

# Управление на оперативната памет

проф. д-р инж. Христо Вълчанов

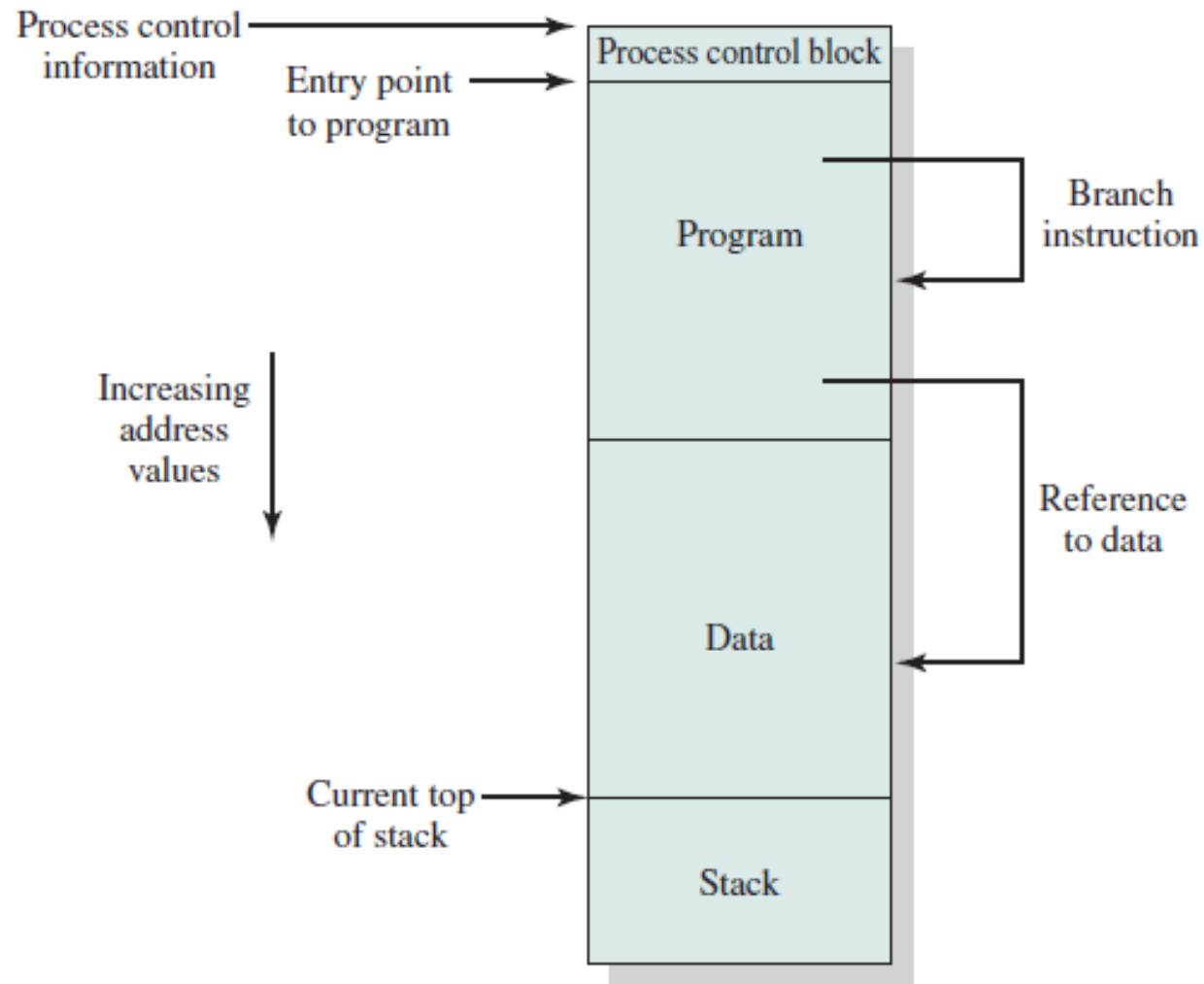
<http://cs.tu-varna.bg>

# Управление на паметта - изисквания

---

- Преместваемост
- Защита
- Споделяне
- Логическа организация
- Физическа организация

# Изисквания към адресите на процес

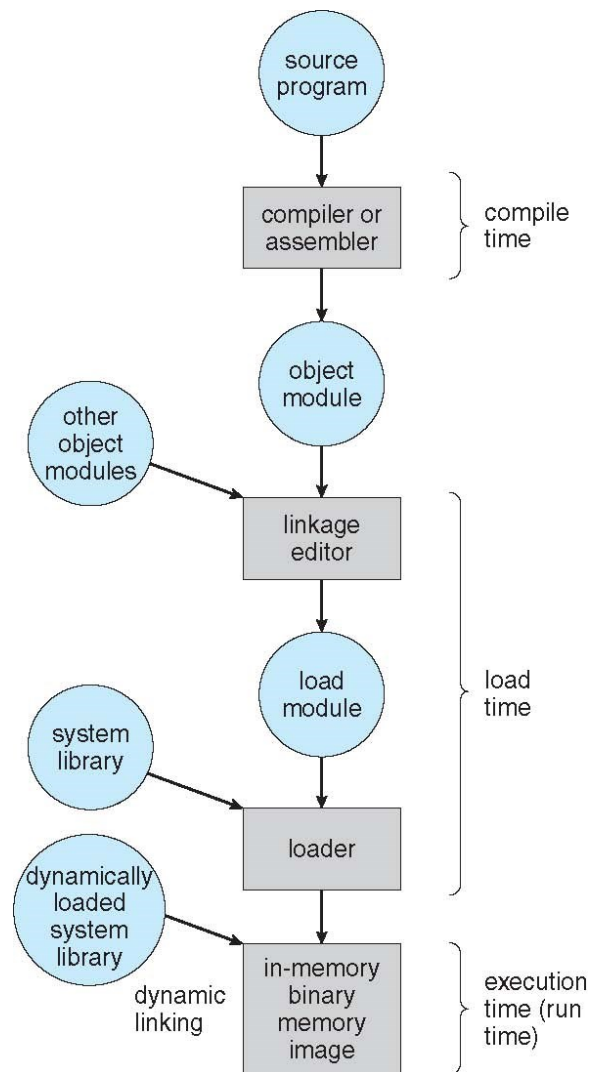


# Настройка на адресите в програмите

---

- **По време на компилация** – ако е известно предварително местоположението на програмата. Генерира се код в абсолютни адреси. При промяна на местоположението трябва да се прекомпилира.
- **По време на зареждане в паметта** - ако не е известно предварително местоположението на програмата. Генерира се код в преместваеми адреси.
- **По време на изпълнение** – адресите се изчисляват преди всяко обръщение към ОП. Изисква допълнителен хардуер.

# Етапи на обработка на програма



# Логически и физически адреси

---

- **Логически адрес** – генериран от CPU.
- **Физически адрес** – адрес от паметта

Те са:

- Едни и същи при настройка по време на компилация или зареждане;
- Различават се при настройка по време на изпълнение. В този случай логическият адрес се указва като **виртуален адрес**.

# Организация на паметта

---

- ОП трябва да поддържа и потребителски и процеси на ОС
- ОП е ограничен ресурс – трябва да се заема ефикасно

# Непрекъснатата организация на паметта

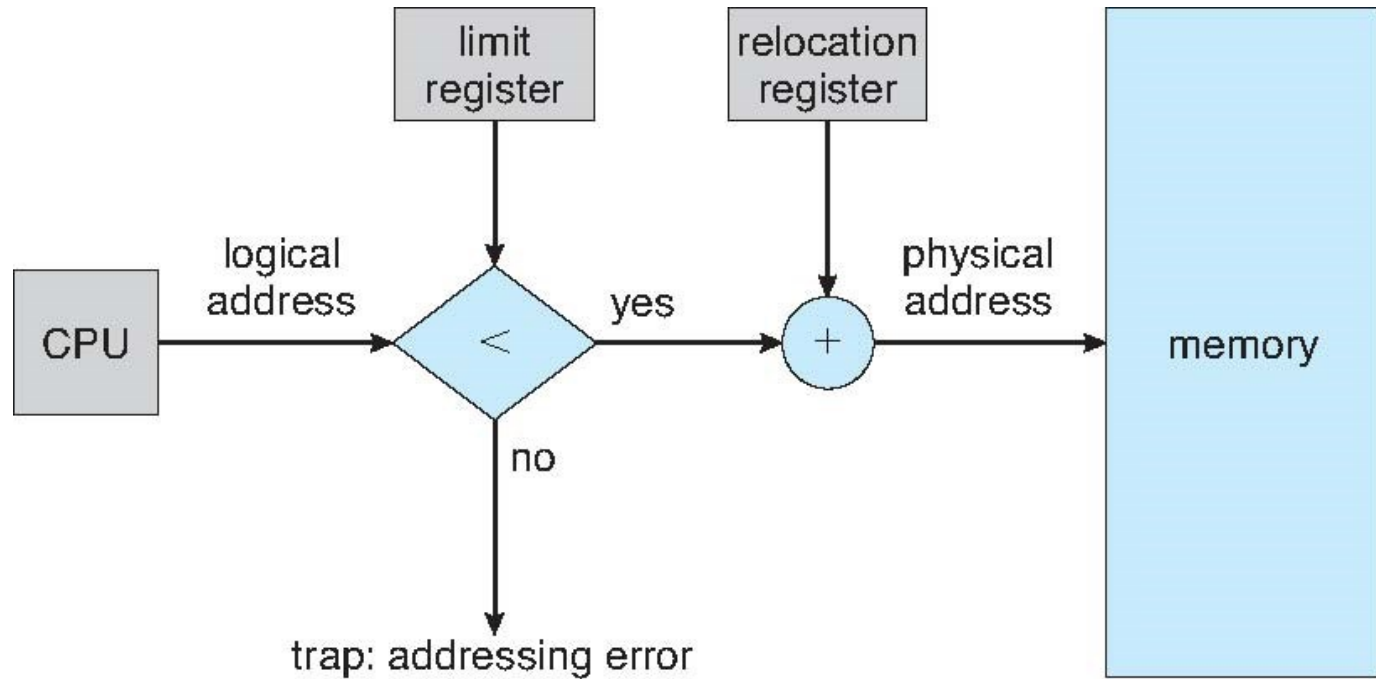
---

- ОП се разделя обикновено на два дяла:
  - Резидентна ОС, типично заемаща паметта в началото на адресното пространство
  - Потребителските процеси заемат горната част на паметта
  - Всеки процес се съдържа в единична непрекъсната секция от паметта
- Използват се регистри на преместване:
  - Базов регистър за начален адрес на разполагане;
  - Ограничителен регистър – всеки логически адрес трябва да бъде по-малък от стойността на този регистър



# Непрекъснатата организация на паметта

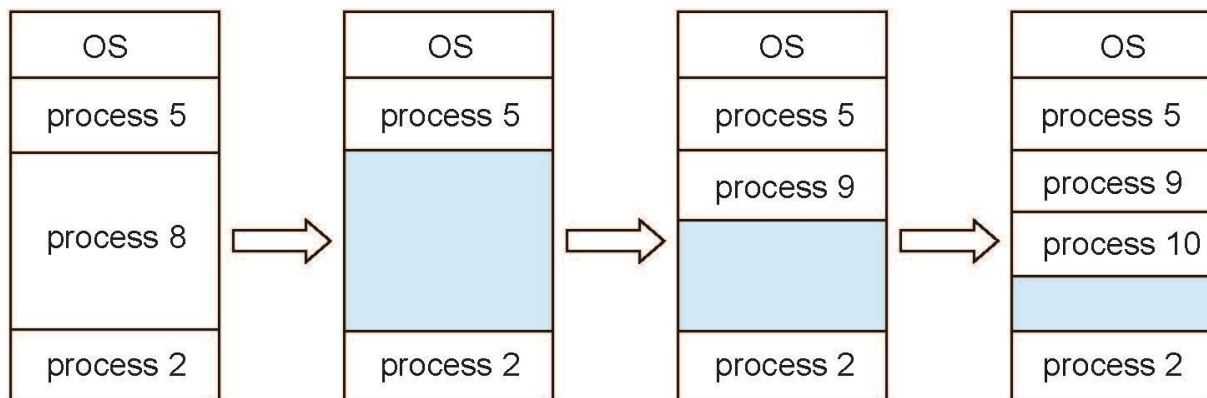
---



# Дялове на паметта

---

- Паметта за потребителските процеси се разделя на дялове.
- Всеки дял съдържа един процес.
- При завършване на процес, дялът се освобождава за друг процес.
- Възможност за получаване на фрагментиране



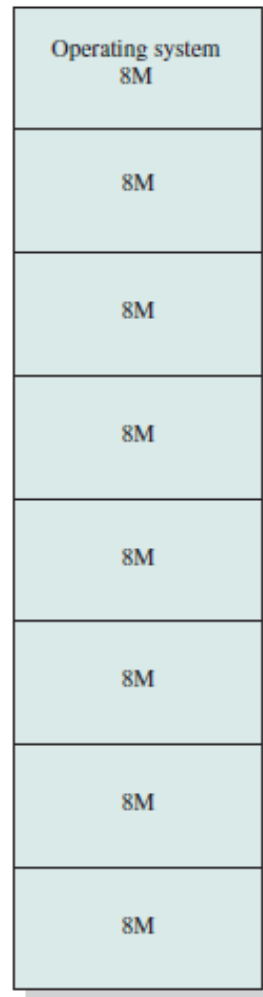
# Фрагментиране на паметта

---

- **Външно** – общият обем памет е достатъчен за процес, но пространството не е непрекъснато и не може да се използва.
- **Вътрешно** – заетата памет в дяла е по-малка от неговия размер.

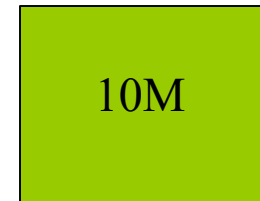
# Дялове с фиксиран размер

---



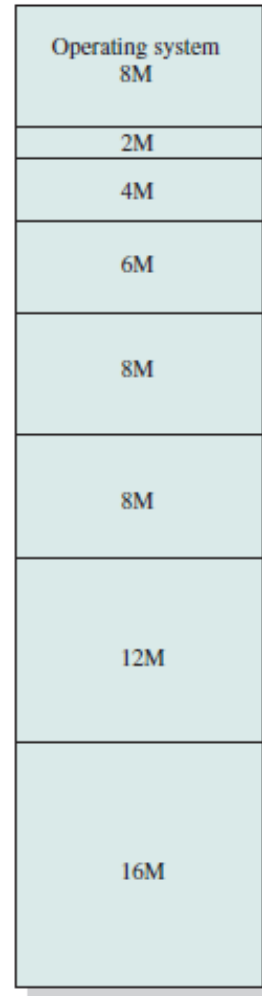
- Всички са с еднакъв размер

Нов процес



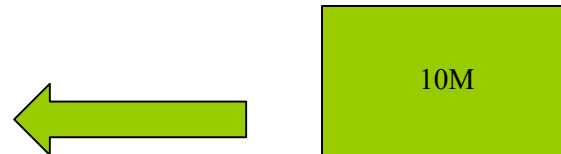
# Дялове с фиксиран размер

---



➤ Дяловете са с различен размер

Нов процес



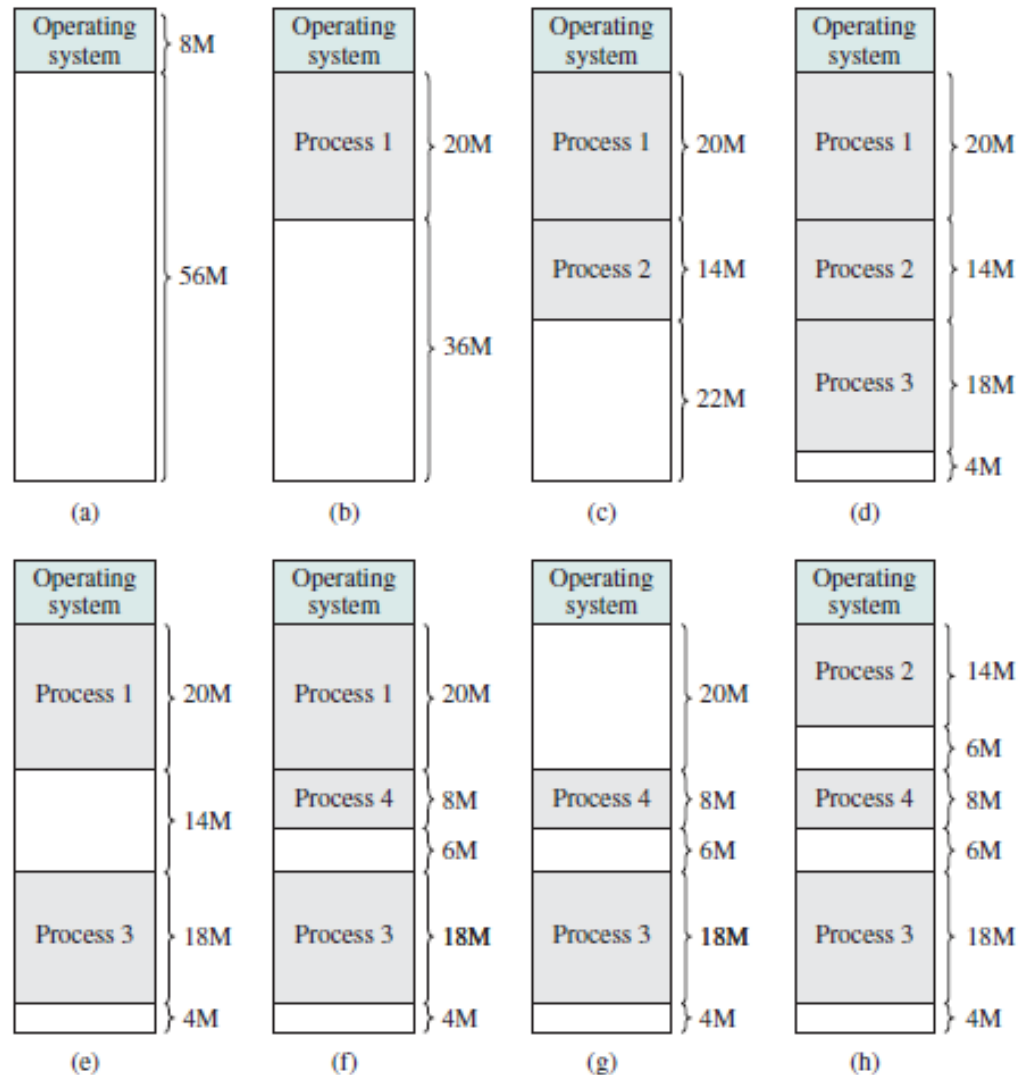
# Вътрешна фрагментация

---



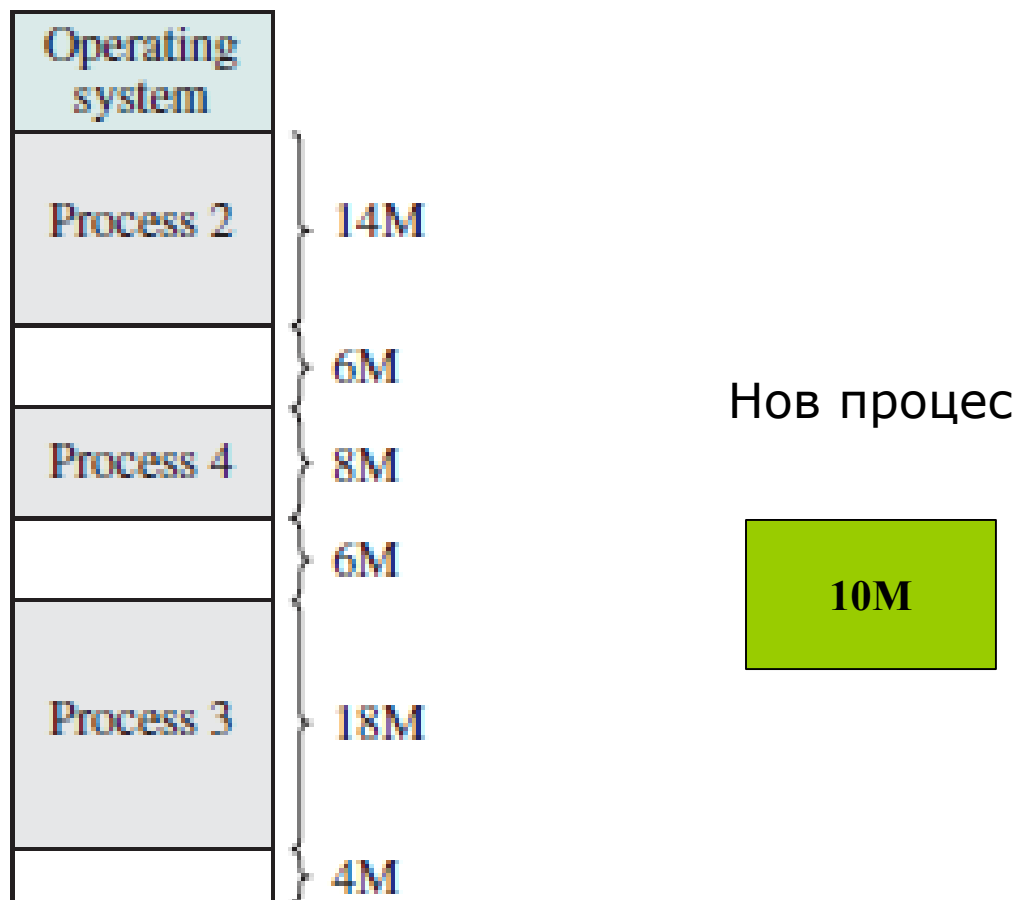
Вътрешно фрагментиране в дъла

# Дялове с динамичен размер



# Външна фрагментация

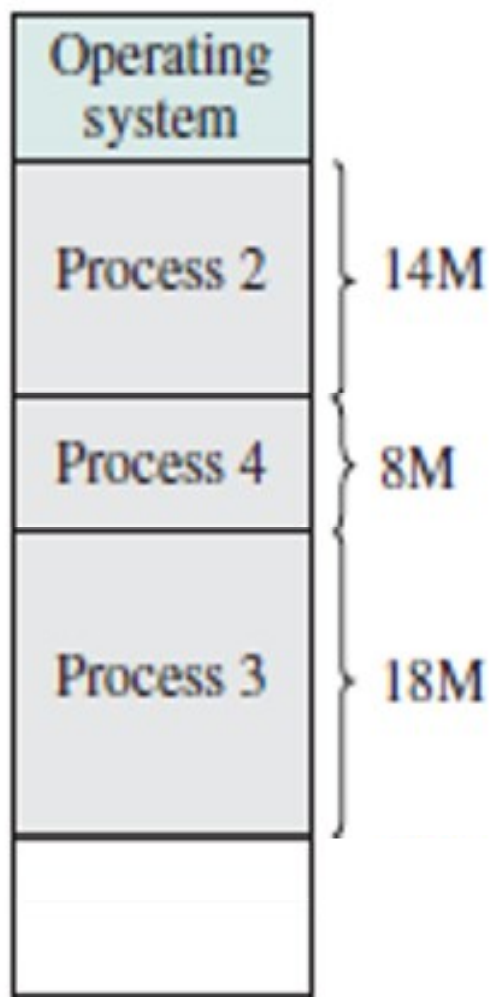
---





# Дефрагментиране на паметта

---

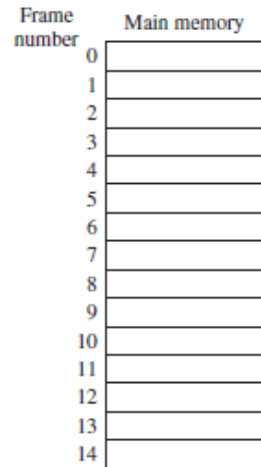


# Странична организация на ОП

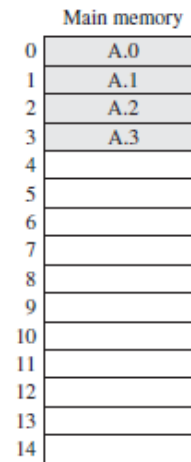
---

- Физическата памет се разделя на блокове с фиксиран размер – фреймове от 512 – 16Mb. Размерът на фрейма е степен на 2.
- Логическата памет се разделя на блокове със същия размер – страници.
- Използва се таблица на страниците за транслиране на логическия адрес.
- Елиминира се външната фрагментация.
- Възможна е вътрешна фрагментация.

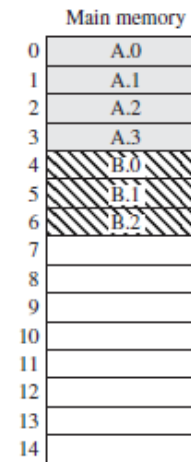
# Зареждане на процеси



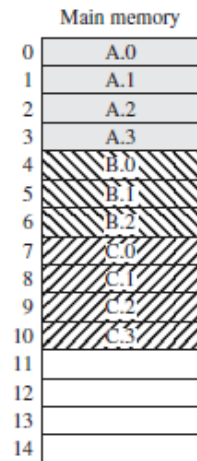
(a) Fifteen available frames



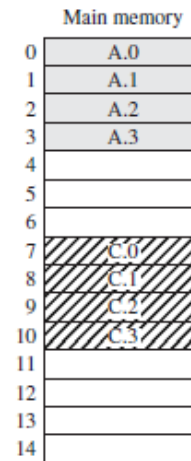
(b) Load process A



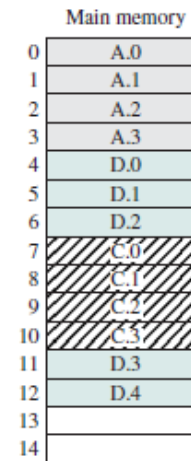
(c) Load process B



(d) Load process C



(e) Swap out B



(f) Load process D

# Таблици на процесите

---

0	0
1	1
2	2
3	3

Process A  
page table

0	—
1	—
2	—

Process B  
page table

0	7
1	8
2	9
3	10

Process C  
page table

0	4
1	5
2	6
3	11
4	12

Process D  
page table

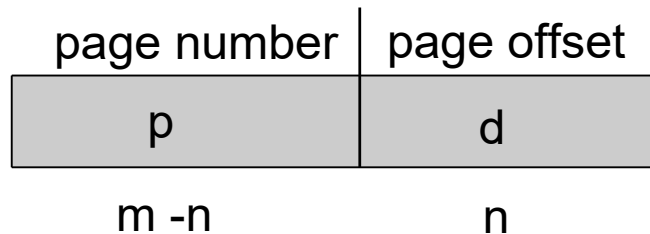
13
14

Free frame  
list

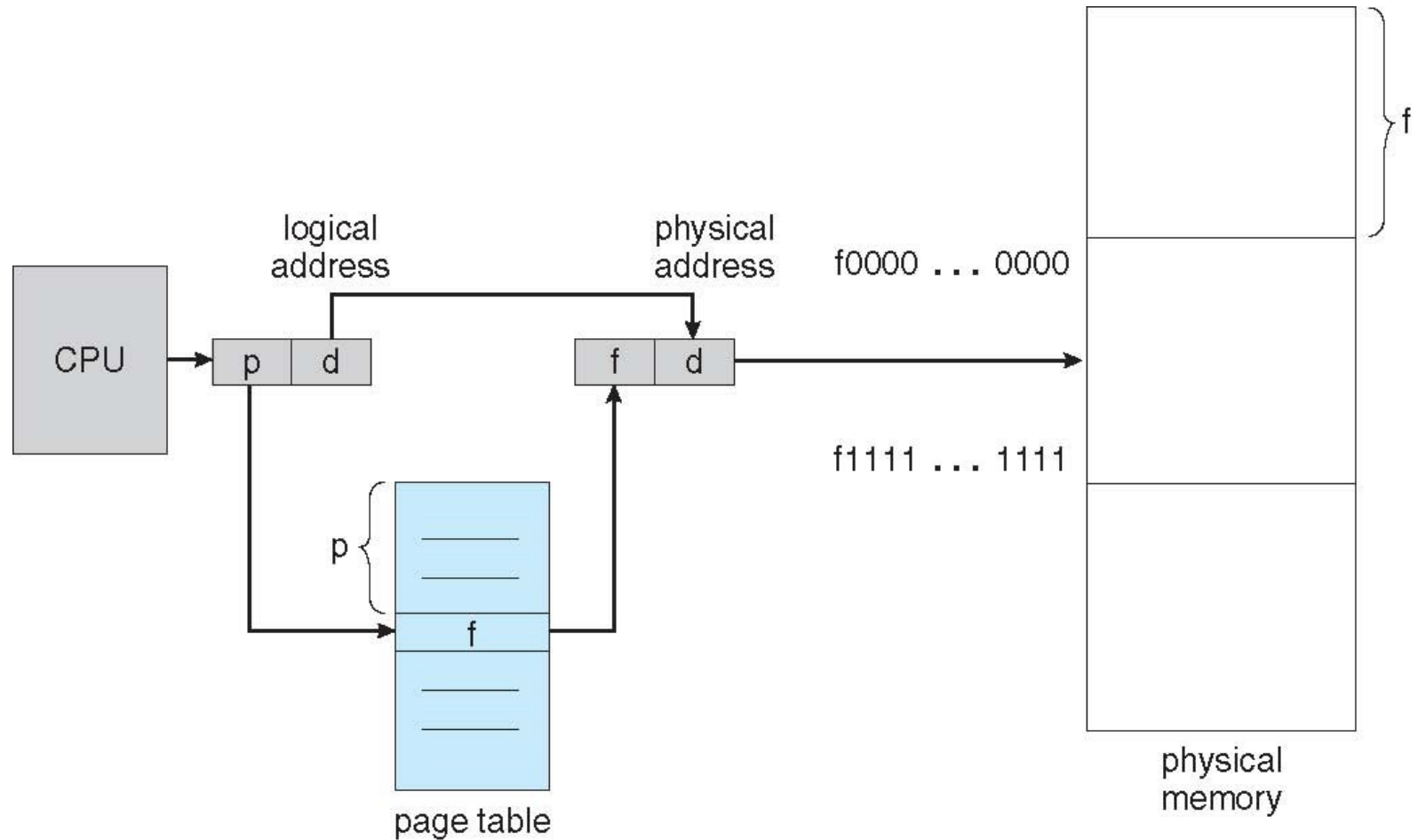
# Схема на транслиране

---

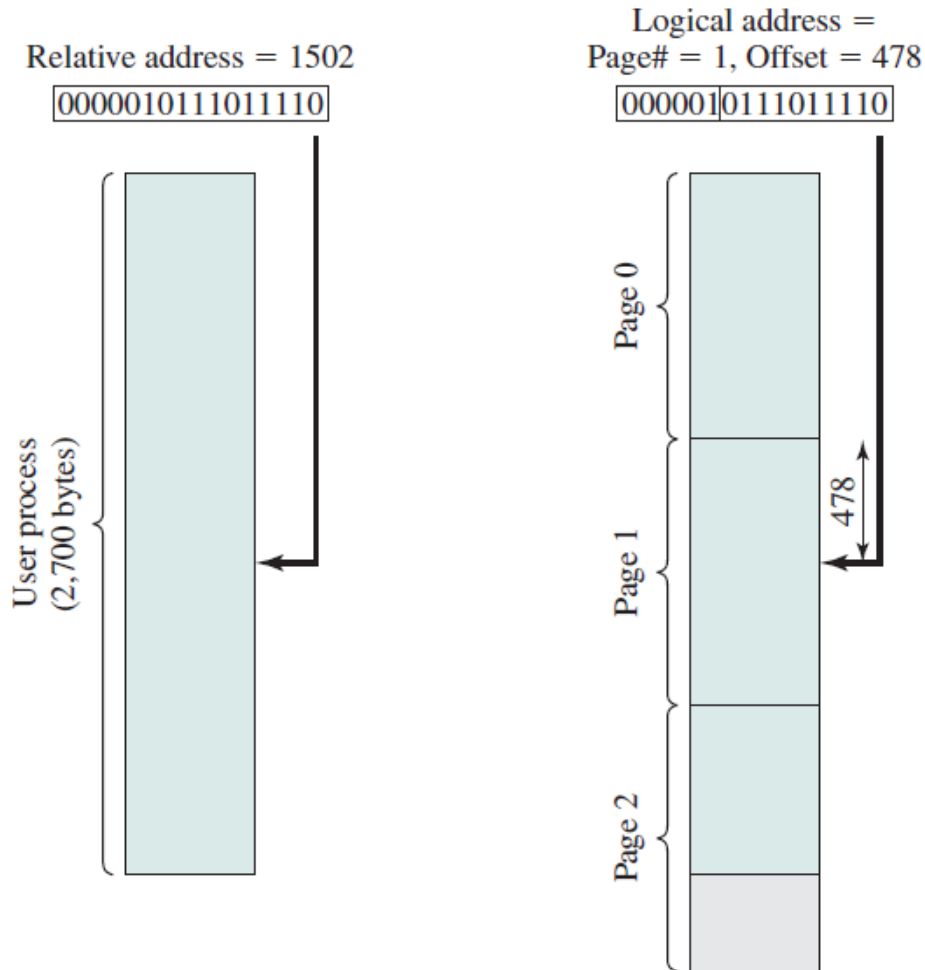
- Логическото адресно пространство е  $2^m$
- Размерът на страница е  $2^n$



# Схема на транслиране



# Изчисляване на адресите



# Асоциативна памет

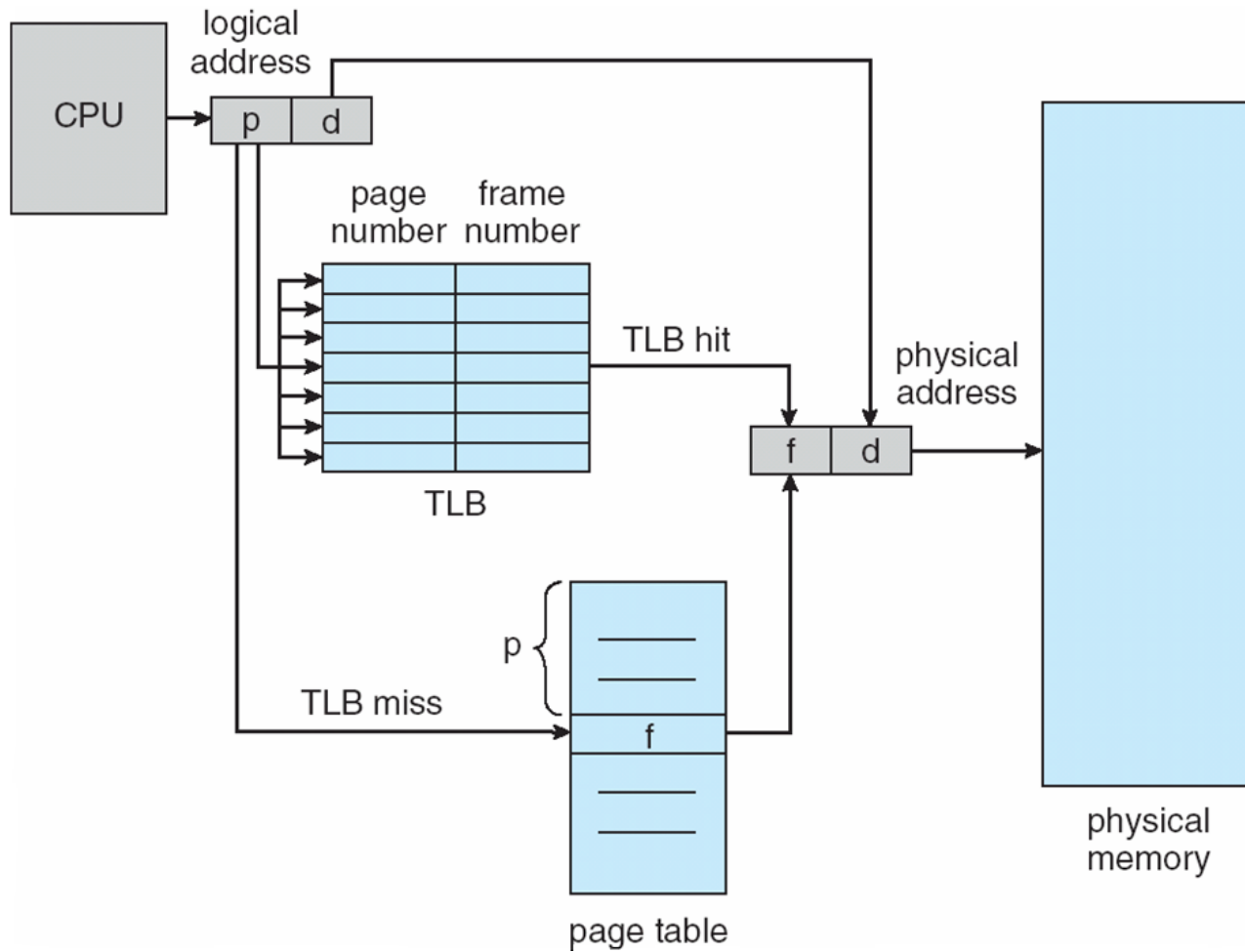
---

- Ако таблицата на страниците е в ОП е необходим два пъти достъп до паметта – един за таблицата и един за инструкцията/данните.
- Използва се бърза кеш памет – Translation Look-aside Buffer (TLB).
- Асоциативна памет – паралелно търсене

Page #	Frame #



# TLB транслиране



# Таблица на страниците

---

Използване на голям обем памет за таблиците.

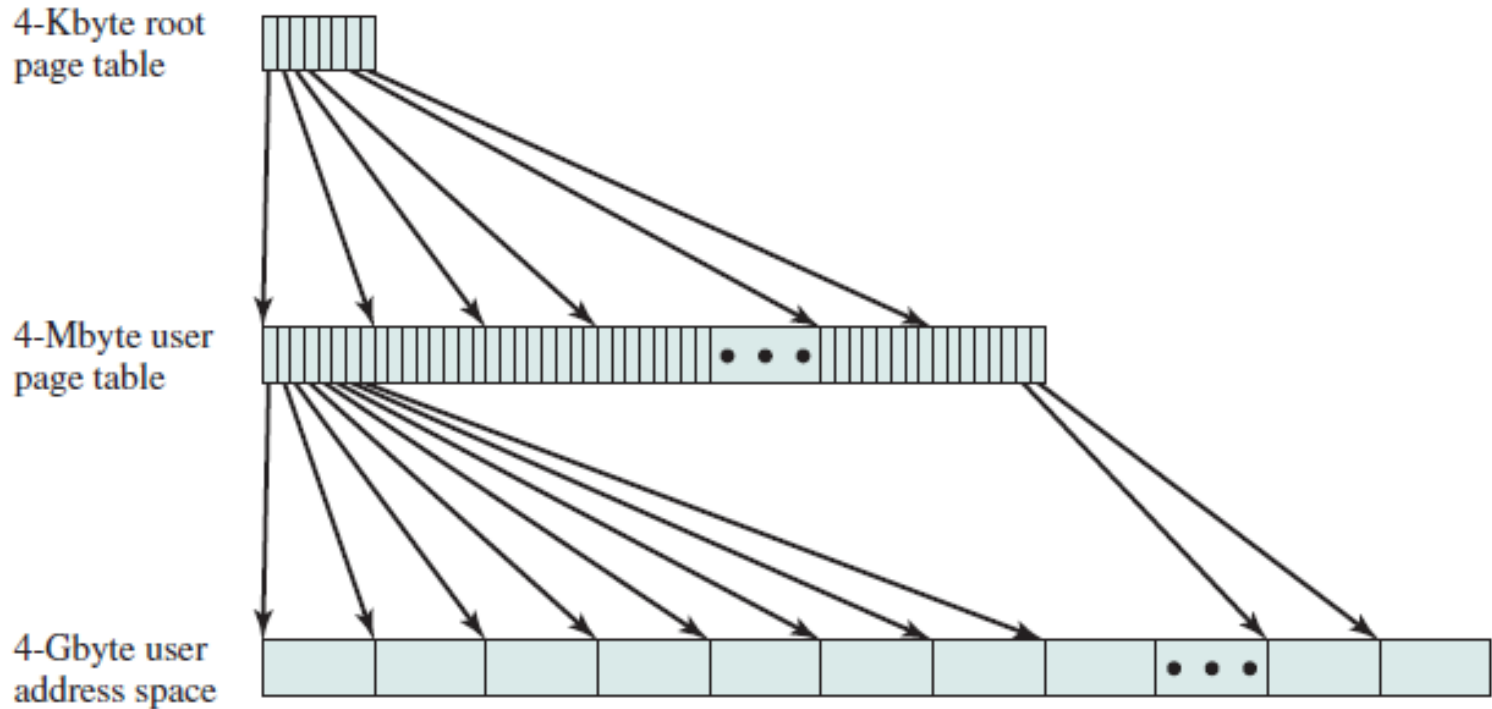
*Пример:*

- Адресно пространство – 32 битово ( $2^{32}$ );
- Размер на страница – 1K ( $2^{10}$ );
- Брой елементи на таблицата – 4 милиона ( $2^{32} / 2^{10}$ );
- Всеки елемент е 4 bytes -> Размер на таблицата 16Mb.

➤ При 64 битово адресно пространство?

# Странициране на нива

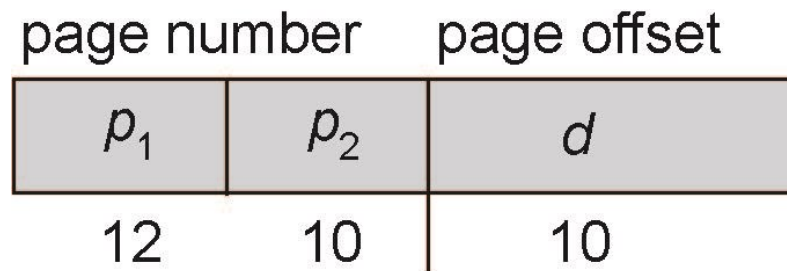
---



# Странициране на две нива

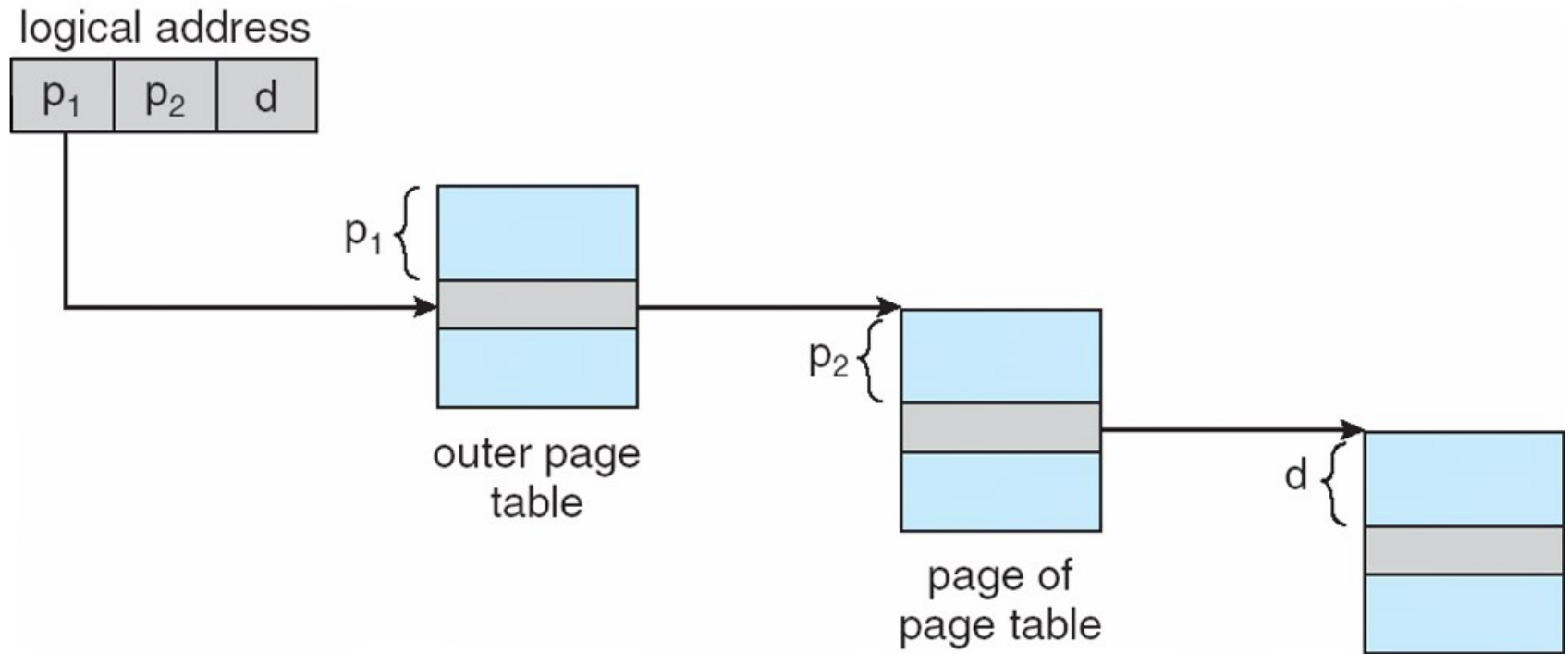
---

- Логическият адрес (32 bits), страница 1K се разделя:
  - номер на страница (22 bits)
  - отместване в страница (10 bits)
- Номерът на страница се разделя:
  - номер на страница (12 bits)
  - отместване в страницата (10 bits)



- **Forward-mapped Page Table**

# Адресна схема

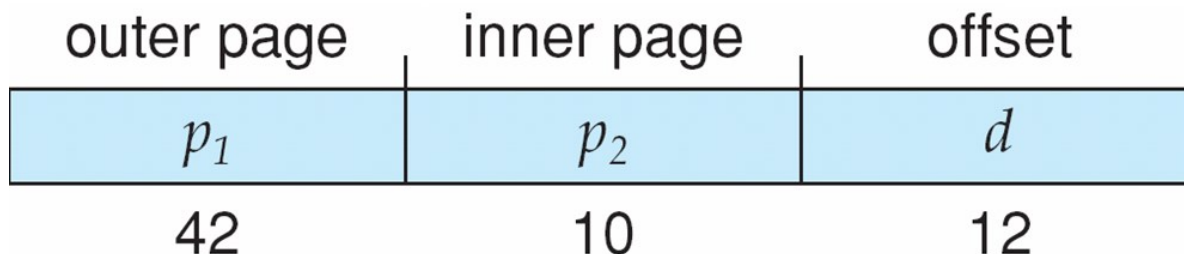


# Адресно пространство 64 bits

---

Пример:

- Странициране на две нива;
- Размер на страница – 4К ( $2^{12}$ );
- Брой елементи на таблицата –  $2^{52}$ ;
- Всеки елемент е 4 bytes
  - > inner table е с  $2^{10}$  елемента
  - > outer table е с  $2^{42}$  елемента



# Странициране на три нива

---

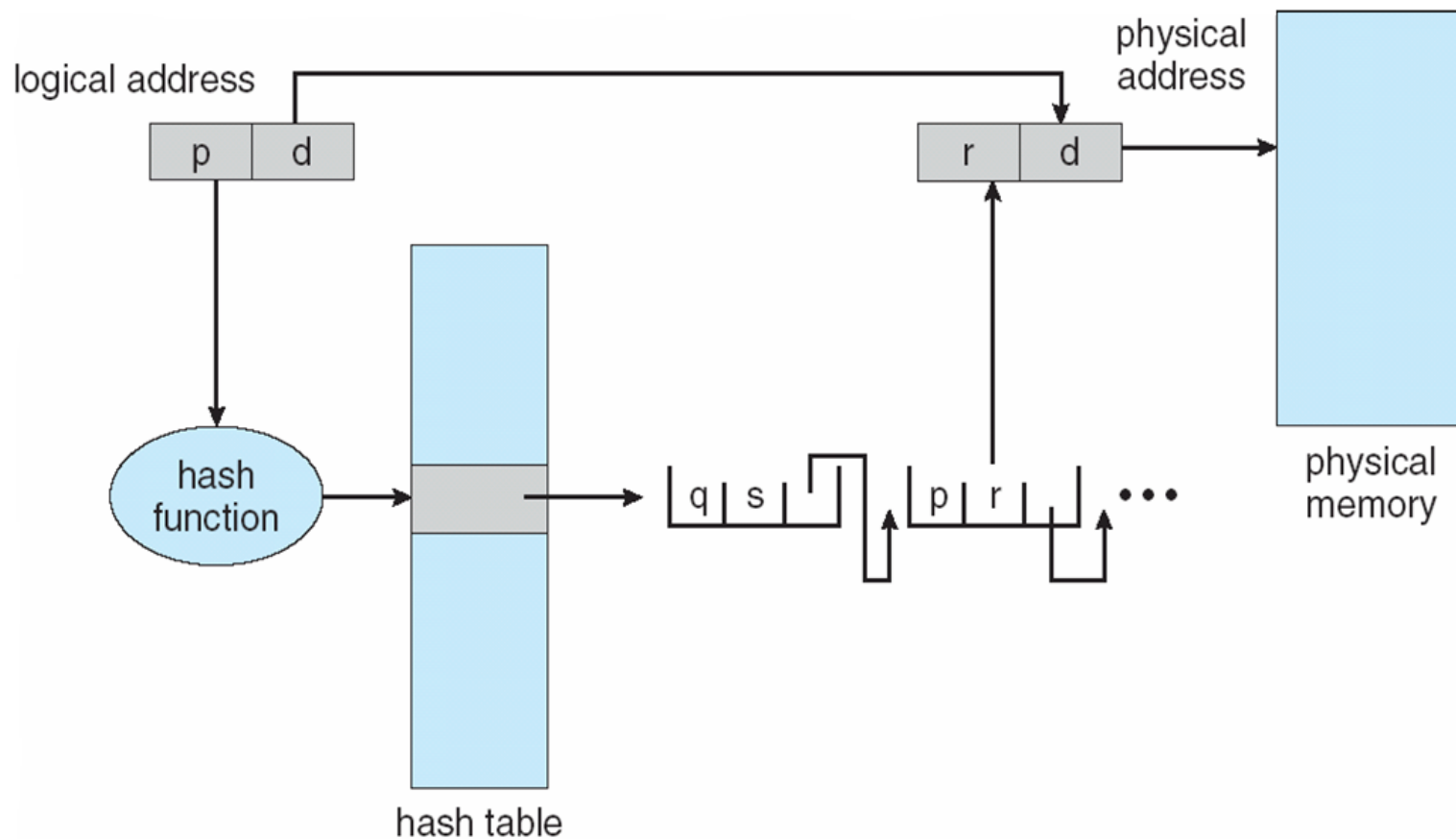
Разделяне на outer table:

2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12

- **Обем на outer table –  $2^{32}$  bytes**
- **За получаване на физически адрес – 4 достъпа до паметта**

# Адресни схеми при 64 bits

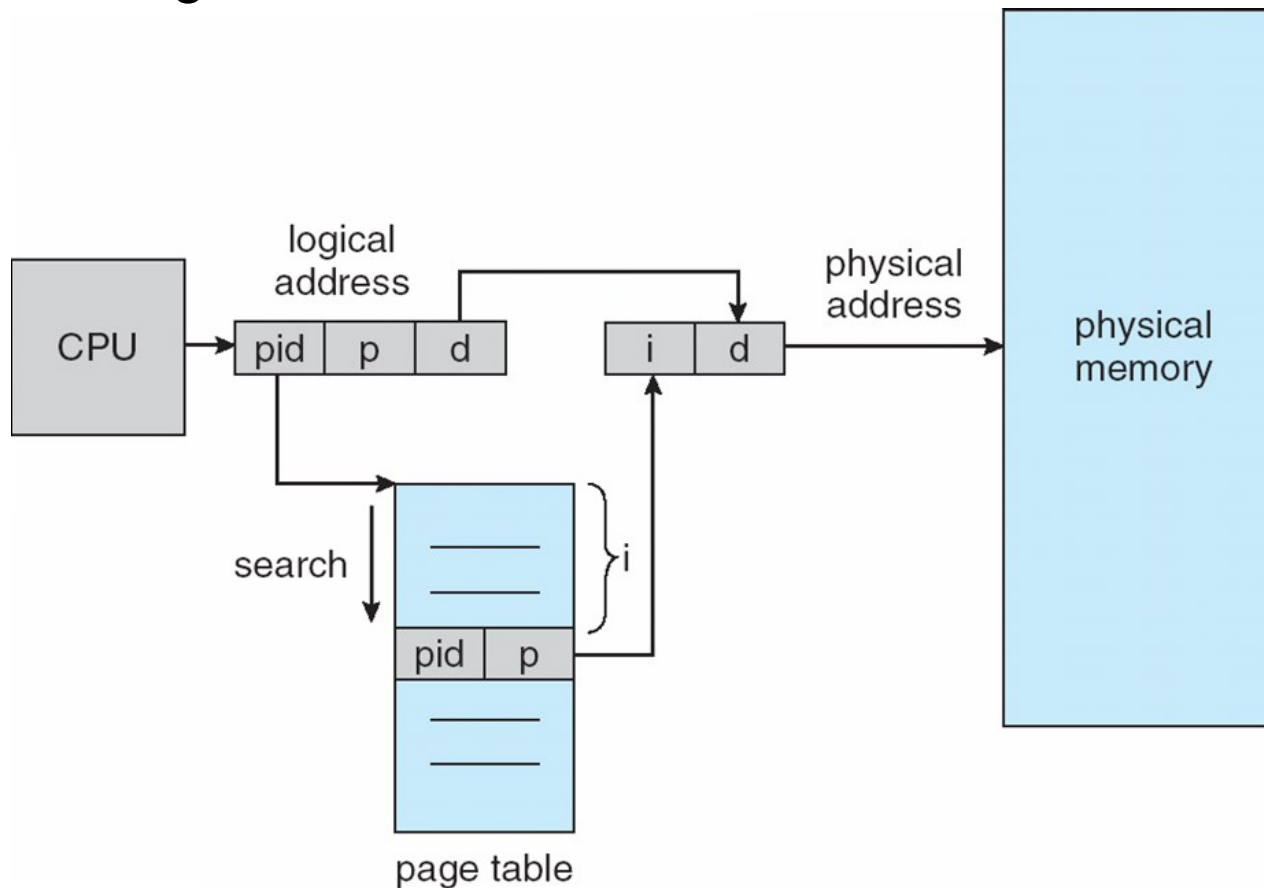
- Hashed Page Table





# Адресни схеми при 64 bits

- Inverted Page Table



# Защита на паметта

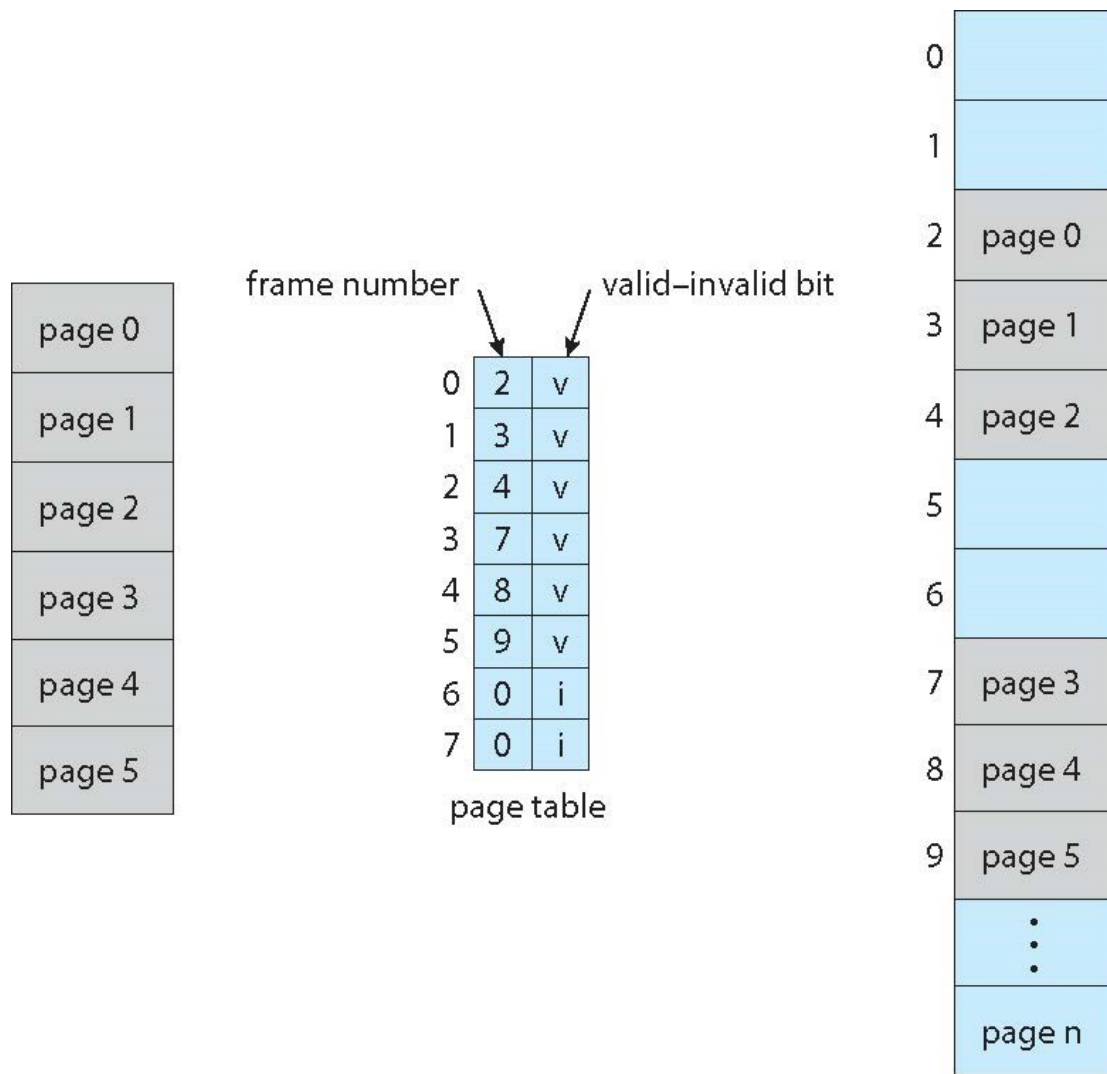
---

Реализира се чрез бит за защита, указващ дали за фрейм е разрешен достъп *read-only* или *read-write*:

- **Valid** – асоциираната страница е в логическото пространство на процеса;
- **Invalid** – страницата не е в логическото пространство на процеса.

В случай на нарушение - прекъсване към ядрото.

# Защита на паметта (2)



# Споделени страници

---

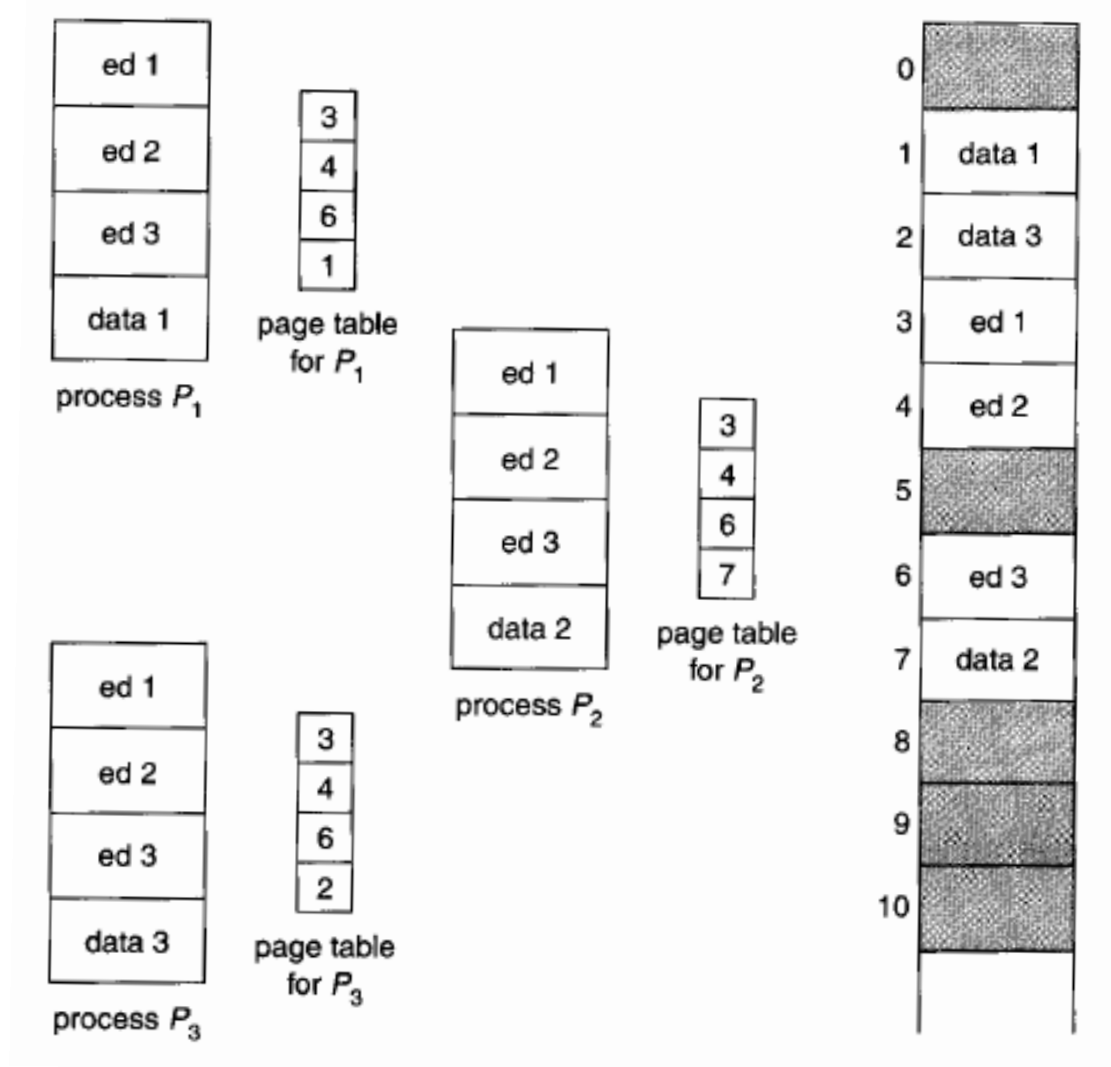
- **Споделен код**

- Едно копие на read-only (*reentrant*) код се споделя между множество процеси (редактори, компилатори и др.);

- **Собствен код и данни**

- Всеки процес съхранява отделно копие на код и данни;
- Страниците за този код и данни могат да са навсякъде в логическото пространство на процеса.

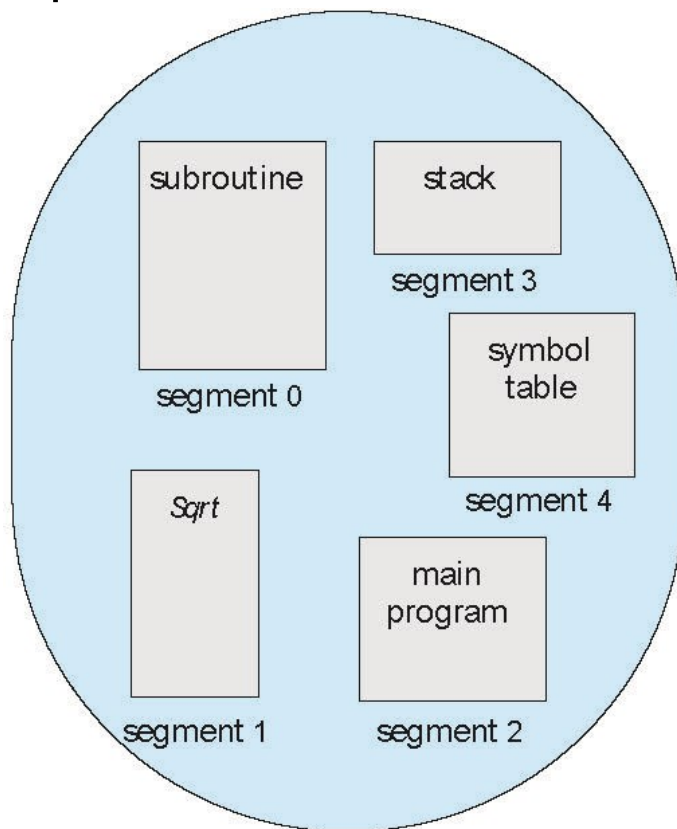
# Споделени страници



# Сегментна организация на ОП

---

- Схема за управление на паметта, поддържаща гледната точка на потребител за паметта.

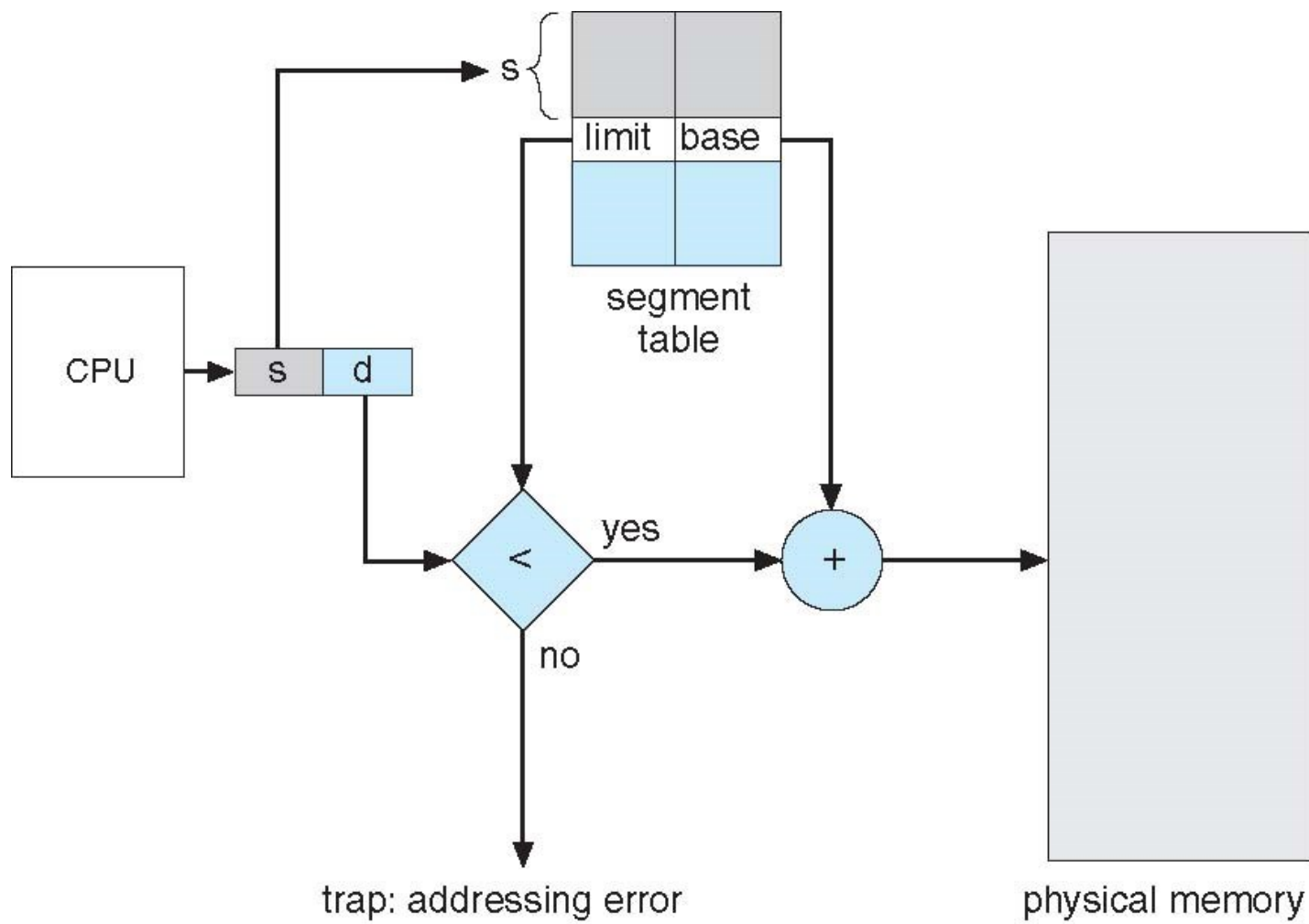


# Сегментна организация на ОП

---

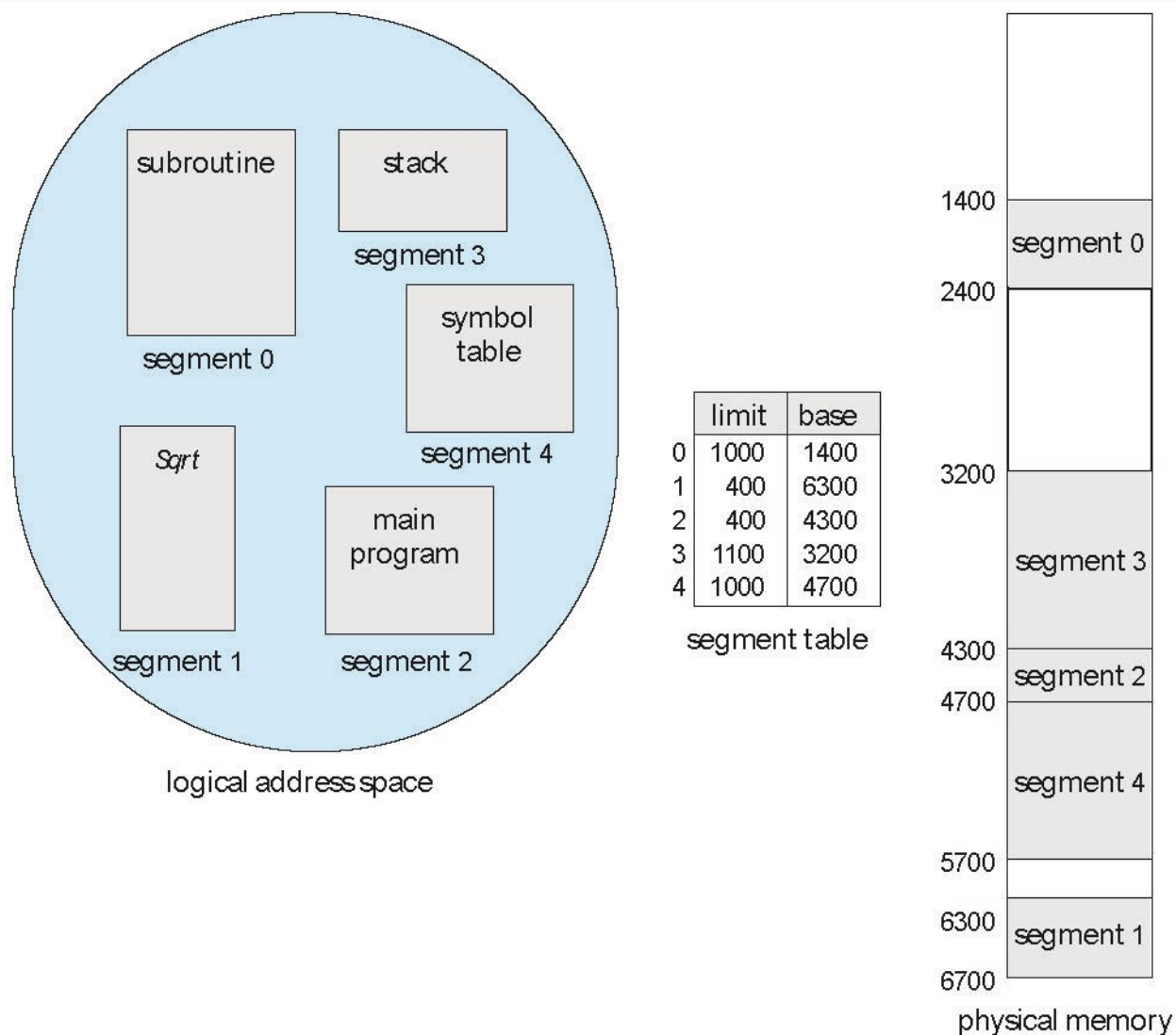
- Логическото адресно пространство се разглежда като съвкупност от сегменти.
- Логическият адрес се представя като двойка  
*<segment\_number, offset>*
- Използва се таблица на сегментите, съдържаща **база-**началния физически адрес на сегмент в паметта и **дължина** на сегмента (лимит).
- Таблицата на сегментите се съхранява в паметта.

# Транслиране





# Пример на сегментиране



# Защита на паметта

---

- Сегменти за инструкции (read-only) и сегменти за данни (read-write).
- Всеки сегмент има бит за защита на достъпа (запис в read-only сегмент).

**Пример:** масив се помещава в сегмент.

# Сегментно-странична организация

---

- Всеки сегмент се състои от страници.
- Логическият адрес се представя като тройка:

*<segment\_number, page\_number, offset>*

- Изисква повече трансляции.

# Пример: IA32

---

- Поддържа сегментна и сегментно-странична организация на паметта.
  - Всеки сегмент до 4Gb.
  - До 16K сегмента за процес.
- Логическото пространство на процеса е разделено в два дяла:
  - 1-ви – съдържа до 8K сегмента собствени за процес (в Local Description Table);
  - 2-ри – съдържа до 8K сегмента, споделени между процеси (в Global DescriptionTable).

# Пример: IA32 (2)

---

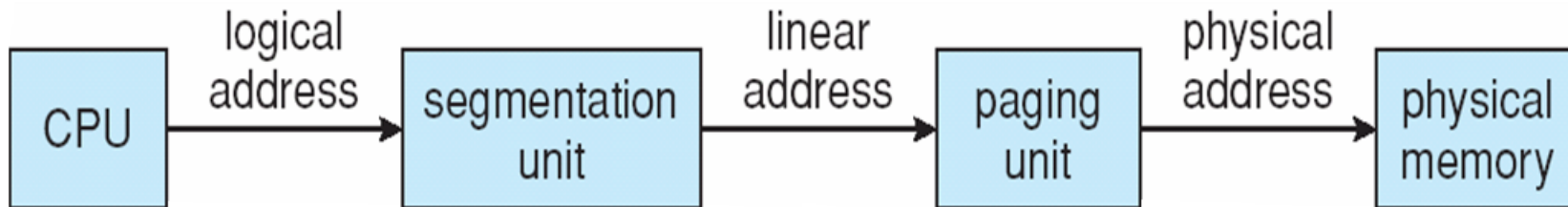
- Логическият адрес с във вида  
**<selector, offset>**
- Селекторът е 16 битов:



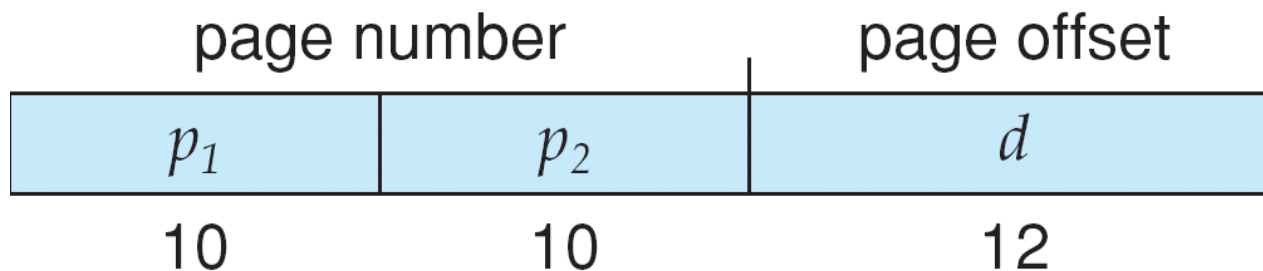
- Отместването е 32 бита.
- На базата на логическия адрес се формира **линеен адрес (32 бита)**.
- На базата на линейния адрес се формира **физически адрес (32 бита)**.

## Пример: IA32 (2)

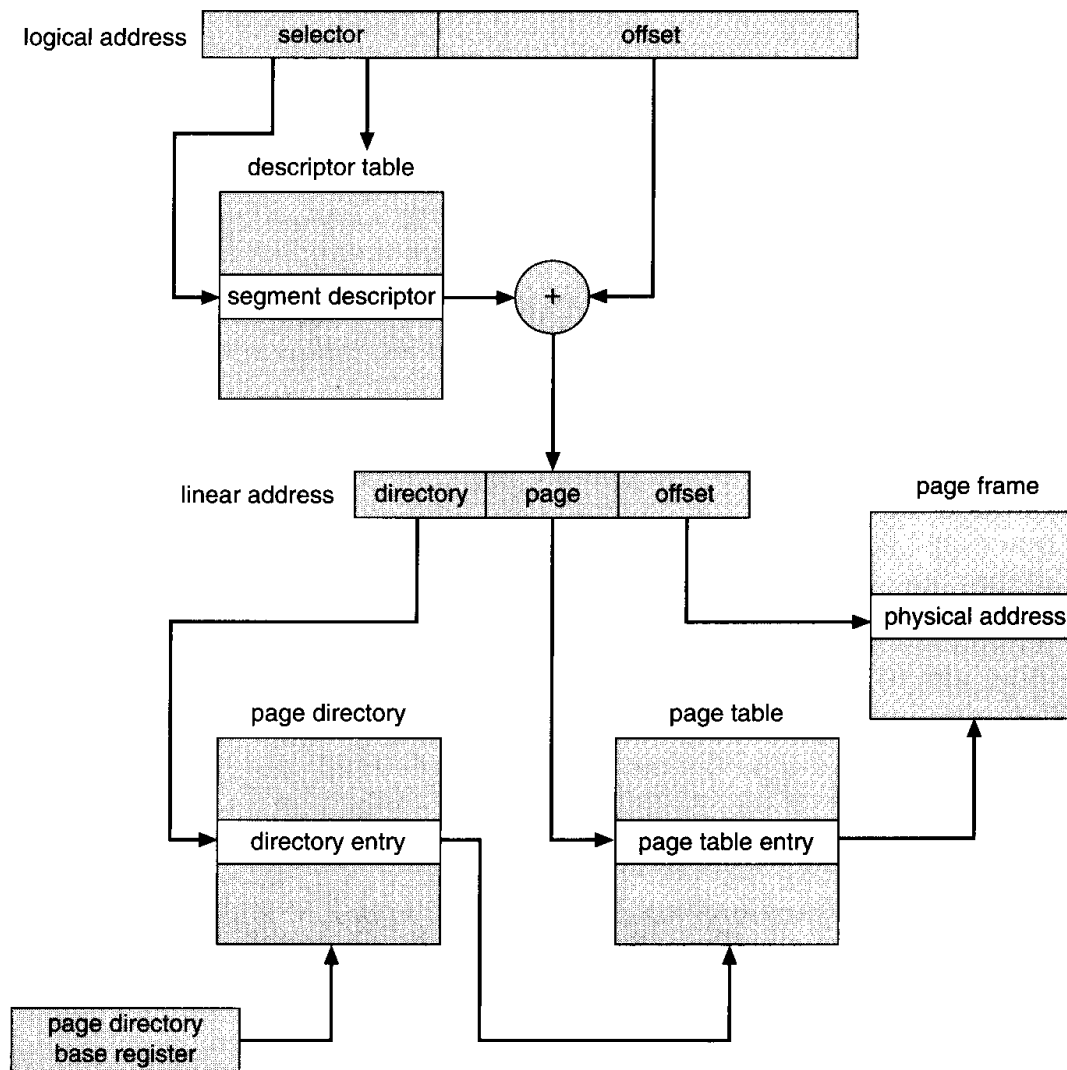
---



Линеен адрес:



# Пример: IA32 (3)



# Виртуална памет

---

- Разделяне на логическата от физическата памет.
- Позволява използване на повече виртуална памет от реално наличната физическа.
- Позволява наличие в паметта само на част от програмата, която се изпълнява.
- Позволява различни процеси да си споделят файлове и памет.
- Реализира се основно на базата на механизма на заявки за страници (*demand paging*).

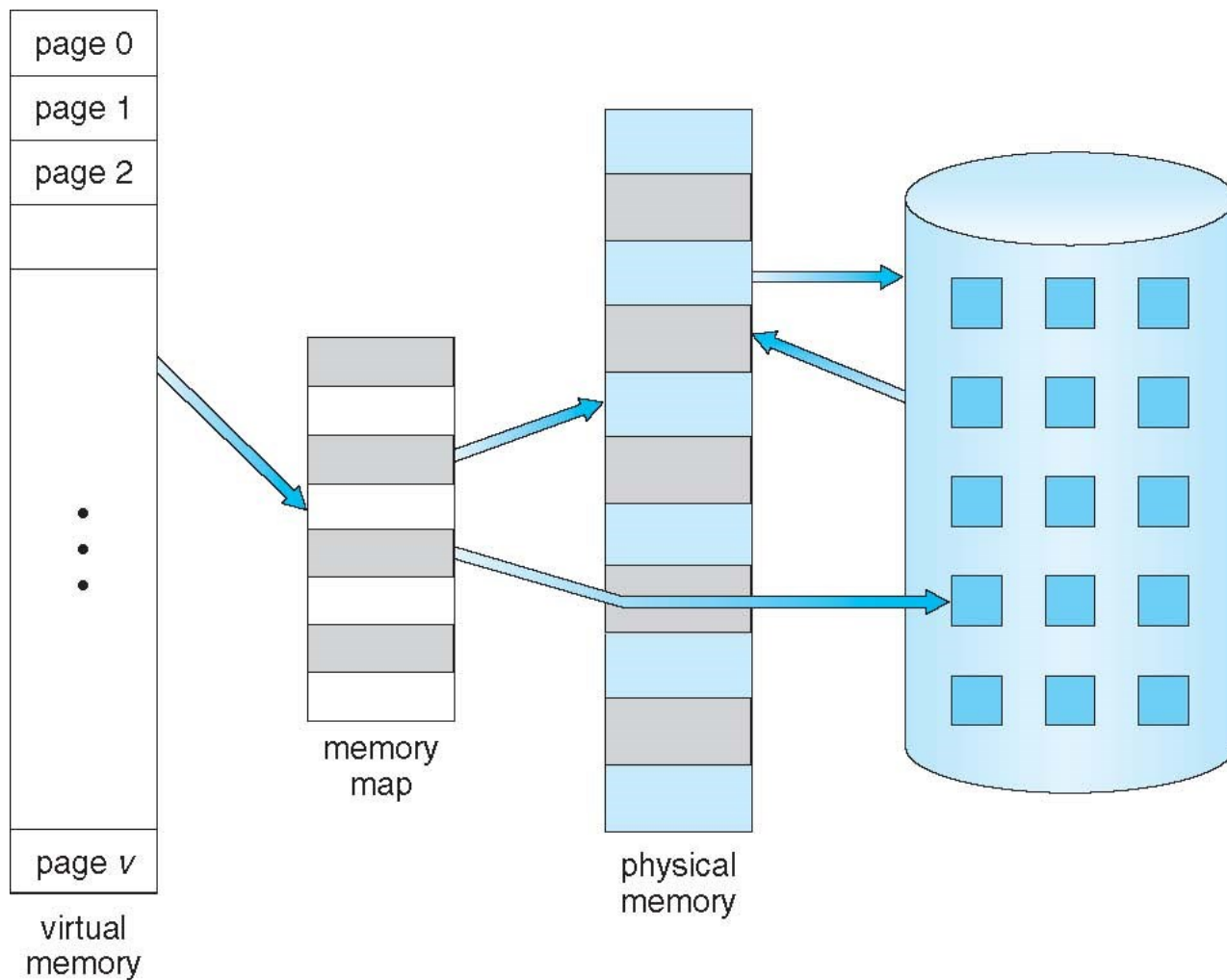


# Прехвърляне на процеси (swapping)

---

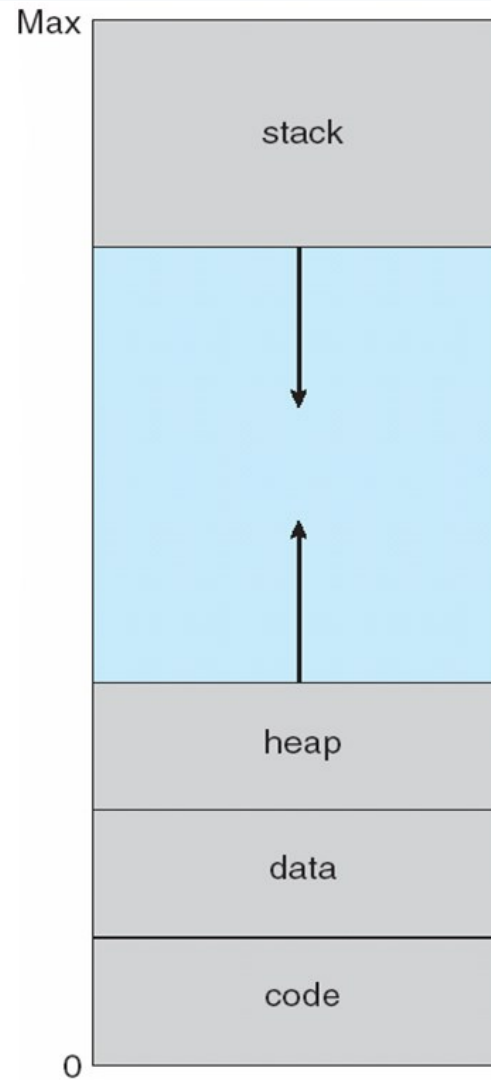
- Съхраняване на паметта за процес на външен носител или зареждането ѝ от диска в ОП.
- Действието се стартира при изчерпване на наличната физическа памет.
- Може да се прехвърля част от процеса.
- Типично е свързано с концепцията за виртуална памет.

# Виртуална памет



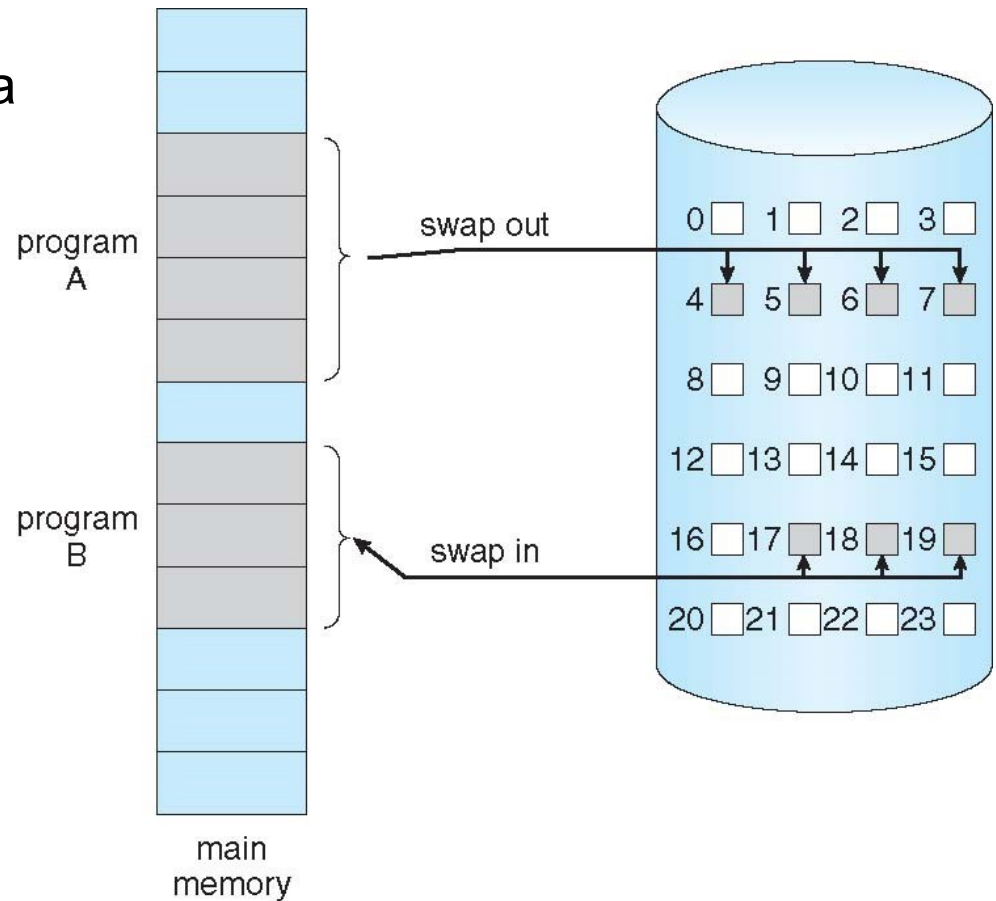
# Виртуално адресно пространство

---



# Заявки за страници

- Зареждане на страница на процес в паметта при необходимост.
- Компонентът, извършващ процеса – ***pager***.



# Заявки за страници

- Ако процесът има достъп до страница, която не е в паметта, Pager трябва да зареди изискваната страница.
- Определянето на това кои страници са в паметта и кои на диска се извършва чрез схема "валидна-невалидна" страница (**v** – в паметта, **i** – на диска).
- Ако по време на транслирането на адрес страницата я няма в паметта – отказ за страница (**page fault**).

Frame #	valid-invalid bit
	<b>v</b>
	<b>v</b>
	<b>v</b>
	<b>i</b>
...	
	<b>i</b>
	<b>i</b>

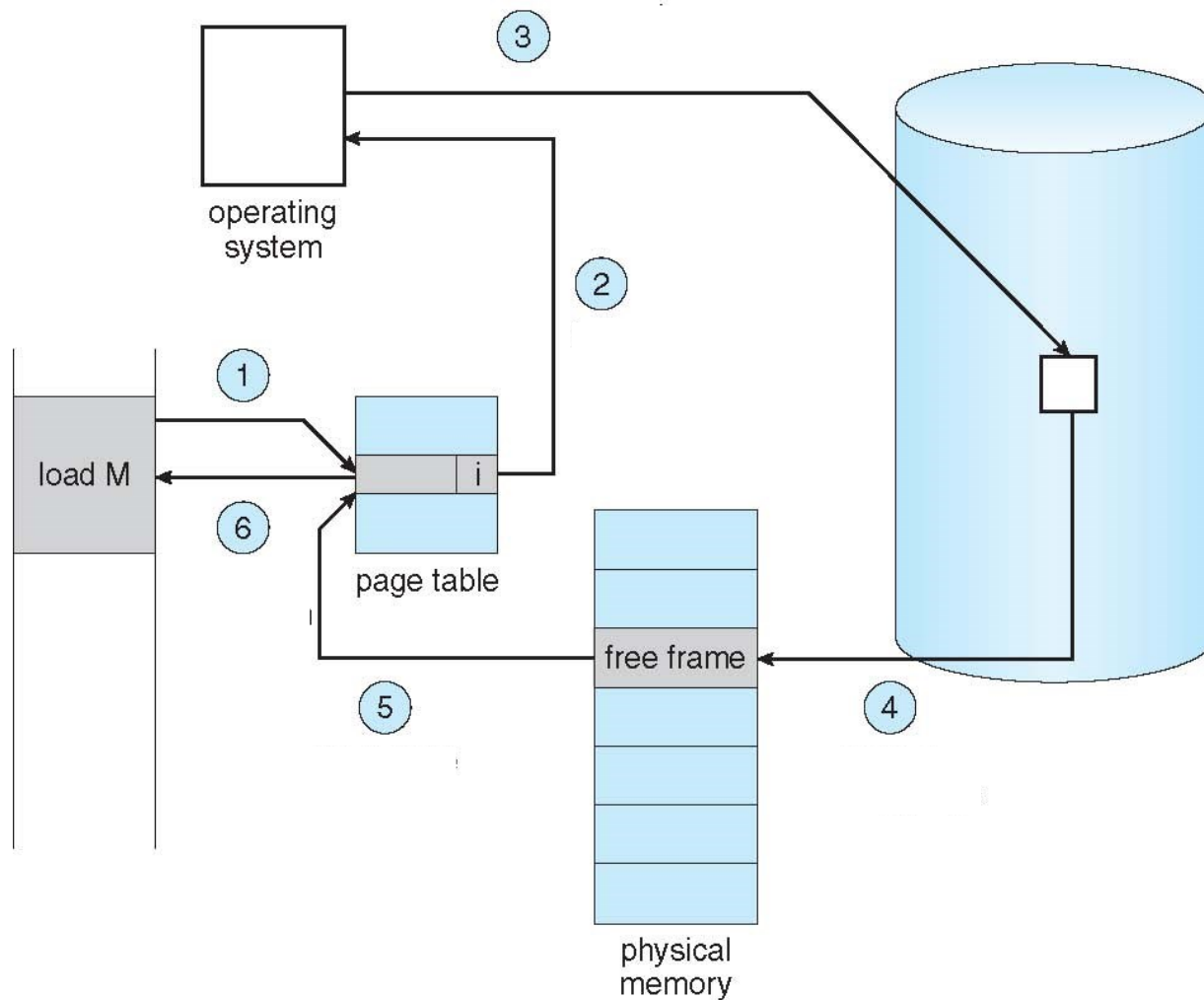
page table

# Отказ за страница

---

➤ Необходимата страница я няма в паметта -> Page Fault.

# Отказ за страница



# Отказ за страница (2)

---

1. ОС преглежда таблицата на страниците
2. Ако е невалидна референция  $\Rightarrow$  abort. Ако е валидна, но не е в паметта - към стъпка (3)
3. Намира свободен фрейм в паметта
4. Зарежда страницата във фрейма чрез планирана I/O операция
5. Модифицира таблиците да указват, че страницата е в паметта (битът за валидиране = **v**)
6. Рестартира инструкцията, която е довела до отказ за страница



# Замяна на страници

---

**Ситуация:** в случай на необходимост от зареждане на нова страница, не са налични свободни фреймове в ОП.

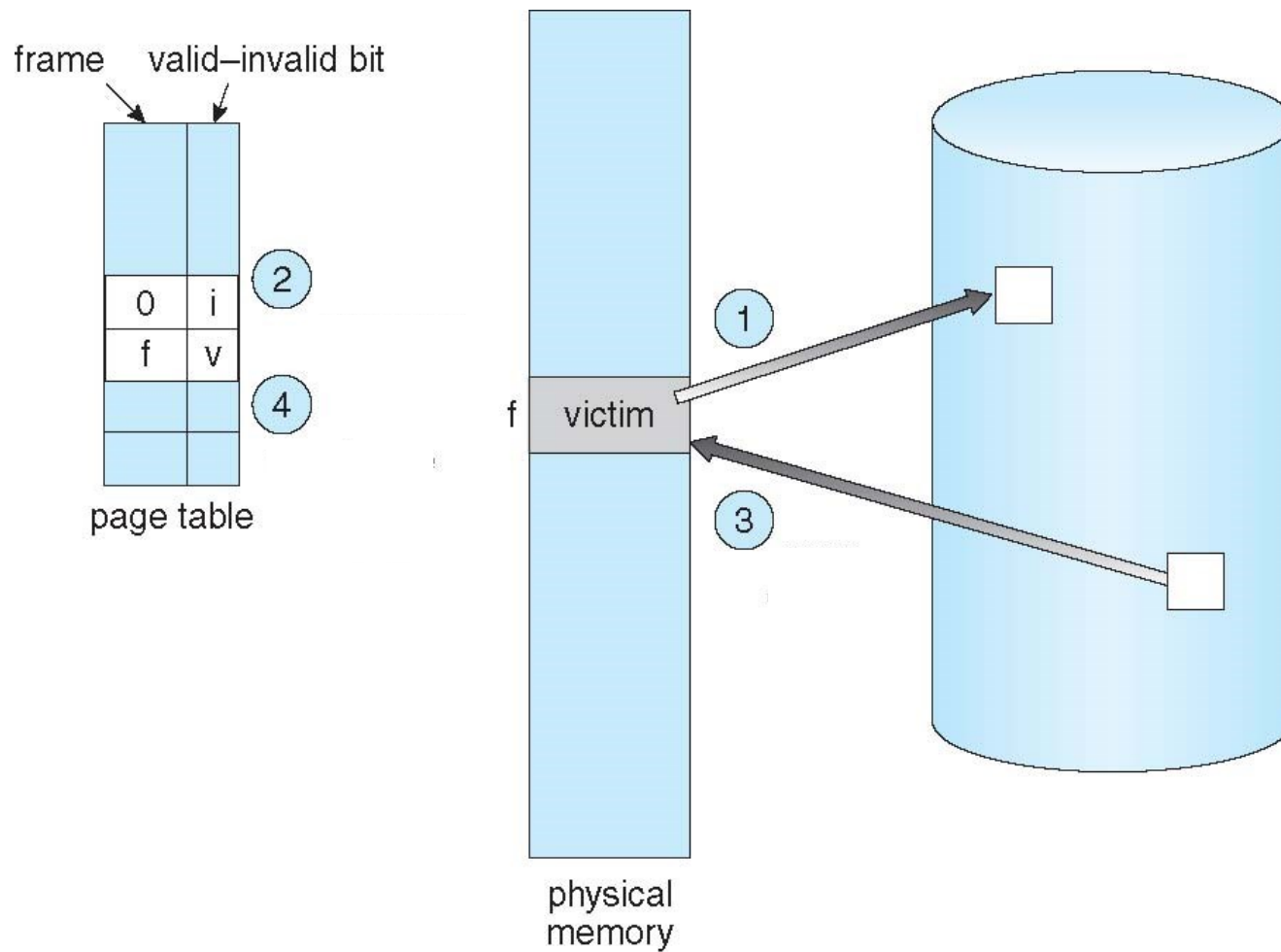
**Решение:** “изхвърляне” на друга страница от ОП и съхраняване на диска.

# Базов алгоритъм

---

1. Намира се позицията на необходимата страница на диска
2. Открива се свободен фрейм:
  - ако е наличен се използва;
  - ако не е се стартира алгоритъм за замяна на страница:
    - избира се определен фрейм за изхвърляне (victim);
    - ако е бил модифициран се записва на диска;
3. Зареждане на необходимата страница във фрейма, обновяване на таблиците на страниците и фреймовете
4. Продължава процеса чрез рестартиране на инструкцията, предизвикала отказ за страница

# Базов алгоритъм



# Стратегии на заместване на страници

---

## **Random стратегия:**

- Избира се случайна страница за изхвърляне.
- Не е добра стратегия – може да се изхвърли страница, която ще се използва в следващия момент

# Стратегии на заместване на страници

---

## **FIFO стратегия:**

- Всяка страница е маркирана с времето на зареждане в паметта.
- При замяна се избира най-отдавна стоялата в паметта.
- Не добра производителност- страницата може да съдържа модул за инициализация (използван преди много време), но да включва променливи, които се използват много често.

# Стратегии на заместване на страници

---

## **Last Recently Used (LRU) стратегия:**

- Базира се на информация в миналото.
- Заменя се страница, не използвана за дълъг период от време.
- За всяка страница се поддържа време на последното използване.
- Изисква специален хардуер.

# Стратегии на заместване на страници

---

## **Least Frequently Used (LFU) стратегия:**

- Брой референции към страница.
- Заменя се страница с най-малък брой референции (активно използваните имат голям брой).

# Стратегии на заместване на страници

---

## **Most Frequently Used (MFU) стратегия:**

- Брой референции към страница.
- Заменя се страница с голям брой референции (предполага се, че страница, която току-що е заредена още не е използвана).



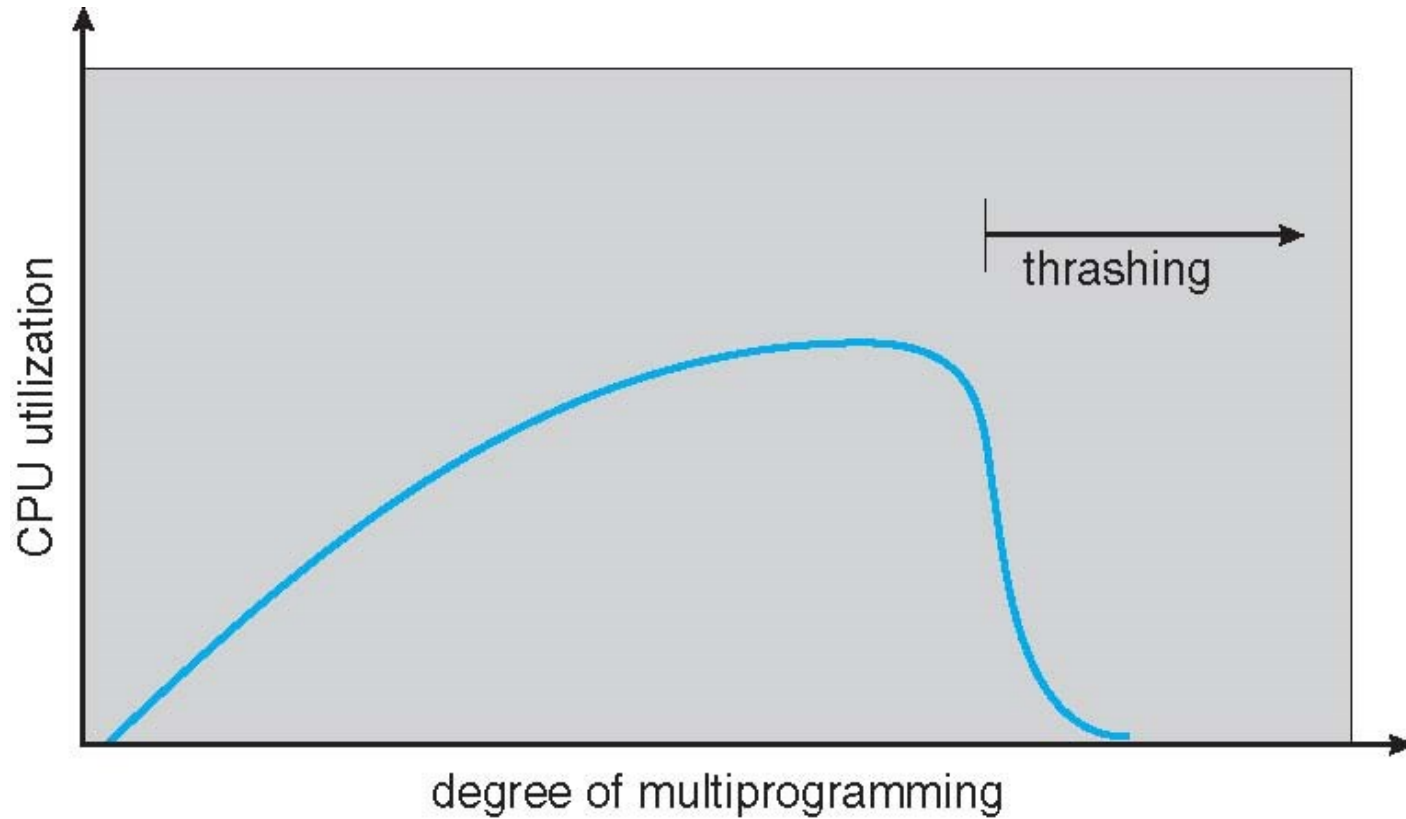
# Проблеми при заместване на страници

---

- Ако процес използва активно всичките си страници и има необходимост от нова страница, трябва да се изхвърли някоя, която отново ще бъде необходима.
- Ако ОС трябва да стартира нов процес.
  - Продължителен процес на прехвърляне на страници между паметта и диска (**trashing**)

# Trashing

---



# Метод на работните множества

---

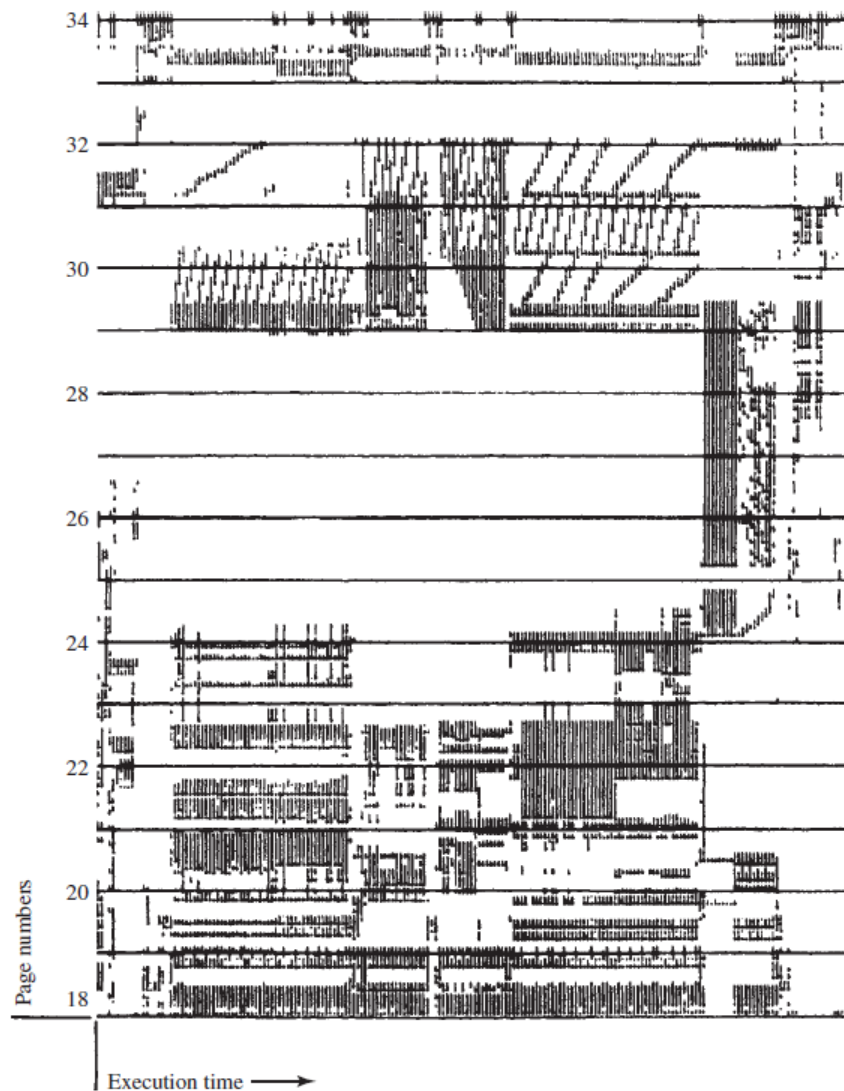
- Модифициран вариант на LRU
- Допускане за *локалност* – достъп на процес до страници, които се използват активно заедно.

За всеки процес в даден момент от време се определя броят на страниците, към които се е обръщал най-скоро за определен период  $\Delta$  – ***работно множество на процеса***.

**$\Delta$ - working set window**

Ако страница не се използва, тя се премахва  $\Delta$  единици виртуално време след последната ѝ референция.

# Локалност

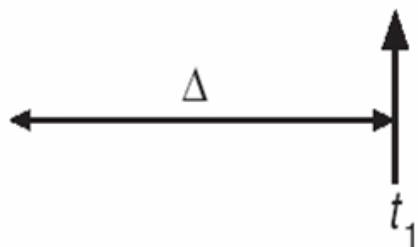


# Метод на работните множества

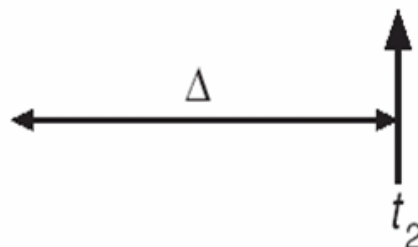
Пример за  $\Delta = 10$

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$

# Метод на работните множества

---

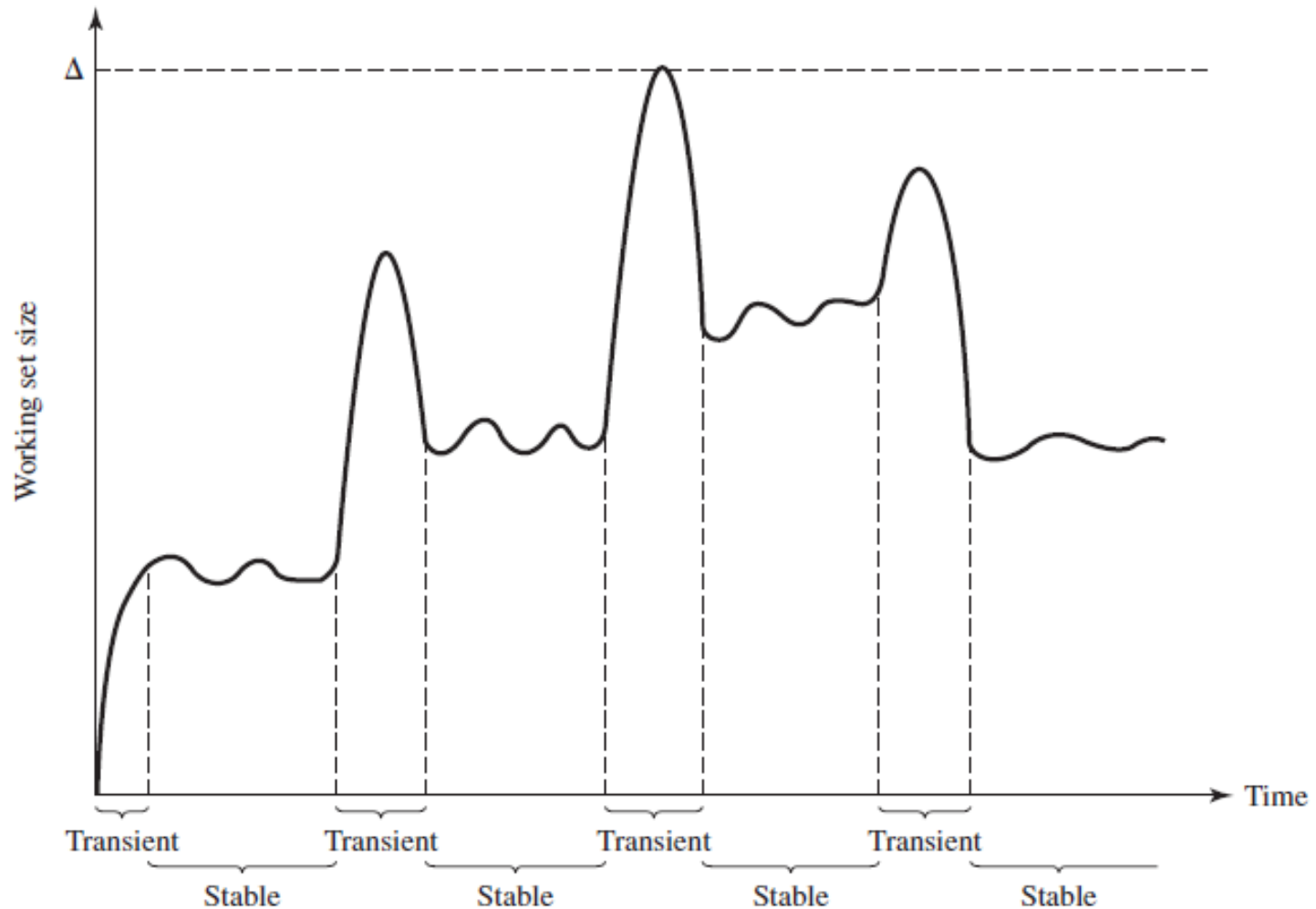
- Коректността на работното множество зависи от  $\Delta$ 
  - ако  $\Delta$  е прекалено малко няма да включи локалността
  - ако  $\Delta$  е прекалено голямо ще включи няколко локалности
  - ако  $\Delta = \infty$  , ще включи цялата програма
- $D = \sum WSS_i$  - приблизително всичките заявки за страници

Ако  $m$  – общи брой налични фреймове.

➤ При  $D > m \Rightarrow \text{Thrashing}$

Политика: Ако  $D > m$ , то спиране или изхвърляне на цял процес

# Метод на работните множества (2)



# Пример за page fault

---

```
int[128,128] data;
```

Всеки ред се съхранява в една страница (128 думи):

data[0,0], data[0,1], ... , data[0,127], data[1,0],..., data[127,127]

Program 1:

```
for (j = 0; j < 128; j++)  
    for (i = 0; i < 128; i++)  
        data[i,j] = 0;
```

Program 2:

```
for (i = 0; i < 128; i++)  
    for (j = 0; j < 128; j++)  
        data[i,j] = 0;
```



# Пример за page fault – 1 вариант

---

<b>data[0,0]</b>
<b>data[0,1]</b>
...
<b>data[0,127]</b>

Page 0

<b>data[1,0]</b>
<b>data[1,1]</b>
...
<b>data[1,127]</b>

Page 1

Program 1:

```
for (j = 0; j < 128; j++)  
  for (i = 0; i < 128; i++)  
    data[i,j] = 0;
```

...

<b>data[127,0]</b>
<b>data[127,1]</b>
...
<b>data[127,127]</b>

Page 127

# Пример за page fault - 1

<b>data[0,0]</b>
data[0,1]
...
data[0,127]

Page 0

**i = 0, j = 0**

data[1,0]
data[1,1]
...
data[1,127]

Page 1

Program 1:

```
for (j = 0; j < 128; j++)  
  for (i = 0; i < 128; i++)  
    data[i,j] = 0;
```

...

data[127,0]
data[127,1]
...
data[127,127]

Page 127

# Пример за page fault - 1

data[0,0]
data[0,1]
...
data[0,127]

Page 0

**i = 1, j = 0**

data[1,0]
data[1,1]
...
data[1,127]

Page 1

Program 1:

```
for (j = 0; j < 128; j++)  
  for (i = 0; i < 128; i++)  
    data[i,j] = 0;
```

...

data[127,0]
data[127,1]
...
data[127,127]

Page 127

# Пример за page fault - 1

data[0,0]
data[0,1]
...
data[0,127]

Page 0

data[1,0]
data[1,1]
...
data[1,127]

Page 1

...

data[127,0]
data[127,1]
...
data[127,127]

Page 127

**i = 127, j = 0**

```
Program 1:  
  for (j = 0; j < 128; j++)  
    for (i = 0; i < 128; i++)  
      data[i,j] = 0;
```

# Пример за page fault - 1

<b>data[0,0]</b>
<b>data[0,1]</b>
...
<b>data[0,127]</b>

Page 0

<b>data[1,0]</b>
<b>data[1,1]</b>
...
<b>data[1,127]</b>

Page 1

Program 1:

```
for (j = 0; j < 128; j++)  
  for (i = 0; i < 128; i++)  
    data[i,j] = 0;
```

...

➤ **128 x 128 = 16 384 откази**

<b>data[127,0]</b>
<b>data[127,1]</b>
...
<b>data[127,127]</b>

Page 127

# Пример за page fault – 2 вариант

---

<b>data[0,0]</b>
<b>data[0,1]</b>
...
<b>data[0,127]</b>

Page 0

<b>data[1,0]</b>
<b>data[1,1]</b>
...
<b>data[1,127]</b>

Page 1

...

<b>data[127,0]</b>
<b>data[127,1]</b>
...
<b>data[127,127]</b>

Page 127

Program 2:

```
for (i = 0; i < 128; i++)  
    for (j = 0; j < 128; j++)  
        data[i,j] = 0;
```

# Пример за page fault – 2

<b>data[0,0]</b>
data[0,1]
...
data[0,127]

Page 0

**j = 0, i = 0**

data[1,0]
data[1,1]
...
data[1,127]

Page 1

Program 2:

```
for (i = 0; i < 128; i++)  
  for (j = 0; j < 128; j++)  
    data[i,j] = 0;
```

...

data[127,0]
data[127,1]
...
data[127,127]

Page 127

# Пример за page fault – 2

data[0,0]
data[0,1]
...
data[0,127]

Page 0

**j = 1, i = 0**

data[1,0]
data[1,1]
...
data[1,127]

Page 1

Program 2:

```
for (i = 0; i < 128; i++)  
  for (j = 0; j < 128; j++)  
    data[i,j] = 0;
```

...

data[127,0]
data[127,1]
...
data[127,127]

Page 127



# Пример за page fault – 2

data[0,0]
data[0,1]
...
data[0,127]

Page 0

**j = 127, i = 0**

data[1,0]
data[1,1]
...
data[1,127]

Page 1

Program 2:

```
for (i = 0; i < 128; i++)  
  for (j = 0; j < 128; j++)  
    data[i,j] = 0;
```

...

data[127,0]
data[127,1]
...
data[127,127]

Page 127

# Пример за page fault – 2

<b>data[0,0]</b>
<b>data[0,1]</b>
...
<b>data[0,127]</b>

Page 0

<b>data[1,0]</b>
<b>data[1,1]</b>
...
<b>data[1,127]</b>

Page 1

...

<b>data[127,0]</b>
<b>data[127,1]</b>
...
<b>data[127,127]</b>

Page 127

Program 2:

```
for (i = 0; i < 128; i++)  
    for (j = 0; j < 128; j++)  
        data[i,j] = 0;
```

➤ **128 откази**

---

# Въпроси ?