( (void *) (B) - VEL(ZB(B)-1) )
/* idući/prethodni po listi slobodnih blokova */
#define IDUCI_SLOB(B)      ( SB(B)->iduci )
#define PRETH_SLOB(B)      ( SB(B)->preth )

/* neki prevoditelji bi se mogli buniti na operacije tipa: ((void *) a + b)
 * tada napraviti: ((void *) ( (unsigned int) a + b )) */
```