Ch08. Memori Virtual

IKI 20250 – Sistem Operasi
Fakultas Ilmu Komputer UI

Revisi: 10 April 2012

A. KONSEP

TUJUAN PEMBELAJARAN

- Memahami manfaat virtual memori
- Memahami bagaimana demand paging bekerja
- Memahami penggunaan *copy-on-write*

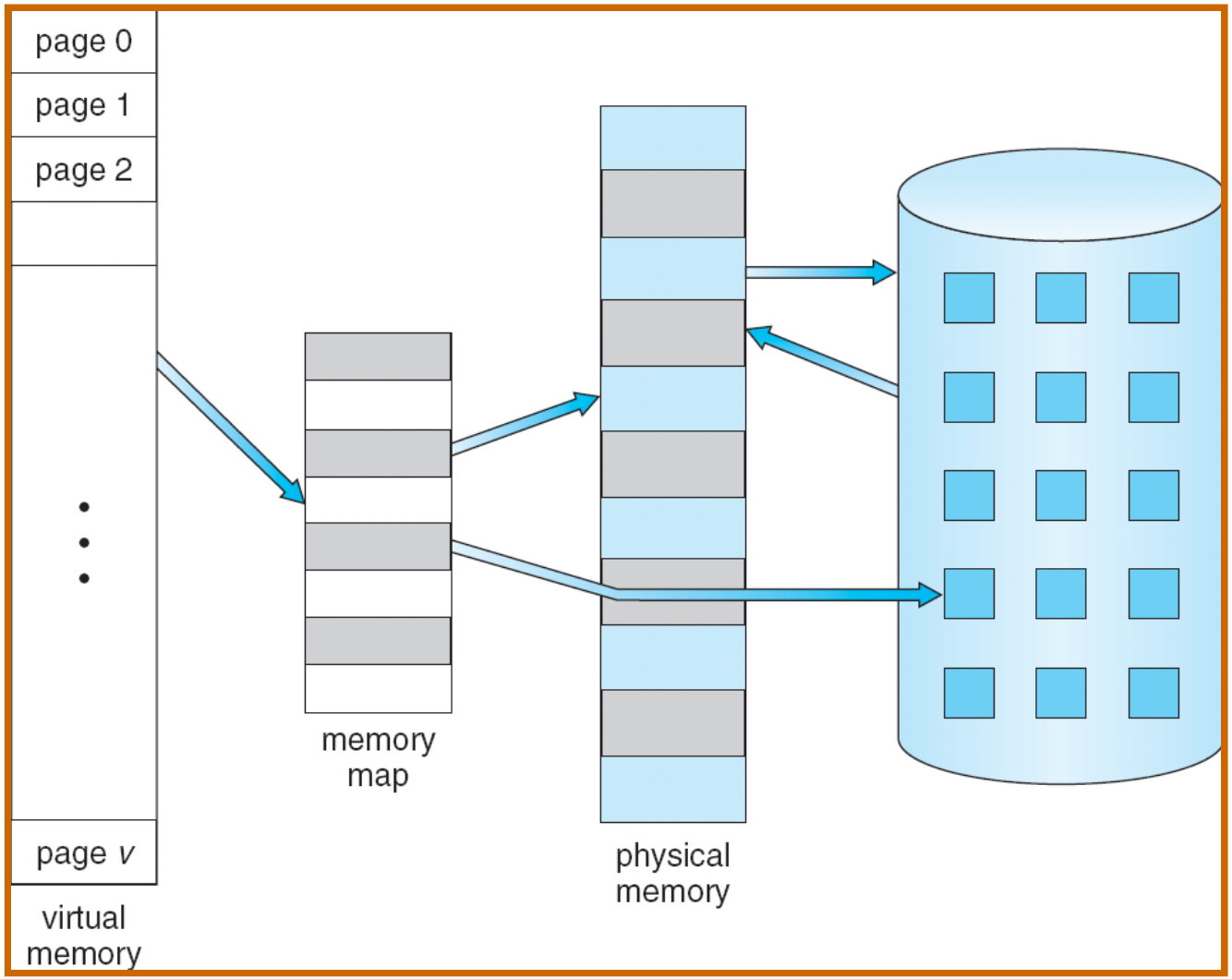
MEMORI VIRTUAL

- Tidak semua bagian proses dieksekusi di memori
- Umumnya proses dieksekusi secara parsial dimemori
- Sebagian proses disimpan pada memori sekunder ditempatkan pada lokasi yang disebut **memori virtual**

MANFAAT MEMORI VIRTUAL

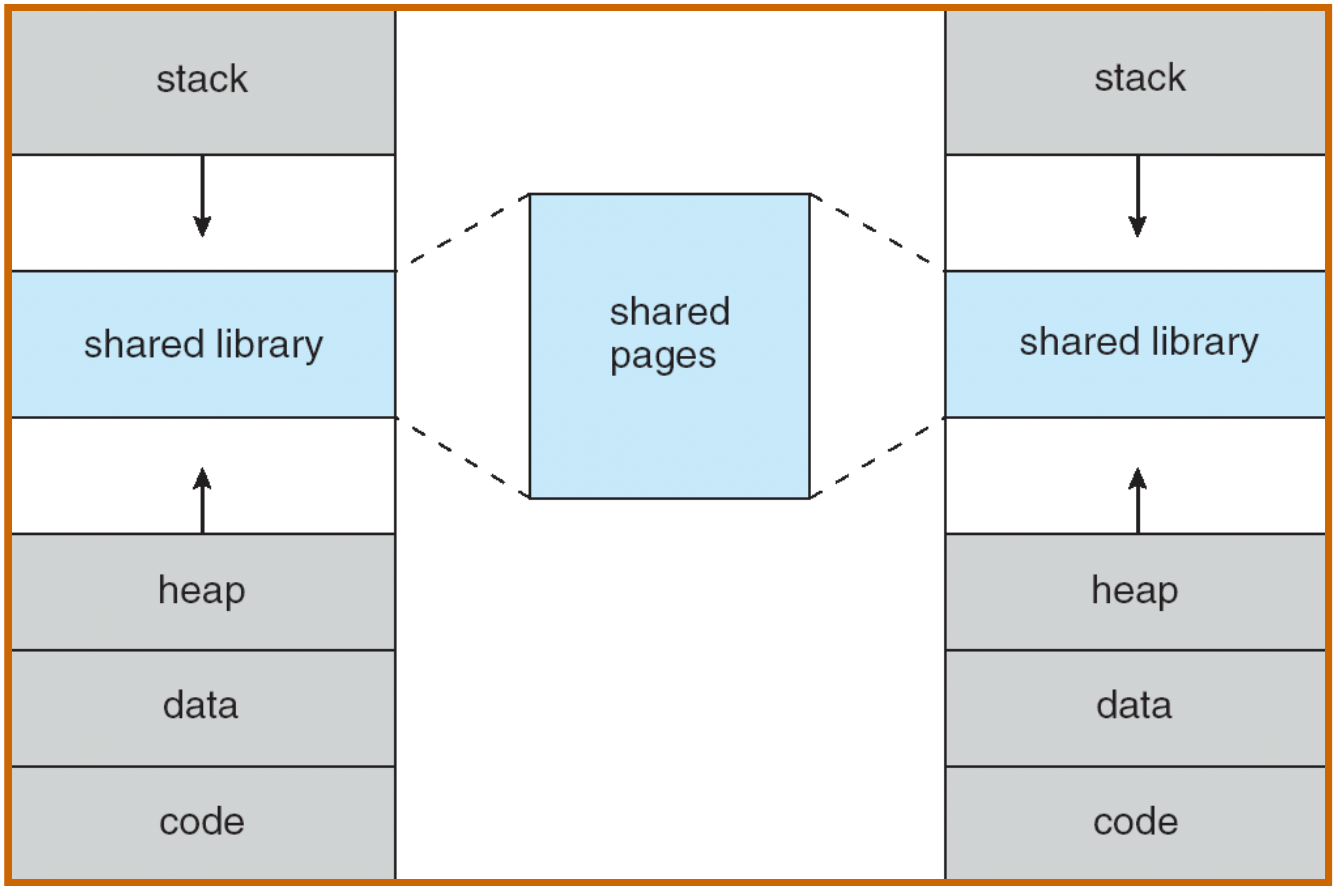
- Programming menjadi lebih mudah
- Tidak ada batasan memori
- Meningkatkan derajat multiprogramming
- Frekuensi swapping page lebih sedikit

MEMORI VIRTUAL



Memori virtual lebih besar dibanding memori fisik

SHARED LIBRARY

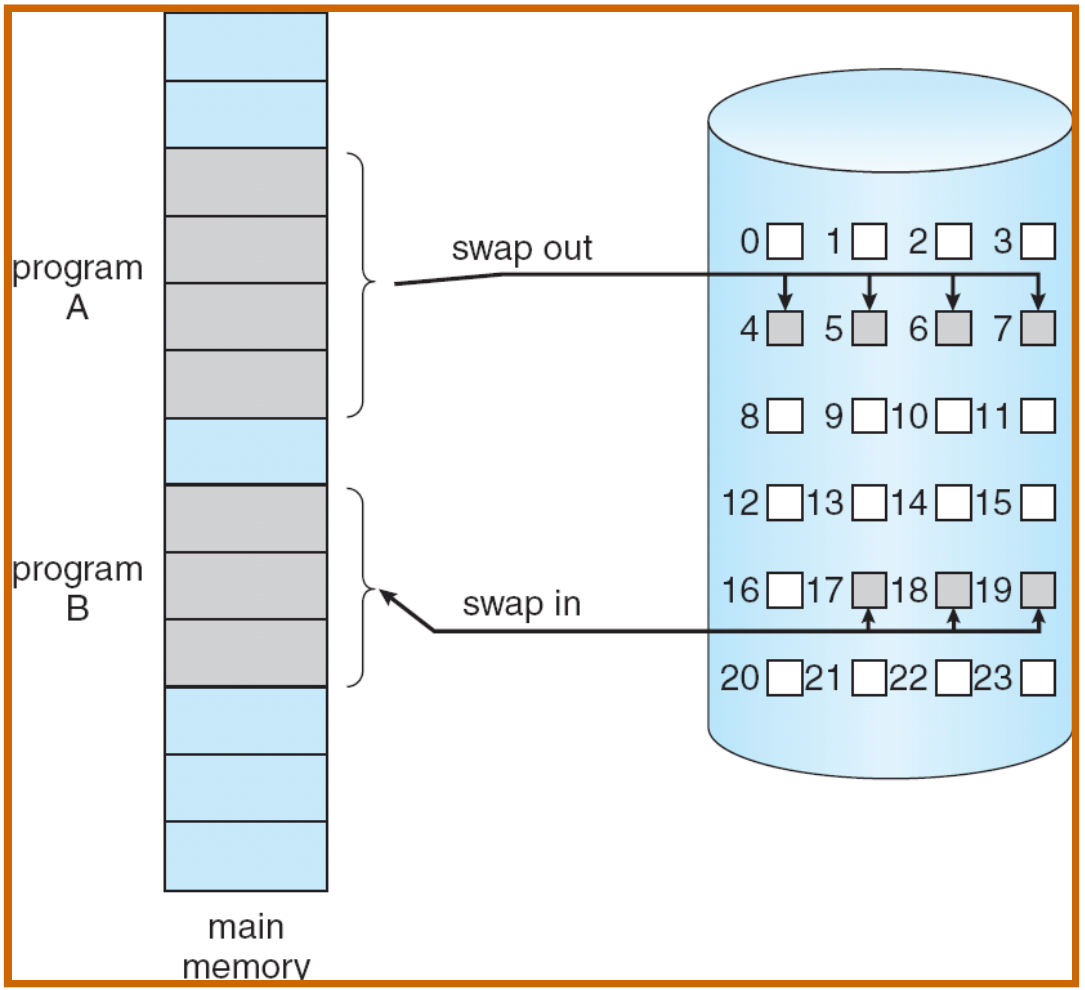


Shared library dengan menggunakan memori virtual

DEMAND PAGING

- Page dimasukkan ke memory hanya ketika dibutuhkan saat waktu eksekusi.
- Keuntungan :
  - Aktifitas I/O lebih sedikit
  - Ruang memori yang dibutuhkan lebih sedikit
  - Respon menjadi lebih cepat
  - Lebih banyak proses user dalam memori

DEMAND PAGING



Proses swapping program ke dalam memori

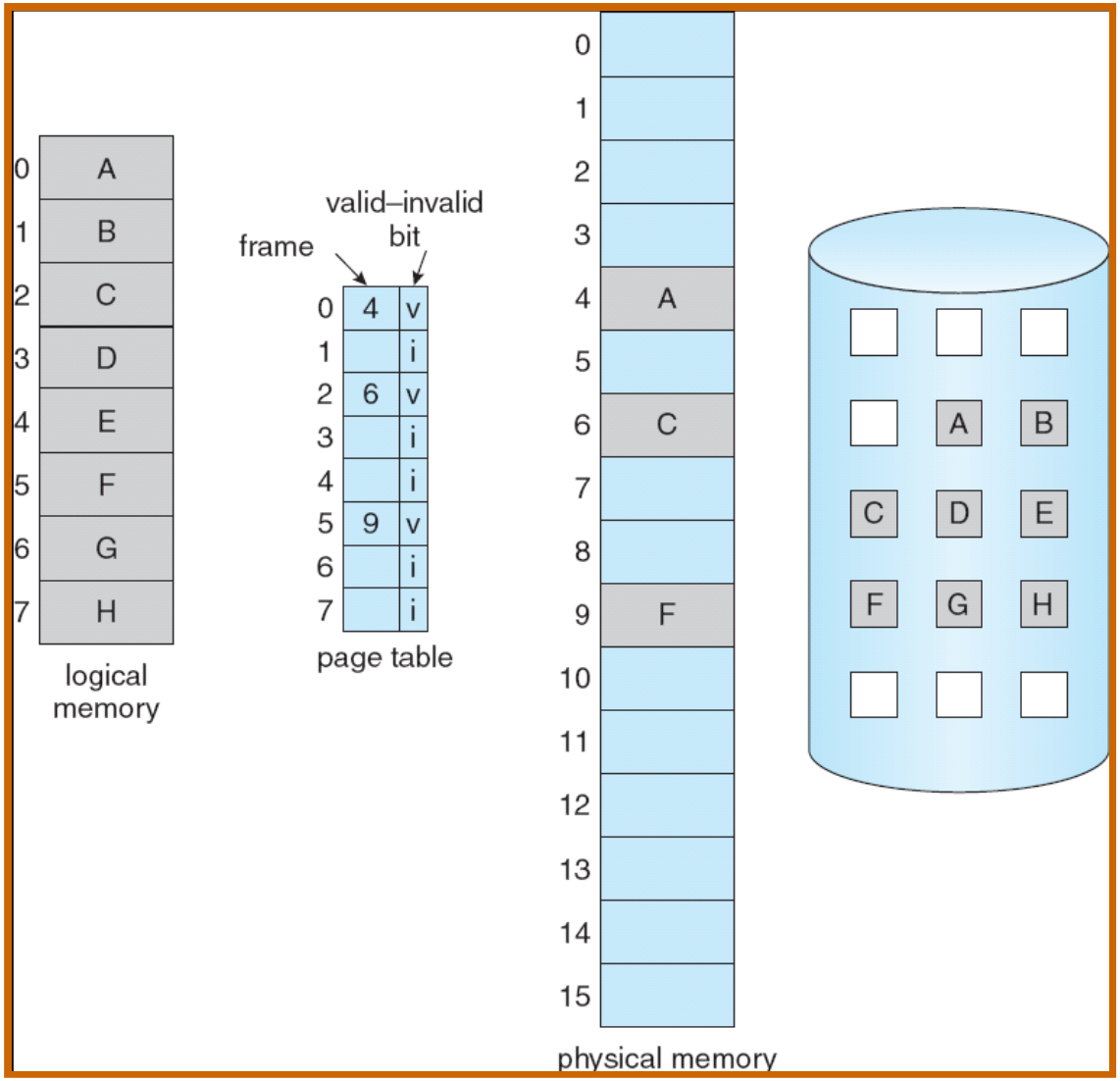
VALID-INVALID BIT

- Cara untuk mengetahui apakah sebuah page ada dimemori atau tidak, gunakan Valid dan Invalid Bit

- **v** (valid) → page ada di memori fisik
- **i** (invalid) → punya dua arti: alamat logika tidak dalam range alamat proses atau page ada tidak ada di memori fisik

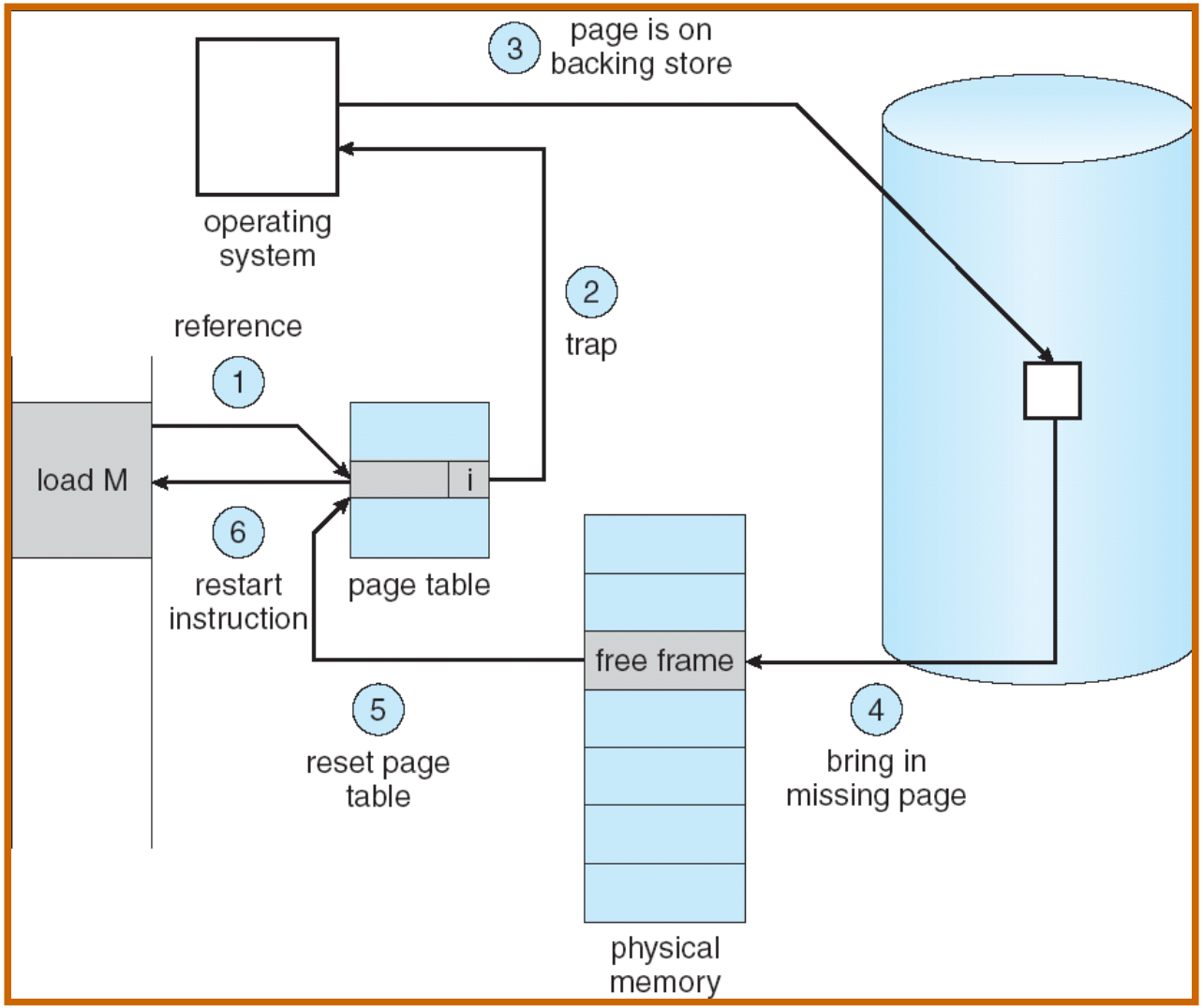
- Jika entri page tabel mempunyai bit **i,** maka terjadi *page fault-trap*, sistem operasi mengambil page dari disk ke memory

VALID-INVALID BIT



Beberapa page berada dalam disk

PAGE FAULT



Alur penanganan page fault

DEMAND PAGING

- *Pure demand paging* → eksekusi proses tanpa ada satu pun page dimemori
  - page di pindah ke memori hanya ketika dibutuhkan
- Hardware untuk demand paging:
  - Page Table (memory)
  - Secondary Memory (disk)

KINERJA DEMAND PAGING

- Probabilitas Page Fault dinotasikan dengan p, dimana p berada dalam range $0 \leq p \leq 1.0$
  - jika $p = 0$, tidak terjadi page fault
  - Jika $p = 1$, page fault untuk semua akses

- Effective Access Time (EAT)

  EAT =((1 – *p*) x memory access) + (*p* x *page fault service time*)

  page fault service time ➔ Waktu yang dibutuhkan untuk menangani page fault
- Jika p=0, Effective Access Time ≈ Memori Access Time

EAT DEMAND PAGING

- Waktu akses memory = 200 nanosecond
- Rata-rata waktu *page-fault service time* = 8 milliseconds (1 ms=10^6 ns)
- EAT = ((1 – p) x 200) + (p x (8 ms))

  = ((1 – p) x 200) + (p x 8,000,000)

  = 200 + (p x 7,999,800)
- Jika 1 dari 1.000 kali akses terjadi fault, maka EAT = 8.2 microseconds.
- EAT bertambah menjadi 40 kali lipat dari waktu akses memori !

EAT DEMAND PAGING

- Jika ingin EAT tidak lebih dari 220ns maka :

  $220 > 200+7.999.800 \times p$

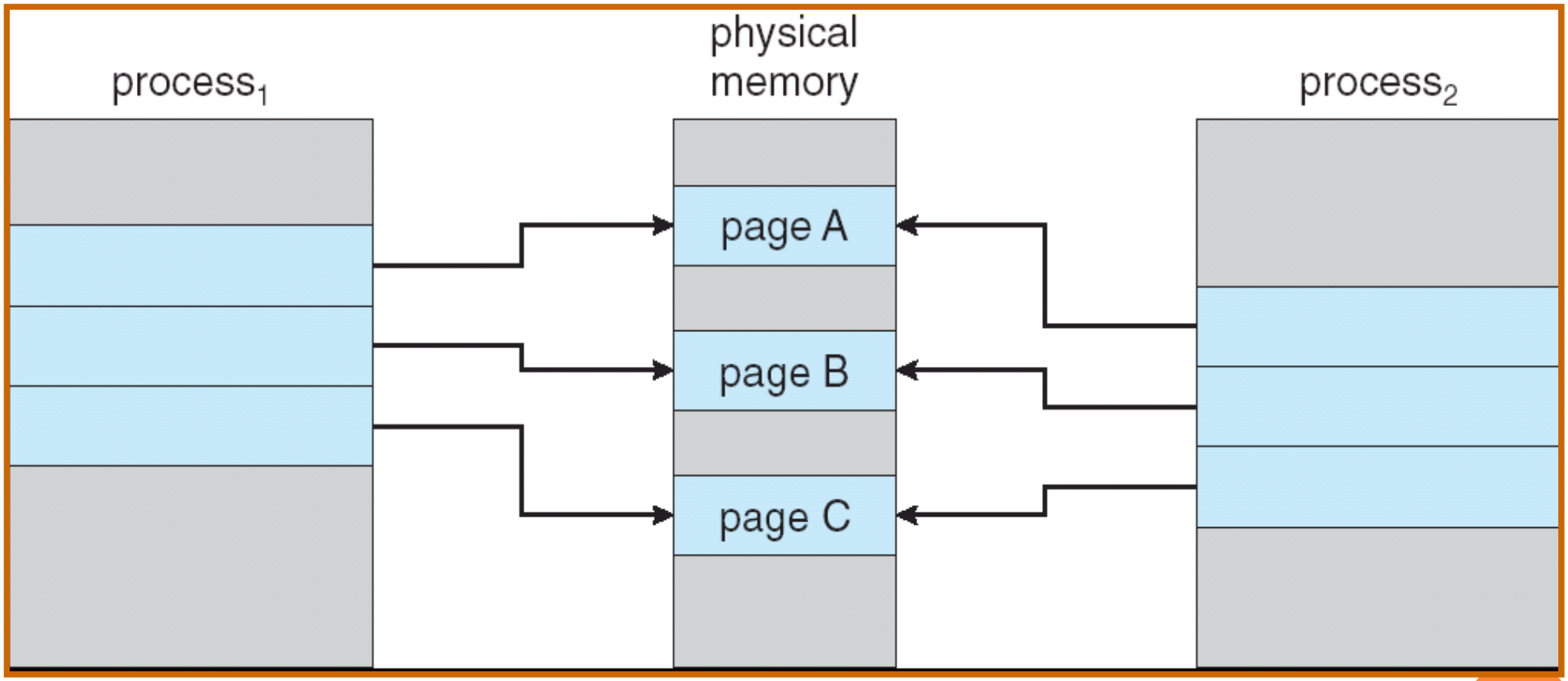
  $20 > 7.999.800 \times p$

  $p < 0{,}0000025$, artinya p harus lebih kecil dari kejadian *page-fault* sekali dalam 400.000 kali akses

COPY-ON-WRITE

- Proses c*hild* dan *parent* menggunakan page yang sama
- Jika *shared-pages* diubah oleh *parent* atau *child*, maka *shared-pages* tersebut diduplikasi ke pihak pengubah (*child* atau *parent*)
- Versi lain : vfork()
  - Proses parent di hentikan sementara
  - Proses *child* dapat merubah page *parent*, tanpa harus menduplikasi page
  - Perubahan dapat terlihat oleh *parent* dan *child*

COPY-ON-WRITE



Sebelum proses 1 melakukan modifikasi page c

COPY-ON-WRITE

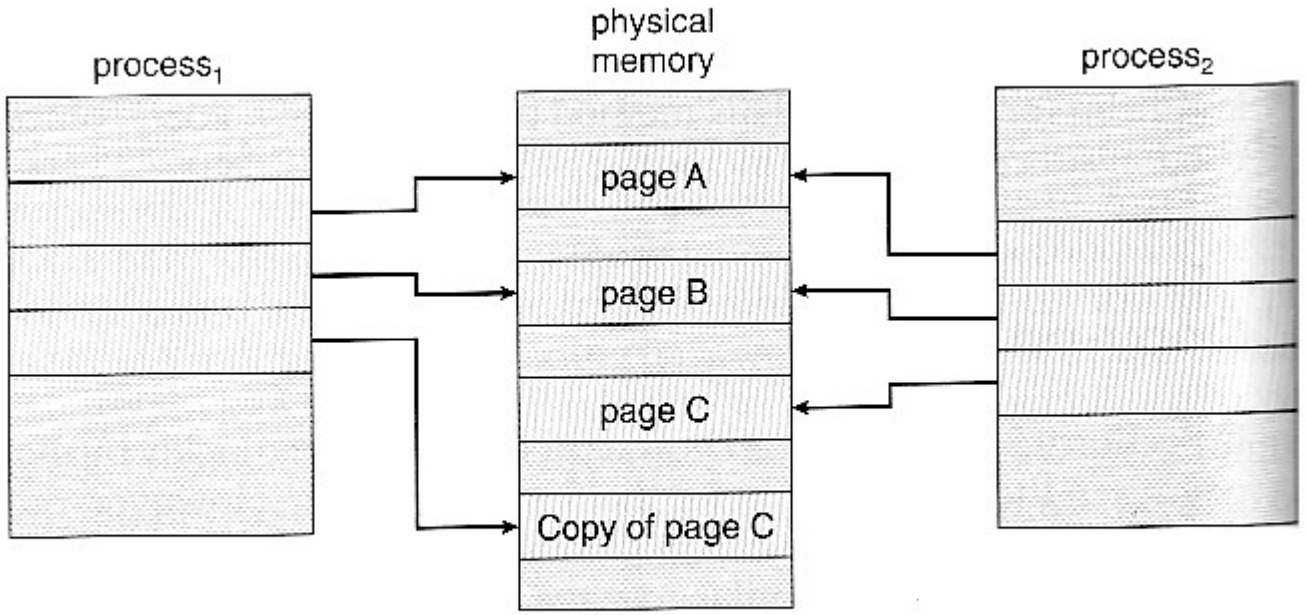


Figure 9.8 After process 1 modifies page C.

Setelah proses 1 melakukan modifikasi page c

B. ALGORITMA PERGANTIAN PAGE

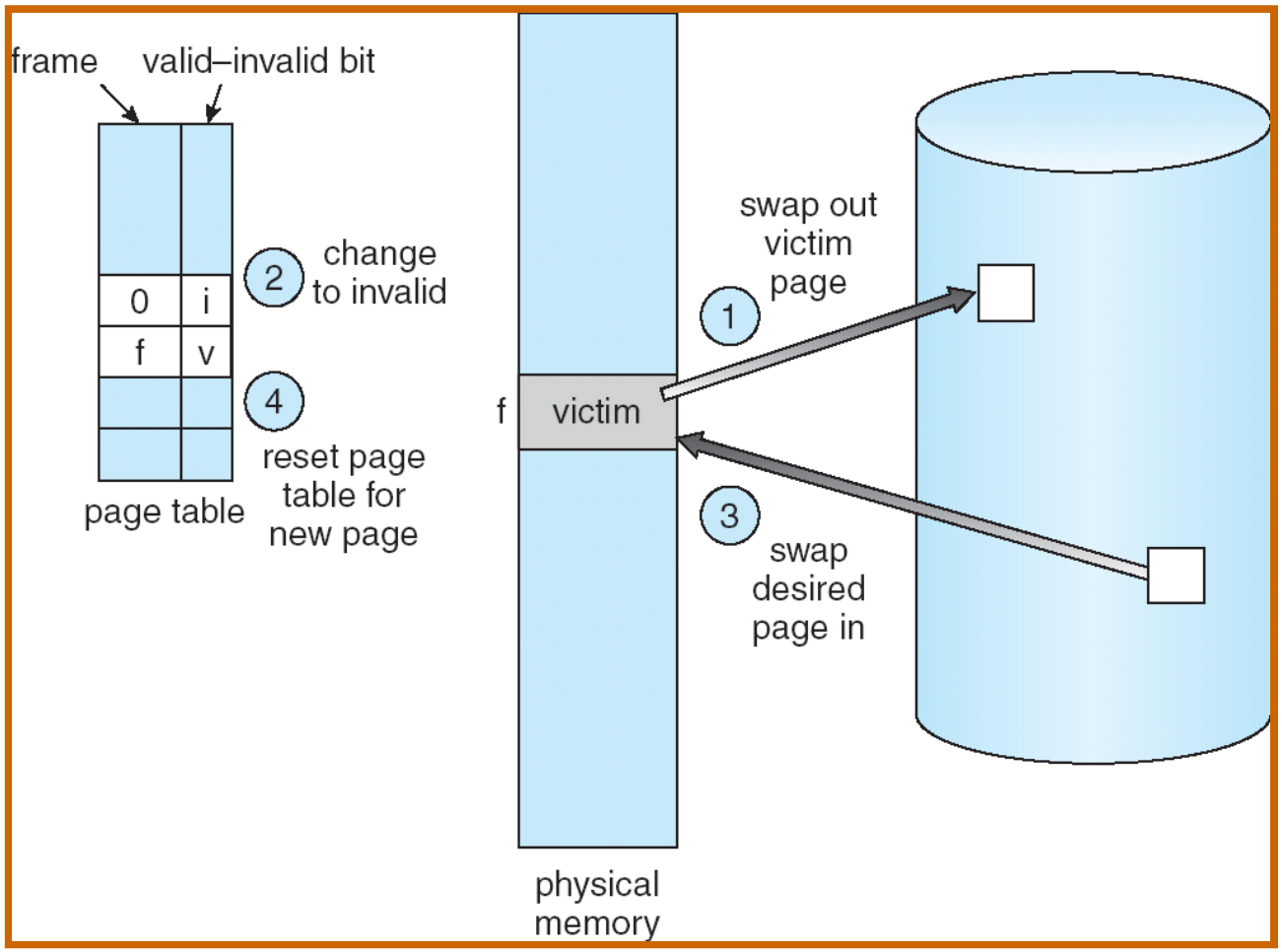
TUJUAN PEMBELAJARAN

- Memahami konsep pergantian page
- Memahami algoritma pergantian page
  - FIFO, Optimal, Least Recently Used (LRU), Least Recently Used (LRU) Approximation, Second-chance, Circular Queue (Algoritma Clock), Enhanced Second-Chance, Counting-Based : LFU & MFU

PERGANTIAN PAGE (*PAGE REPLACEMENT*)

- Terjadi jika page akan dimasukkan ke memori untuk dieksekusi namun tidak ada frame kosong yang tersedia
- Pergantian page – cari 'korban' page dalam frame memory yang sedang tidak digunakan, kemudian lakukan page out
  - Page A pada frame X dipindahkan ke harddisk
  - Update entri page A pada page table menjadi invalid
  - Gunakan slot frame X untuk page baru (Y) yang akan dieksekusi
  - Update entri page Y pada page table menjadi valid

PERGANTIAN PAGE (*PAGE REPLACEMENT*)



alur pergantian page (*page replacement*)

PERGANTIAN PAGE (*PAGE REPLACEMENT*)

- Langkah diatas menggunakan dua kali page transfer (swap out & swap in)
- Agar satu kali transfer? Tambah bit untuk *modify bit* pada entri page table!
- *Modify bit* (*dirty bit*)
  - Jika bit diset, maka page pernah diubah
    - page di memori != page didisk
    - page harus dipindah ke disk
  - Jika bit tidak diset, maka page belum pernah diubah
    - page dimemori ≈ page didisk
    - page tidak dipindah ke disk, langsung dihapus!
    - Contoh : shared code, binary code

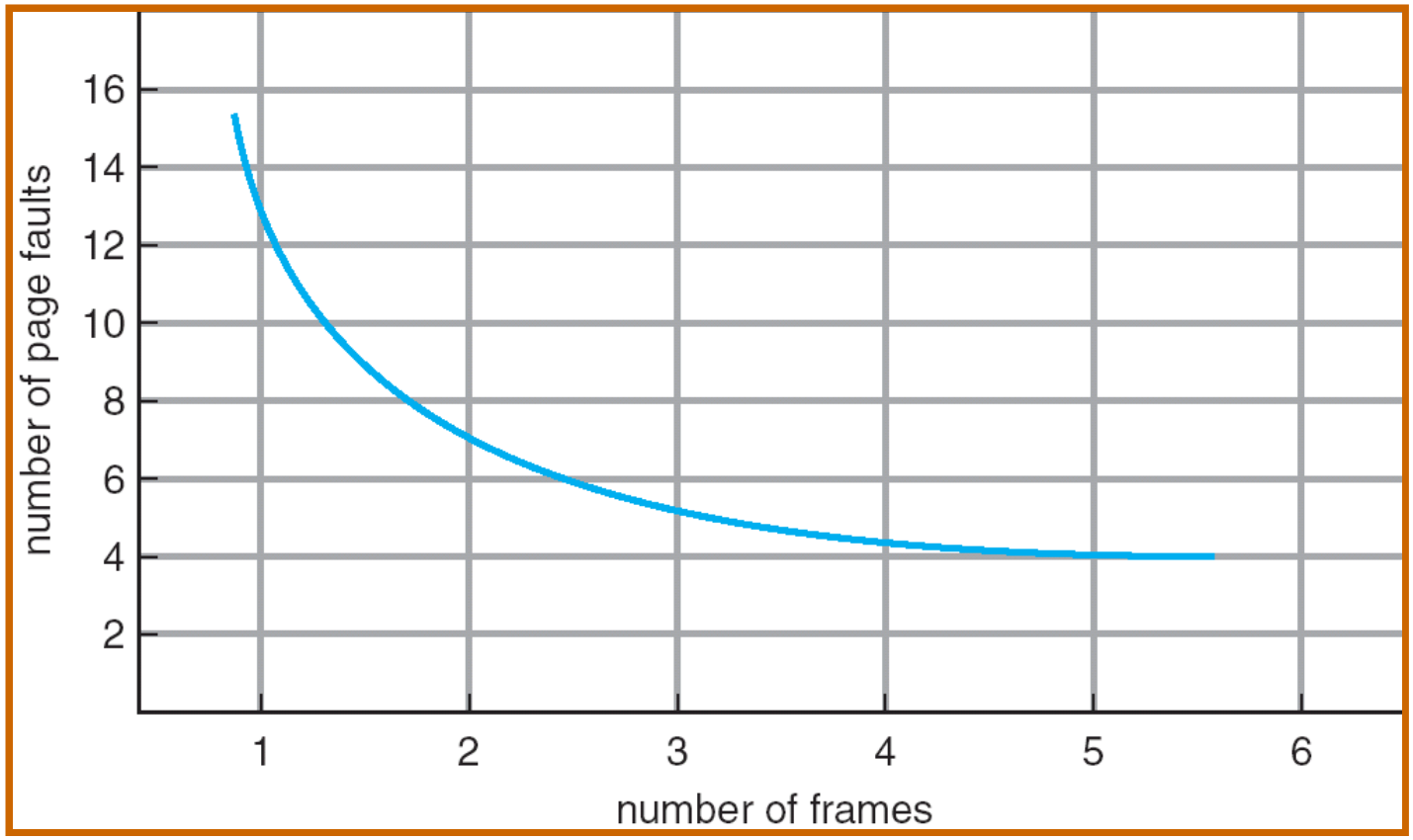
ALGORITMA PAGE REPLACEMENT

- Tujuan ⇒ Mencari algoritma dengan *page fault rate* terkecil
- Evaluasi algoritma → Hitung *page fault* dari deretan alamat logic yang diakses oleh CPU.
- Misalnya, akses terhadap alamat:

  0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
- Jika 1 page = 100 byte, maka akses alamat diatas dapat disederhanakan dengan penomoran (*reference string*):

  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

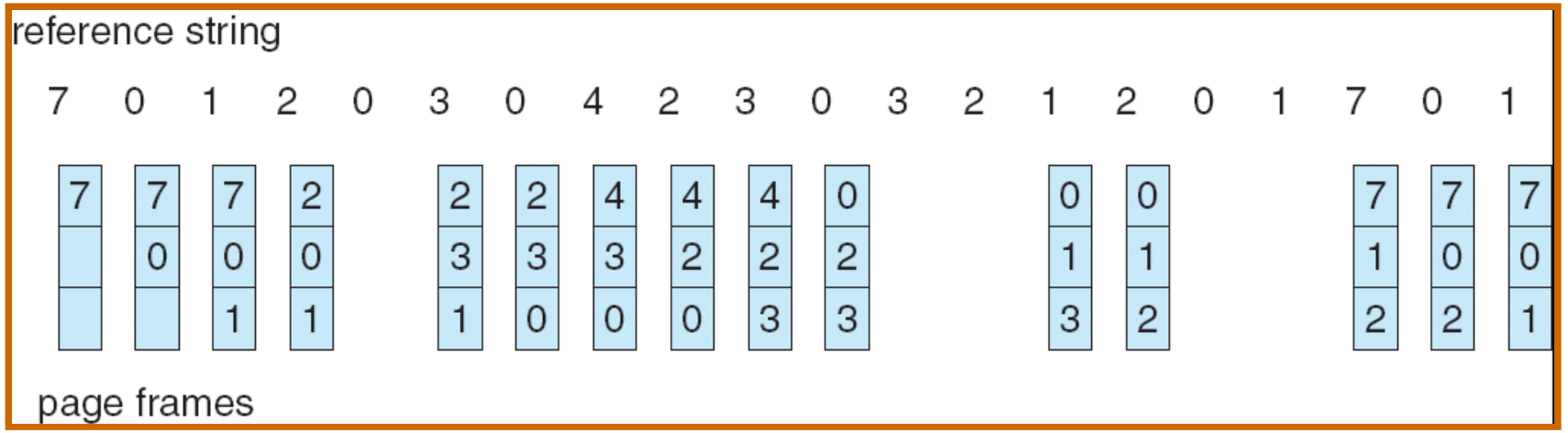
GRAFIK PAGE FAULT VS JUMLAH FRAME



Semakin besar jumlah frame, jumlah page fault semakin kecil

ALGORITMA FIFO

Page yang menempati memori paling lama dipilih untuk diganti



Total page fault = 15

Awalnya 3 frame kosong. Secara berturut-turut terjadi page fault untuk akses ke page 7,0 dan 1 !

(7,0,1)->(2,0,1)
2 menggantikan 7 karena page 7 paling tua

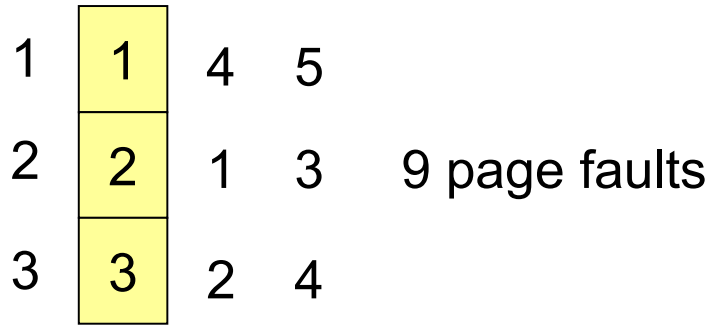
(2,0,1)->(2,3,1)
3 menggantikan 0 karena page 0 paling tua

FIFO

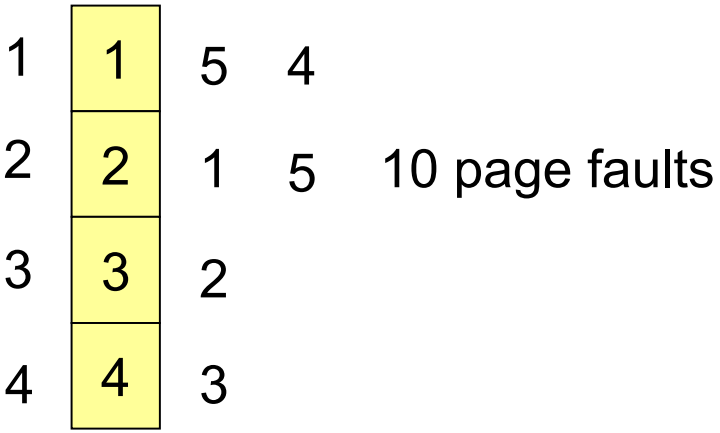
- FIFO
  - Pros
    - page lama berisikan inisialisasi modul yang tidak digunakan
  - Cons
    - Page lama berisikan inisialisasi variabel yang masih digunakan
- Akibat salah memilih page ‘korban’
  - Page Fault bertambah
  - Memperlambat Eksekusi

FIFO

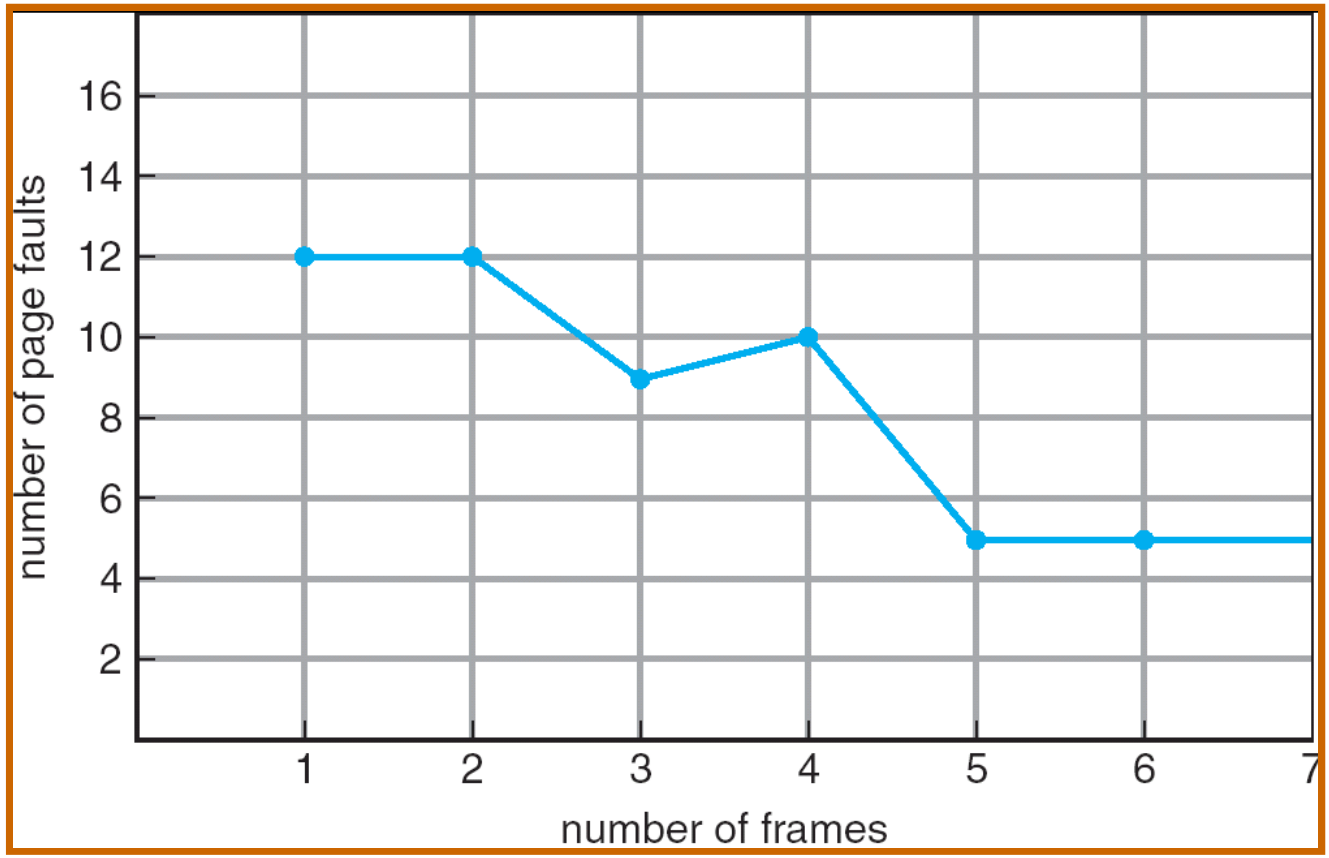
- Referensi : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frame



- 4 frames



FIFO-ANOMALY BELADY

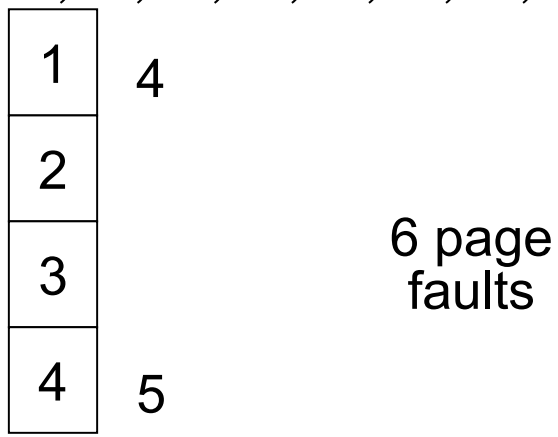


Anomali Belady: frame bertambah → page fault bertambah

ALGORITMA OPTIMAL

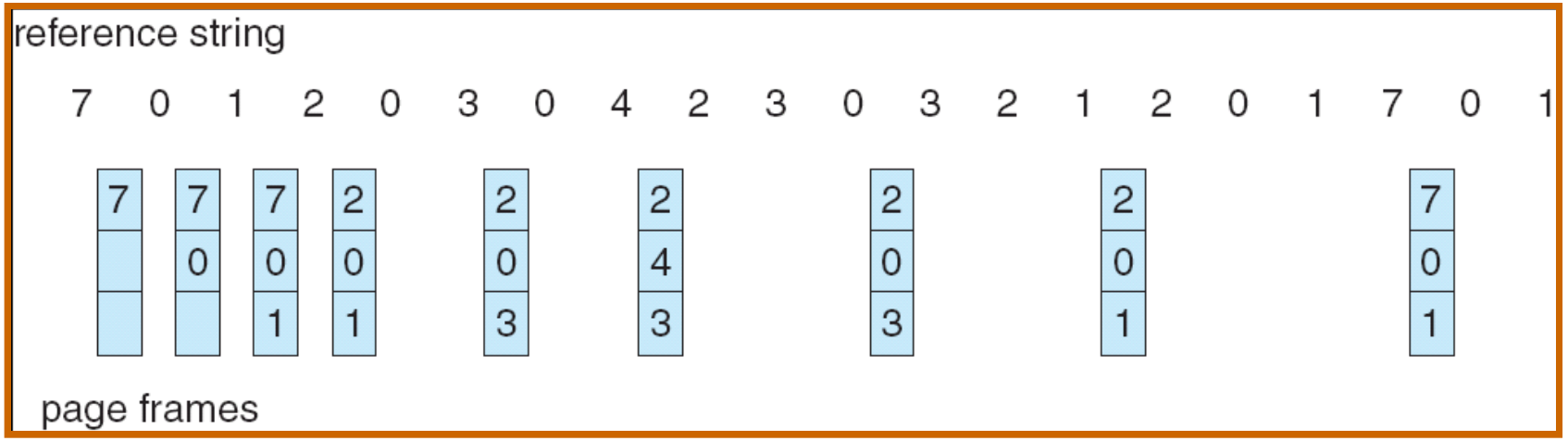
- Ganti page yang tidak akan digunakan pada periode berikutnya dengan waktu gilir yang terlama.
- 4 frame

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- Mempunyai jumlah *page fault* paling rendah
- Bagaimana cara memprediksinya?
- Sulit untuk diimplementasikan

ALGORITMA OPTIMAL



Total page fault = 9

(7,0,1)->(2,0,1)
2 menggantikan 7 karena 7 tidak akan digunakan hingga referensi ke 18

(2,0,1)->(2,0,3)
3 menggantikan 1 karena waktu referensi kembali untuk 1 lebih lama diantara 2 dan 0

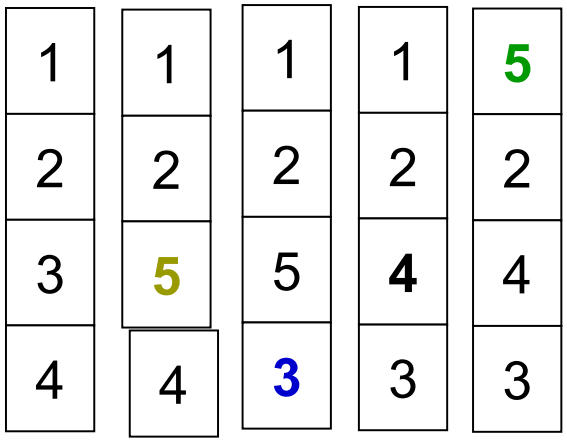
ALGORITMA LEAST RECENTLY USED (LRU)

- Perbedaan FIFO & OPT
  - FIFO ➔ Ganti page yang paling tua
  - OPT ➔ Ganti page yang punya waktu paling lama akan digunakan/direferensi kembali

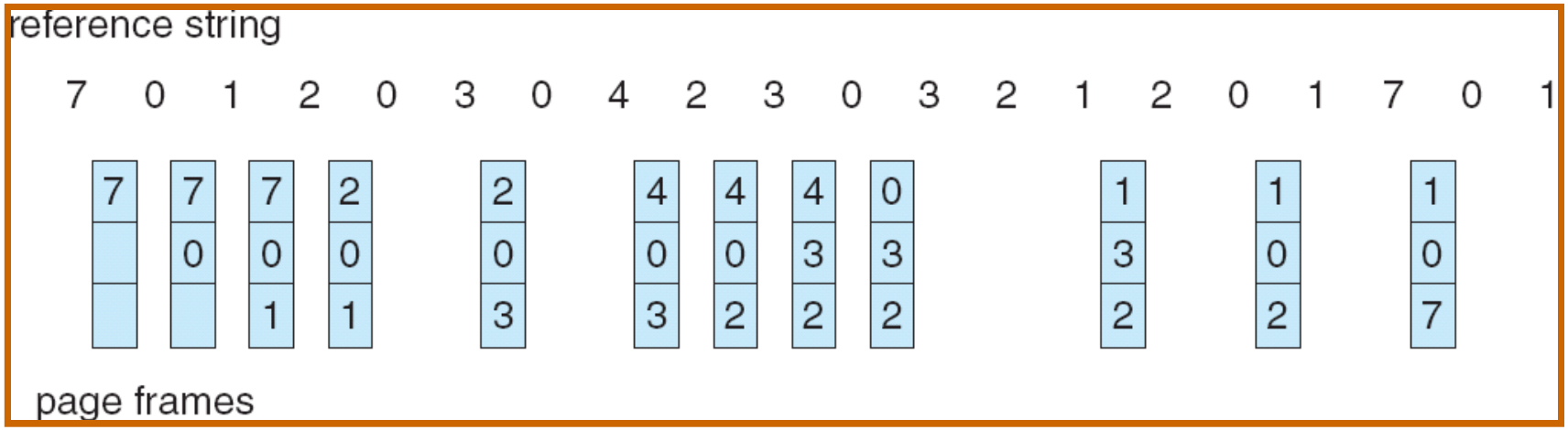
- LRU ➔ ganti page yang paling lama belum diakses

ALGORITMA LEAST RECENTLY USED (LRU)

- Contoh :
  - Akses page: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



ALGORITMA LEAST RECENTLY USED (LRU)



Total page fault = 12

(2,0,1) -> (2,0,3)
3 menggantikan 1 karena page 1 paling lama belum diakses

(2,0,3) -> (4,0,3)
4 menggantikan 2 karena page 2 paling lama belum diakses

ALGORITMA LEAST RECENTLY USED (LRU)

- Implementasi LRU dengan struktur data:
  - Clock Counter
  - Stack
- Clock Counter
  - Setiap entri page punya field counter
  - Ganti page yang mempunyai counter paling kecil

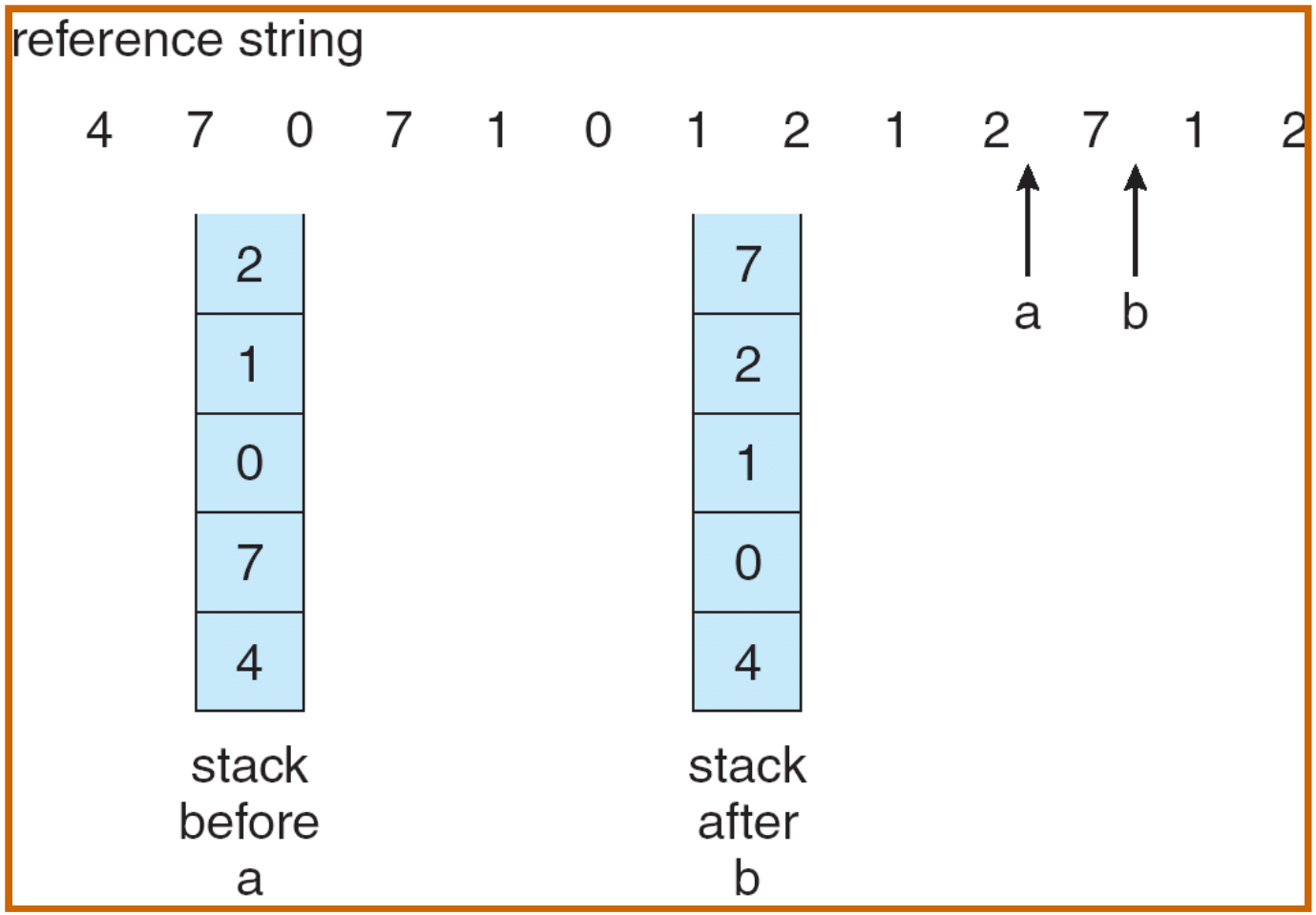
LRU DENGAN COUNTER CLOCK

page:	faults:	frames:			time:		
7	1	7	-1	-1	1		
0	2	7	0	-1	1	2	
1	3	7	0	1	1	2	3
2	4	2	0	1	4	2	3
0	4	2	0	1	4	5	3
3	5	2	0	3	4	5	6
0	5	2	0	3	4	7	6
4	6	4	0	3	8	7	6
2	7	4	0	2	8	7	9
3	8	4	3	2	8	10	9
0	9	0	3	2	11	10	9
3	9	0	3	2	11	12	9
2	9	0	3	2	11	12	13
1	10	1	3	2	14	12	13
2	10	1	3	2	14	12	15
0	11	1	0	2	14	16	15
1	11	1	0	2	17	16	15
7	12	1	0	7	17	16	18
0	12	1	0	7	17	19	18
1	12	1	0	7	20	19	18

ALGORITMA LEAST RECENTLY USED (LRU)

- Stack
  - Setiap ada referensi page, pindahkan page ke posisi paling atas
  - Page yang paling sering diakses (*most recently used*) berada diposisi atas.
  - Page yang paling jarang diakses (*least recently used*) berada diposisi bawah.

ALGORITMA LEAST RECENTLY USED (LRU)



Penggunaan stack

LRU - STACK

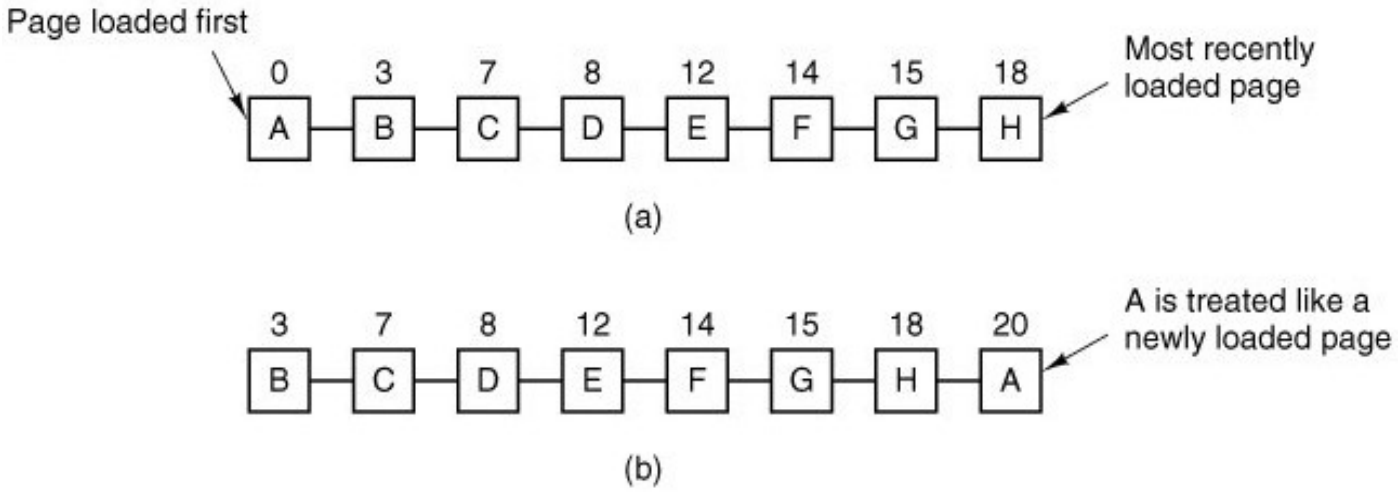
page:	faults:	frames:			stack:		
7	1	7	-1	-1	7		
0	2	7	0	-1	7	0	
1	3	7	0	1	7	0	1
2	4	2	0	1	0	1	2
0	4	2	0	1	1	2	0
3	5	2	0	3	2	0	3
0	5	2	0	3	2	3	0
4	6	4	0	3	3	0	4
2	7	4	0	2	0	4	2
3	8	4	3	2	4	2	3
0	9	0	3	2	2	3	0
3	9	0	3	2	2	0	3
2	9	0	3	2	0	3	2
1	10	1	3	2	3	2	1
2	10	1	3	2	3	1	2
0	11	1	0	2	1	2	0
1	11	1	0	2	2	0	1
7	12	1	0	7	0	1	7
0	12	1	0	7	1	7	0
1	12	1	0	7	7	0	1

ALGORITMA SECOND-CHANCE

- Algoritma *Second-Chance*
  - Modifikasi dari algoritma FIFO
    - Menghindari pergantian page tua yang masih diakses
    - Mencari page tua yang jarang diakses
  - Menggunakan bit referensi (*reference bit*)
    - Jika nilai bit = 0, page diganti
    - Jika nilai bit = 1
      - Ubah *arrival time* ke *current time*
      - Ubah nilai bit = 0
  - Page yang sering diakses akan selalu berada dimemori

ALGORITMA SECOND-CHANCE

- Page yang terpilih (*victim page*), digantikan dengan page baru pada posisi tersebut
- Jika reference bit semua adalah 0, algoritma second chance menyerupai FIFO

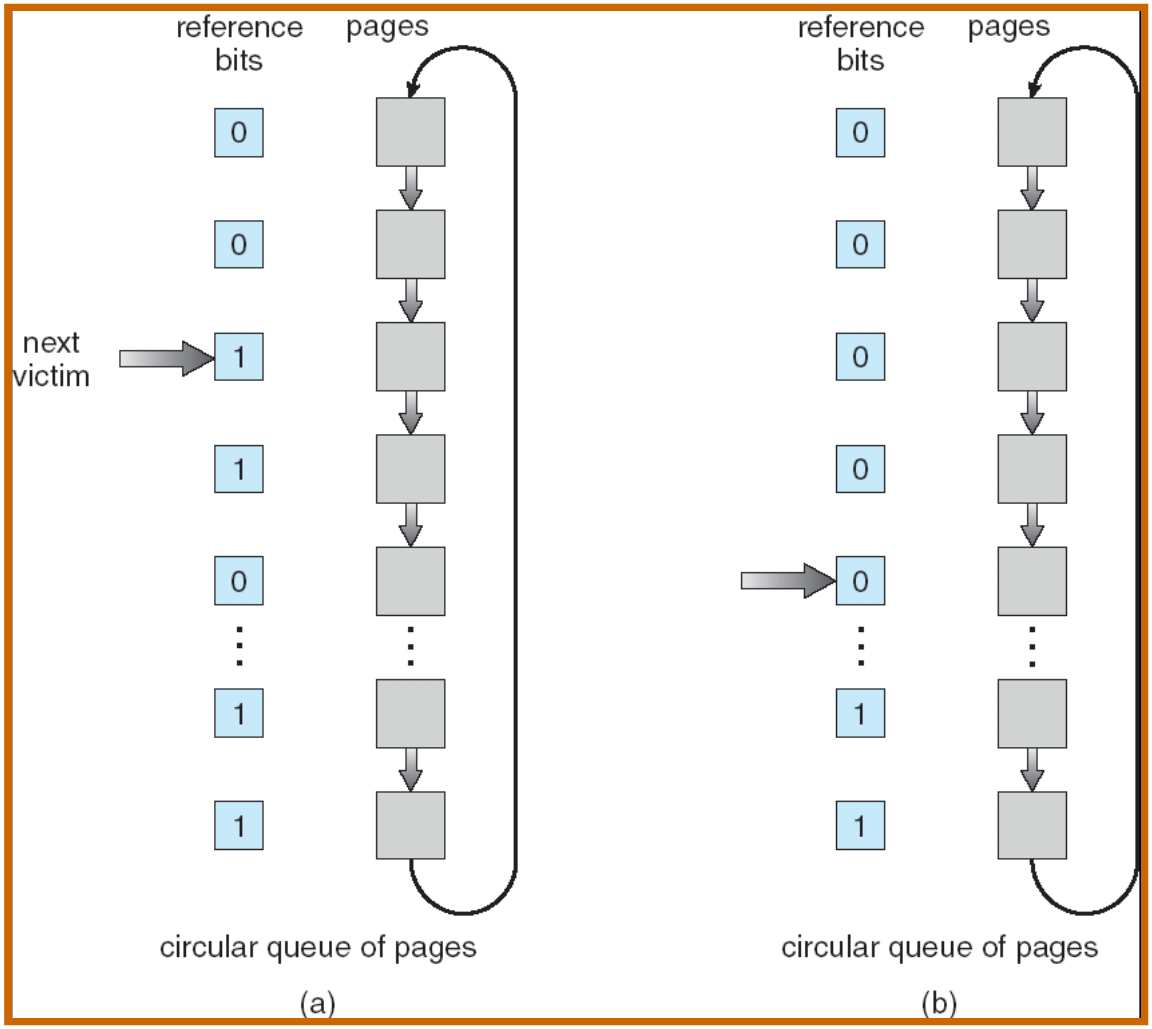


a. Deretan page dalam bentuk FIFO
b. Jika nilai ref bit page A=1, page A dipindah dari posisi *arrival time* (0) ke posisi *current time* (20)

CIRCULAR QUEUE (ALGORITMA *CLOCK*)

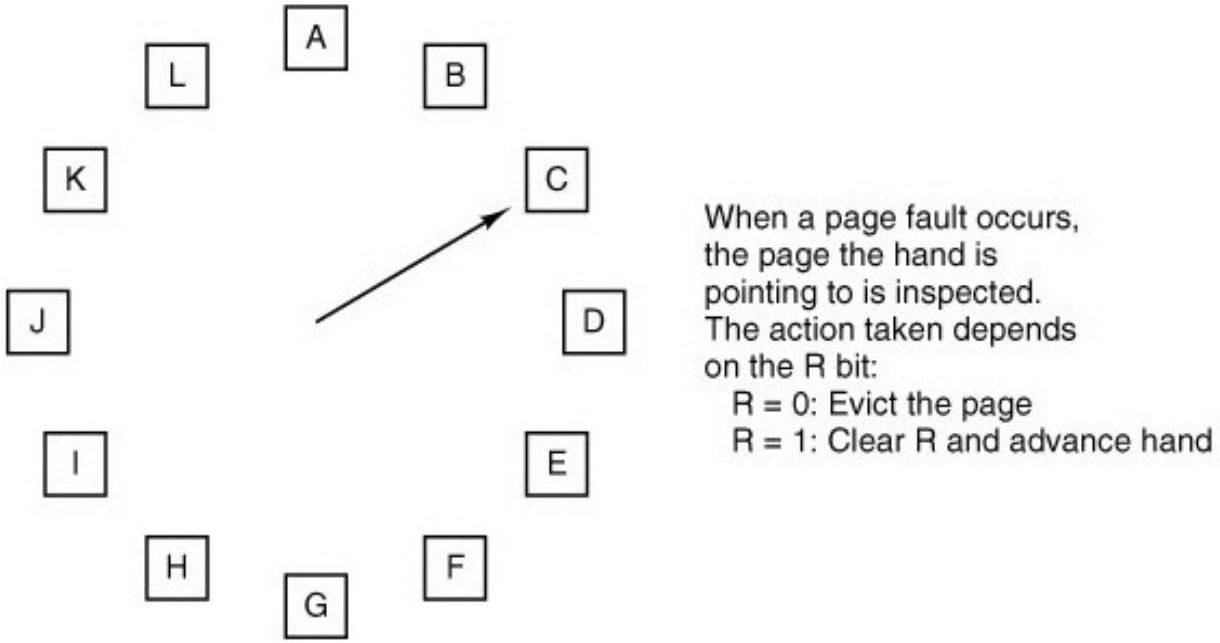
- Circular Queue (Algoritma *clock*)
  - Posisi page seakan-akan menyerupai lingkaran
  - Jika :
    - Nilai bit = 0, ganti page
    - Nilai bit = 1
      - Ubah nilai bit=0
      - Pointer bergerak ke page berikutnya searah jarum jam

CIRCULAR QUEUE (ALGORITMA *CLOCK*)



Ilustrasi 1

CIRCULAR QUEUE (ALGORITMA *CLOCK*)



Ilustrasi 2

CIRCULAR QUEUE (ALGORITMA *CLOCK*)

```
page: 7   faults: 1   frames:   7 -1 -1    ref:   1
      0             2           7  0 -1           1   1
      1             3           7  0  1           1   1   1
      2             4           2  0  1           1   0   0
      0             4           2  0  1           1   1   0
      3             5           2  0  3           1   0   1
      0             5           2  0  3           1   1   1
      4             6           4  0  3           1   0   0
      2             7           4  2  3           1   1   0
      3             7           4  2  3           1   1   1
      0             8           4  2  0           0   0   1
      3             9           3  2  0           1   0   1
      2             9           3  2  0           1   1   1
      1            10           3  1  0           0   1   0
      2            11           3  1  2           0   1   1
      0            12           0  1  2           1   1   1
      1            12           0  1  2           1   1   1
      7            13           0  7  2           0   1   0
      0            13           0  7  2           1   1   0
      1            14           0  7  1           1   1   1
```

ALGORITMA ENHANCED SECOND-CHANCE

- Referenced (R) = Read / Write
- Modify (M) = Write
- (R,M) = (Referenced bit, Modify bit)
  - Kelas 0 (0,0) : page belum pernah diakses dan dimodifikasi
  - Kelas 1 (0,1) : page belum pernah diakses namun pernah dimodifikasi
  - Kelas 2 (1,0) : page pernah diakses dan belum pernah dimodifikasi
  - Kelas 3 (1,1) : page pernah diakses dan dimodifikasi
- Prioritas pergantian page, mulai dari kelas yang paling rendah

PROSEDUR ENHANCED SECOND CHANCE

PASS :

1. Jika ditemukan frame kosong, gunakan frame tersebut.
2. Jika tidak ditemukan frame kosong:
   - Cari frame page kelas 0, pada frame.
     1. Jika ditemukan, ganti page kelas 0 dan pindahkan pointer ke frame tersebut.
     2. Jika tidak ditemukan, secara linear cari page kelas 1. Ubah referensi bit ke 0 untuk setiap page yang dilewati.
3. Jika PASS pertama tidak berhasil, ulangi PASS satu kali lagi

ALGORITMA ENHANCED SECOND-CHANCE (CONT.)

rpage:	7	faults:	1	frames:	7	-1	-1	ref,dirty:	1,0	
	0		2		7	0	-1		1,0	1,0
wpage:	1		3		7	0	1		1,0	1,0 1,1
	2		4		2	0	1		1,0	0,0 0,1
	0		4		2	0	1		1,0	1,0 0,1
	3		5		2	0	3		1,0	0,0 1,0
wpage:	0		5		2	0	3		1,0	1,1 1,0
	4		6		4	0	3		1,0	0,1 0,0
	2		7		4	0	2		1,0	0,1 1,0
	3		8		4	3	2		0,0	1,0 1,0
	0		9		0	3	2		1,0	1,0 0,0
	3		9		0	3	2		1,0	1,0 0,0
	2		9		0	3	2		1,0	1,0 1,0
	1		10		0	1	2		0,0	1,0 0,0
	2		10		0	1	2		0,0	1,0 1,0
	0		10		0	1	2		1,0	1,0 1,0
	1		10		0	1	2		1,0	1,0 1,0
	7		11		0	1	7		0,0	0,0 1,0
	0		11		0	1	7		1,0	0,0 1,0
	1		11		0	1	7		1,0	1,0 1,0

ALGORITMA COUNTING-BASED

- Rekam counter dari jumlah akses masing-masing page
- **Algoritma LFU (Least Frequently Used)**: ganti page yang mempunyai jumlah frekuensi akses paling sedikit
- **Algoritma MFU (Most Frequently Used)**: ganti page yang mempunyai jumlah frekuensi akses paling tinggi.

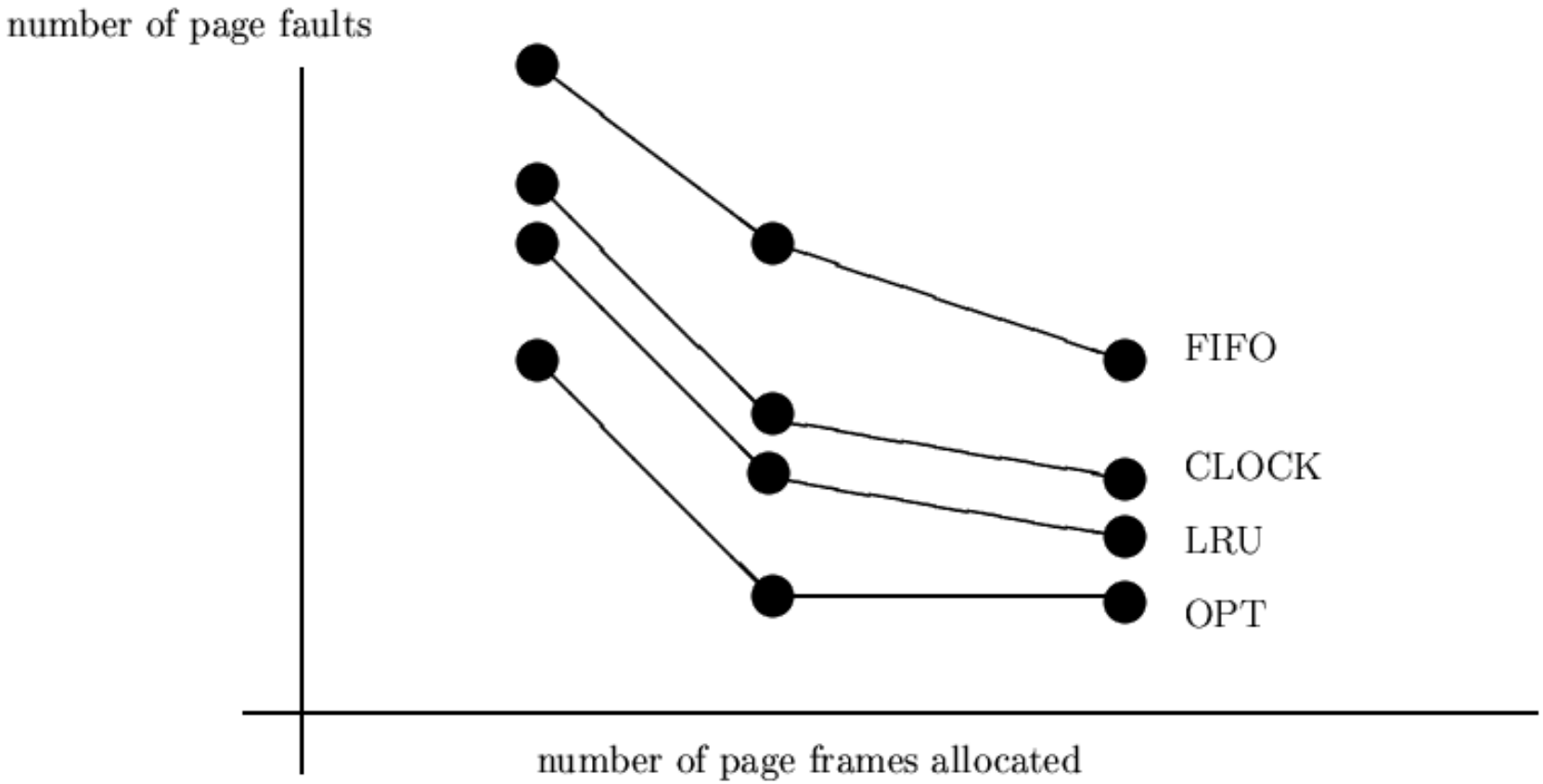
LFU

page:	faults:	frames:			count:		
7	1	7	-1	-1	1		
0	2	7	0	-1	1	1	
1	3	7	0	1	1	1	1
2	4	2	0	1	1	1	1
0	4	2	0	1	1	2	1
3	5	2	0	3	1	2	1
0	5	2	0	3	1	3	1
4	6	4	0	3	1	3	1
2	7	4	0	2	1	3	1
3	8	3	0	2	1	3	1
0	8	3	0	2	1	4	1
3	8	3	0	2	2	4	1
2	8	3	0	2	2	4	2
1	9	3	0	1	2	4	1
2	10	3	0	2	2	4	1
0	10	3	0	2	2	5	1
1	11	3	0	1	2	5	1
7	12	3	0	7	2	5	1
0	12	3	0	7	2	6	1
1	13	3	0	1	2	6	1

MFU

page:	faults:	frames:			count:		
7	1	7	-1	-1	1		
0	2	7	0	-1	1	1	
1	3	7	0	1	1	1	1
2	4	2	0	1	1	1	1
0	4	2	0	1	1	2	1
3	5	2	3	1	1	1	1
0	6	2	3	0	1	1	1
4	7	4	3	0	1	1	1
2	8	4	2	0	1	1	1
3	9	4	2	3	1	1	1
0	10	0	2	3	1	1	1
3	10	0	2	3	1	1	2
2	10	0	2	3	1	2	2
1	11	0	1	3	1	1	2
2	12	0	1	2	1	1	1
0	12	0	1	2	2	1	1
1	12	0	1	2	2	2	1
7	13	7	1	2	1	2	1
0	14	7	0	2	1	1	1
1	15	7	0	1	1	1	1

KINERJA ALGORITMA PERGANTIAN PAGE



LATIHAN

Sebuah proses mempunyai 3 frame menggunakan algoritma LRU untuk melakukan pergantian page. Antrian page mempunyai format (waktu, nomor page), secara berturut-turut adalah (0,4), (1,7), (2,4), (3,1), (4,7), (5,2), (6,9), (7,1), (8,7), (9,9),(10,4). Secara berturut-turut, bagaimanakah posisi frame saat page fault ke 6 dan 7?

A. 2,7,1 dan 2,7,9
B. 2,1,9 dan 7,1,9
C. 2,7,9 dan 2,1,9
D. 7,1,9 dan 7,4,9
E. 4,7,1 dan 2,7,1

LATIHAN

Frame	0	1	2	3	4	5	6	7
Page	17	32	41	5	7	13	2	20
Reference Bit	1	0	0	0	0	1	1	0

- Berdasarkan tabel diatas, diketahui frame 0 berisikan page 17, frame 1 berisikan page 32, dan seterusnya. Diasumsikan sistem menggunakan algoritma clock untuk melakukan pergantian page. Pada baris ketiga terdapat reference bit yang digunakan sistem saat ini. Reference bit diset 1 jika page dipindahkan ke memori.
- Saat ini posisi pointer berada pada frame ke 3. Frame ke 4 belum dieksekusi sama sekali. Jika terjadi permintaan page dengan referensi string sebagai berikut : 32, 14, 15, 2, 18, Jawablah pertanyaan berikut:

Isilah kolom berikut:

Frame	0	1	2	3	4	5	6	7
Page								
Reference Bit								

Berapa jumlah page yang diganti ?
Page mana saja yang diganti ?
Pada frame berapa, posisi terakhir dari pointer clock ?

C. ALOKASI FRAME

ALOKASI FRAME

- Berapa frame yang cukup untuk sebuah proses?
  - Equal Allocation
  - Proportional Allocation
  - Priority Allocation
- Equal Allocation

– Setiap proses mendapatkan jumlah frame yang sama
  - Jika total frame = 93 frame dan jumlah proses = 5
  - Maka 1 proses (93/5) = 18 frame, 3 frame sisanya bisa digunakan untuk *frame buffer*

ALOKASI FRAME

- Proportional Allocation
  - Alokasi frame secara proporsional berdasarkan besar proses

s_i= size of process p_i

$S=\sum s_i$

m= total number of frames

a_i= allocation for $p_i=\frac{s_i}{S}\times m$

$m=64$

$s_i=10$

$s_2=127$

$a_1=\frac{10}{137}\times 64\approx 5$

$a_2=\frac{127}{137}\times 64\approx 59$

ALOKASI FRAME

- Priority Allocation
  - Proses dengan prioritas tinggi lebih banyak mendapatkan frame

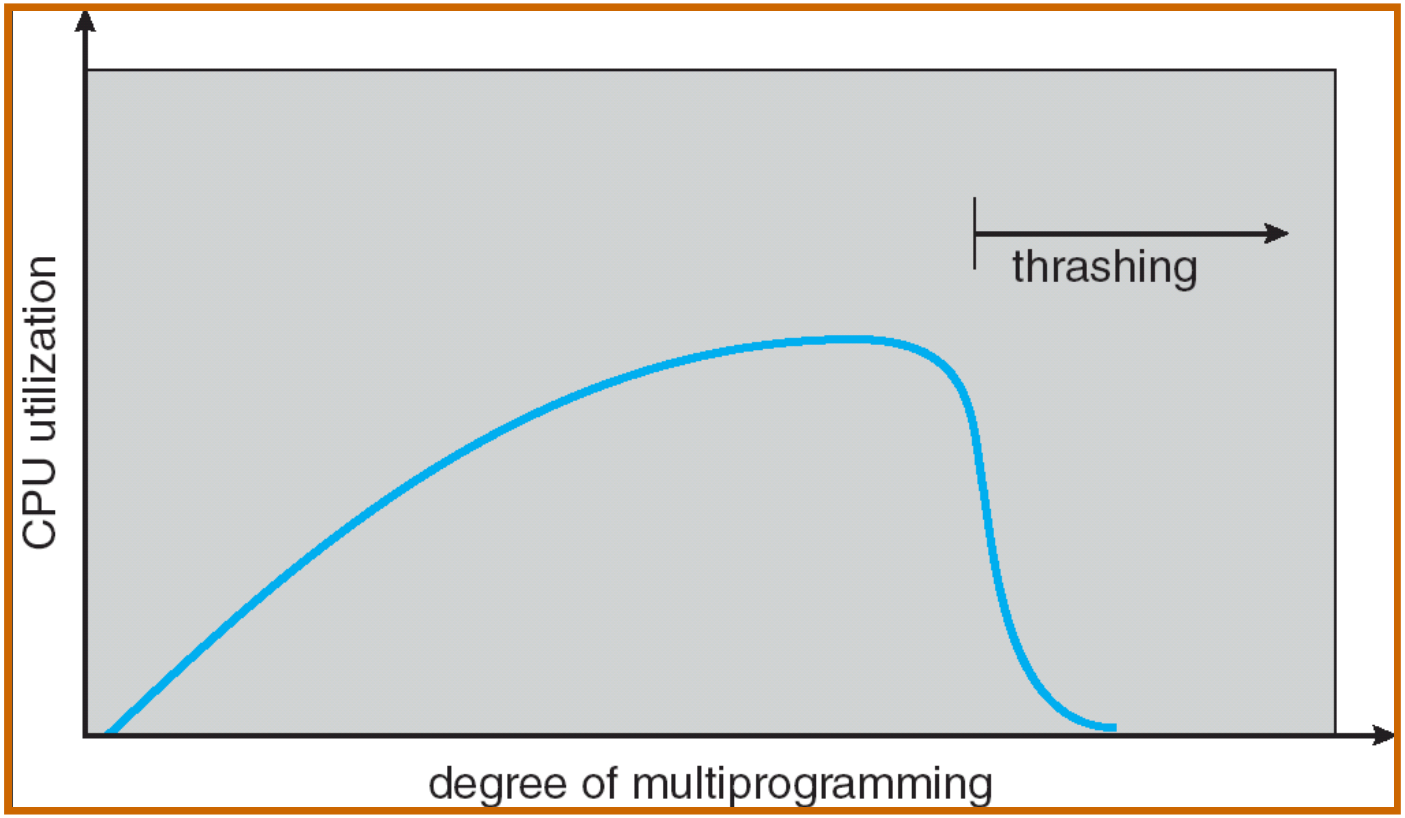
KATEGORI PERGANTIAN PAGE

- Global Replacement
  - Mengambil frame yang tersedia di Memori Fisik
- Local Replacement
  - Mengambil frame dari total frame yang diperuntukkan oleh sebuah proses

TRASHING

- Trashing ➔ terjadi page fault untuk setiap eksekusi instruksi
- Dampak: Processor lebih sibuk memindahkan page daripada mengeksekusi instruksi

THRASHING



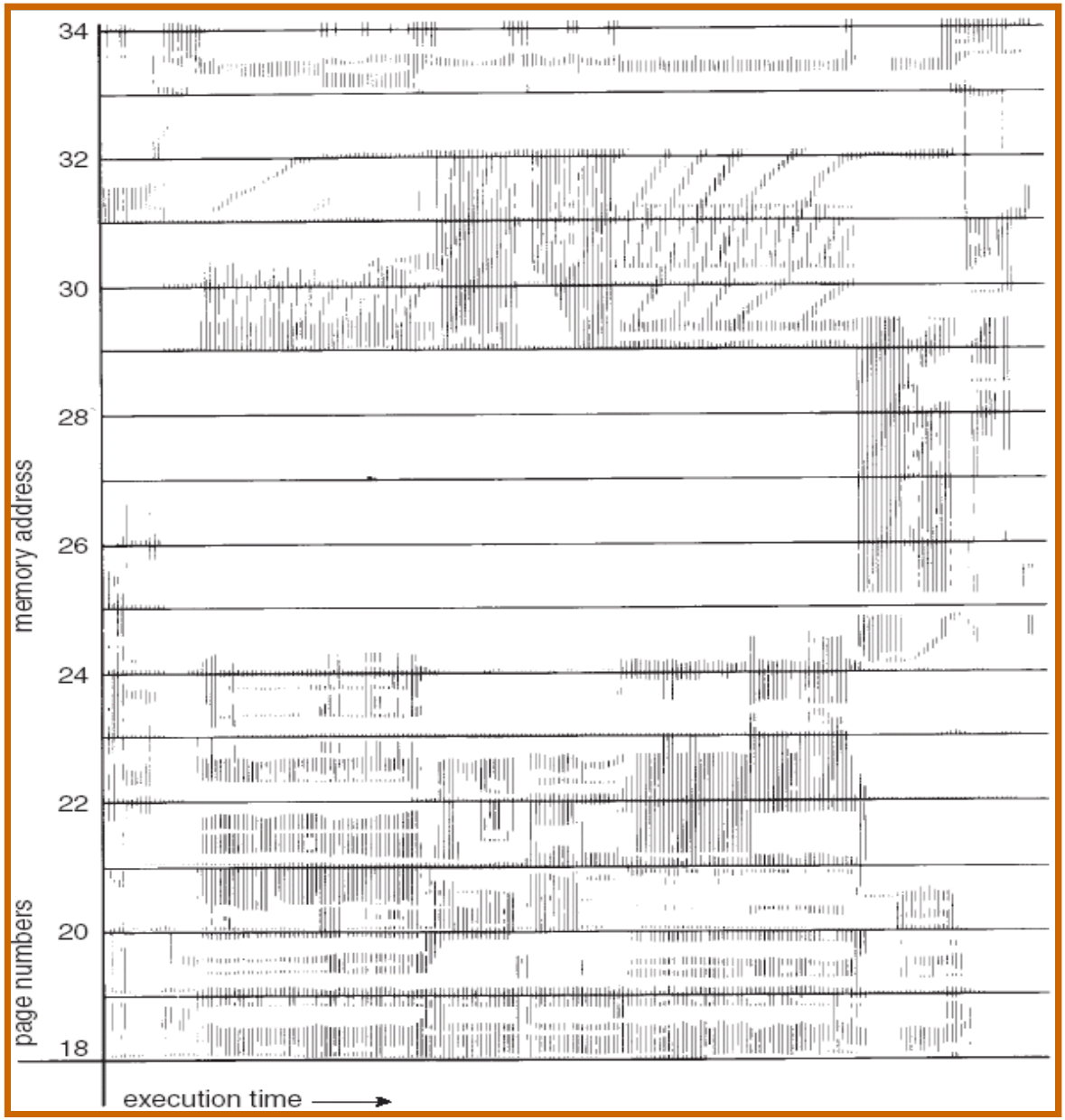
THRASHING

- Alur kejadian :
  1. Jumlah frame yang digunakan sebuah proses kurang dari jumlah page yang aktif
  2. Terjadi page fault
  3. Terjadi pergantian page
  4. Semua page dalam frame adalah page aktif
  5. Page aktif diganti padahal akan segera digunakan kembali
  6. Dalam waktu singkat, page fault terjadi, dilanjutkan dengan pergantian page, dan seterusnya.
  7. Frekuensi page fault meningkat

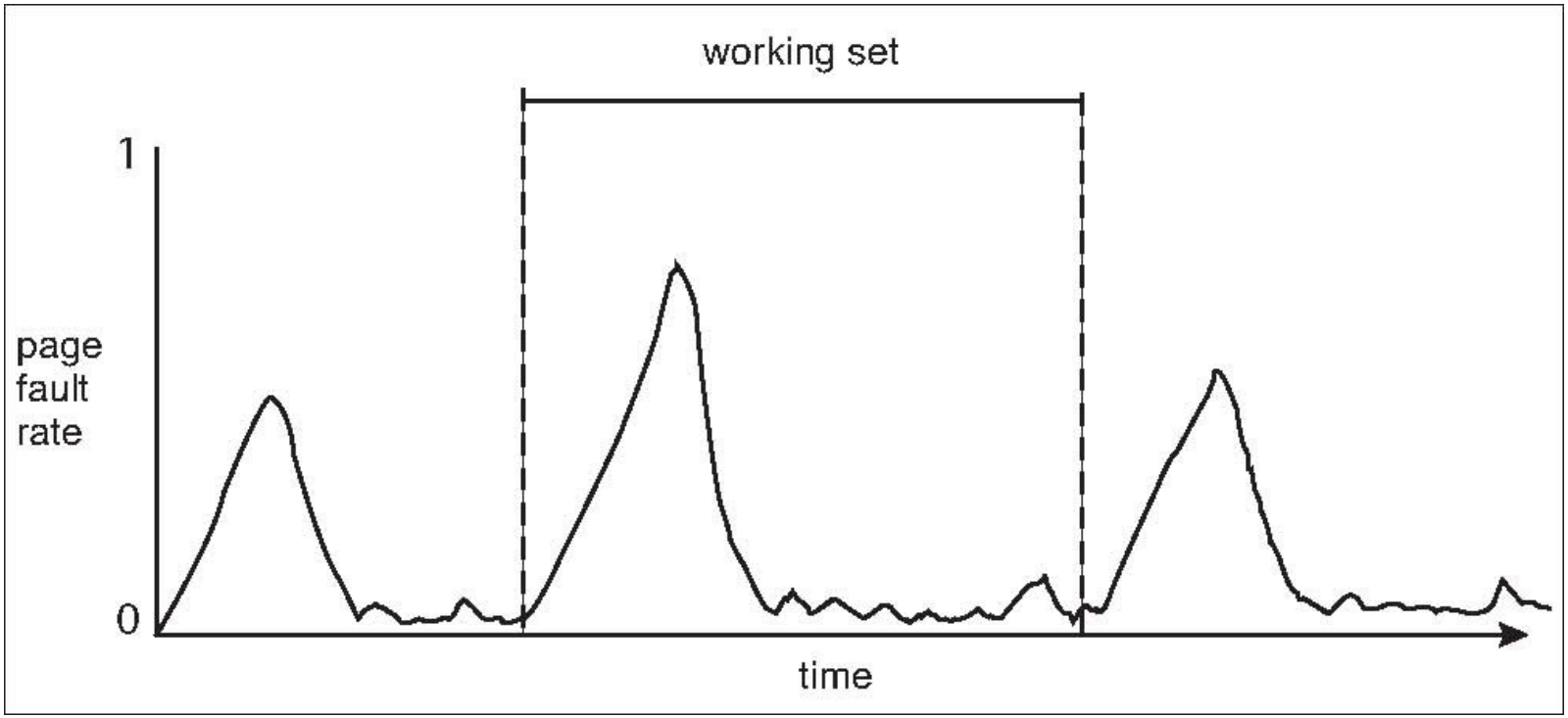
BEBERAPA SOLUSI TRASHING

- Local Replacement
  - Hanya mengambil frame milik sendiri
  - Kurang efektif, menyebabkan antrian untuk mengakses akses perangkat paging
- Working Set (WS)
  - WS → Sekumpulan page yang aktif diakses pada interval waktu tertentu
  - Berikan sejumlah frame sesuai dengan besaran working set
- Indikator Frekuensi Page Fault
  - Menggunakan batas frekuensi upper bound dan lowerbound

ILUSTRASI AKSES SEJUMLAH PAGE

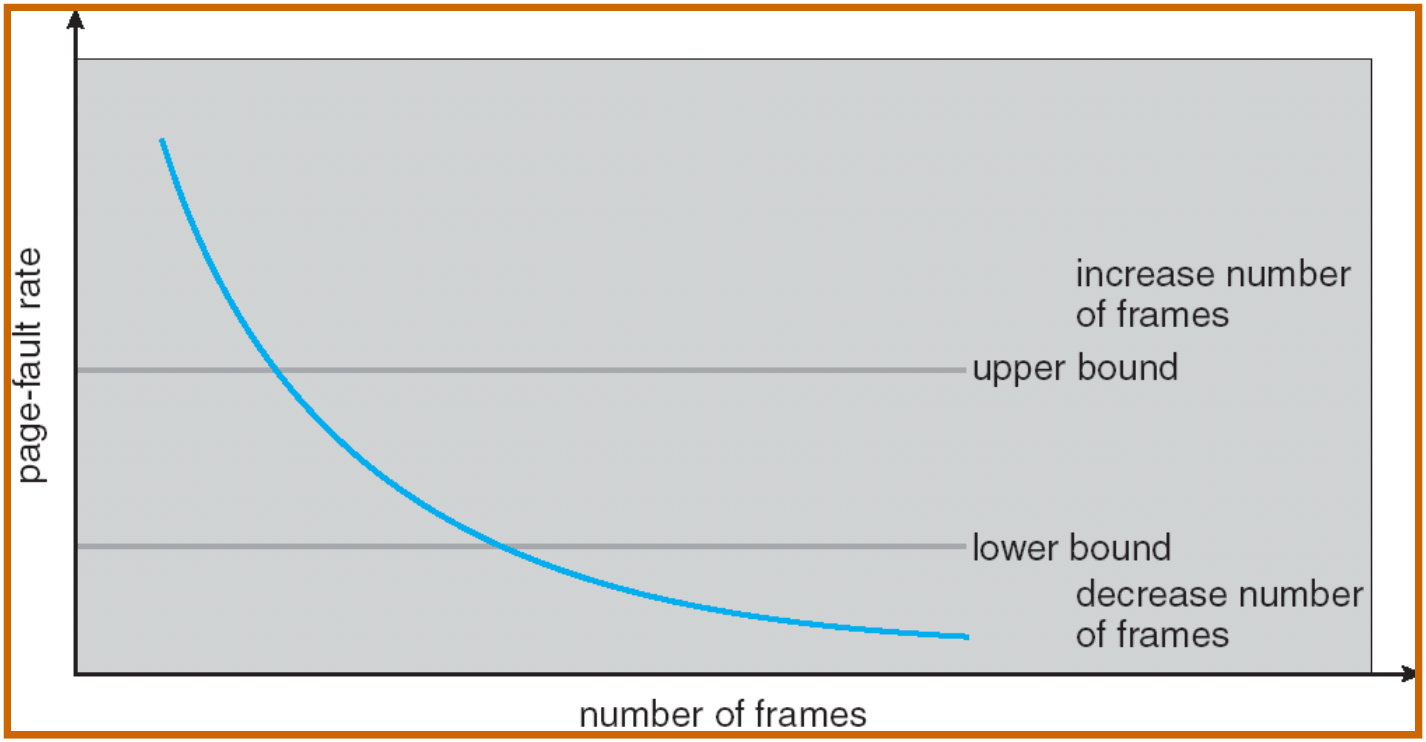


WORKING SET DAN PAGE FAULT



INDIKATOR FREKUENSI PAGE-FAULT

- Pencegahan trashing dengan Frekuensi Page Fault
  - Dibawah lower bound → bebaskan frame kosong yang dimiliki proses
  - Diatas upper bound → proses membutuhkan tambahan frame. Jika tidak ada frame kosong yang tersedia ?



D. ALOKASI MEMORI KERNEL

TUJUAN PEMBELAJARAN

- Memahami alokasi memori kernel
  - Buddy system
  - Slab Allocator
- Memahami topik tambahan :
  - Prepaging
  - Ukuran page
  - TLBReach
  - Struktur Program

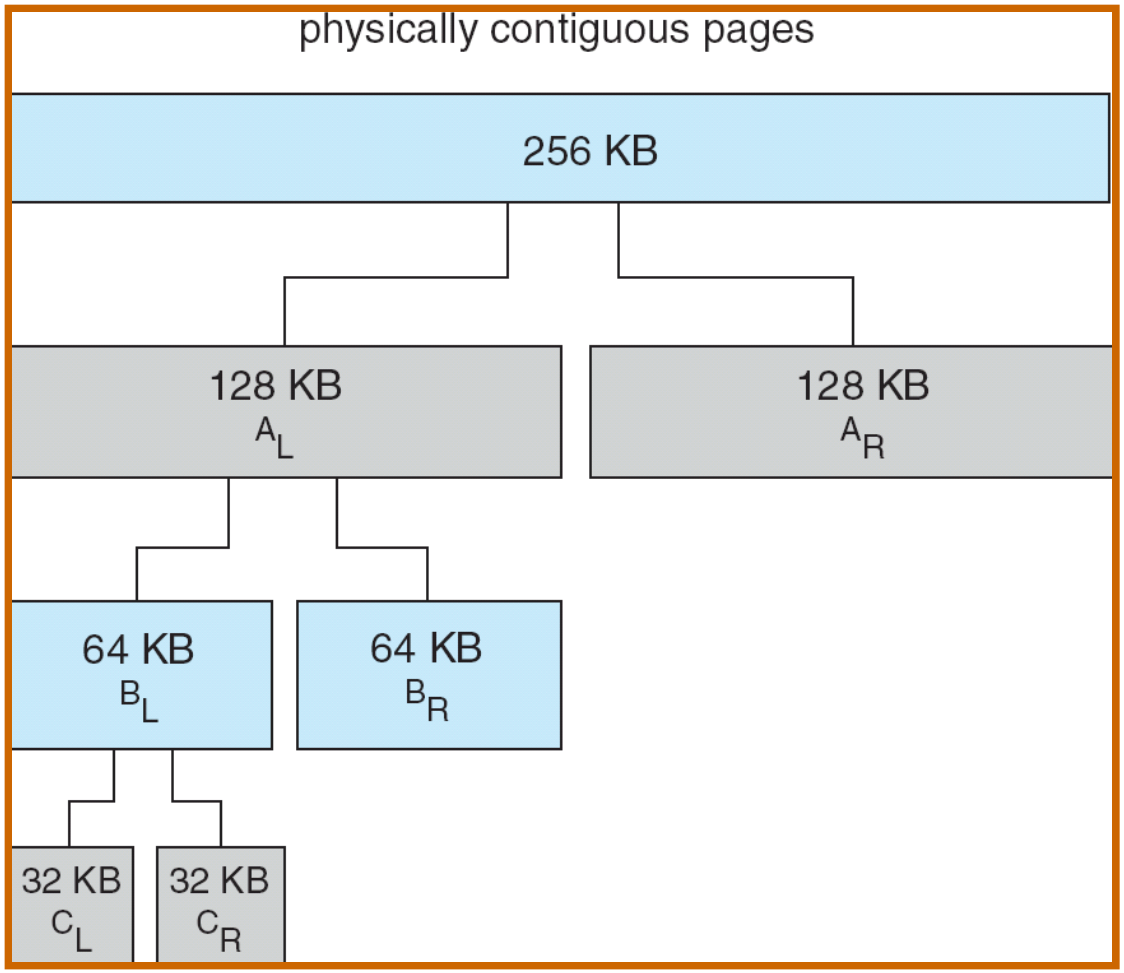
ALOKASI KERNEL MEMORI

- Alokasi memory untuk proses kernel berbeda dengan proses user
  - Kebutuhan memory kernel bervariasi
  - Beberapa perangkat keras mengakses page secara sekuensial pada alamat memori paling bawah.
- Manajemen memori untuk kernel
  - *Buddy system*
  - *Slab allocation*

BUDDY SYSTEM

- Alokasi memori menggunakan segmen dengan ukuran segmen tetap.
- Segmen berisikan page yang berurutan (*contiguous*)
- Kelebihan : *buddy* yang berdekatan dapat digabung ➔*coalescing*
- Kekurangan : fragmentasi internal

BUDDY SYSTEM ALLOCATOR



Jika besar segmen = 256Kb, kernel membutuhkan 21kb, segmen dibagi menjadi dua *buddy*, salah satu *buddy* dibagi kembali hingga *buddy* memenuhi kebutuhkan kernel.

ALOKASI SLAB

- **Slab** terdiri dari satu atau lebih dari satu page
- **Cache** terdiri dari satu atau lebih dari satu *slab*
- Satu cache untuk satu struktur data kernel yang unik
  - Misalnya cache untuk semaphore, process descriptor, file, dan sebagainya
  - Saat cache dibuat, objek dibuat, diberi tanda *free* artinya bebas digunakan

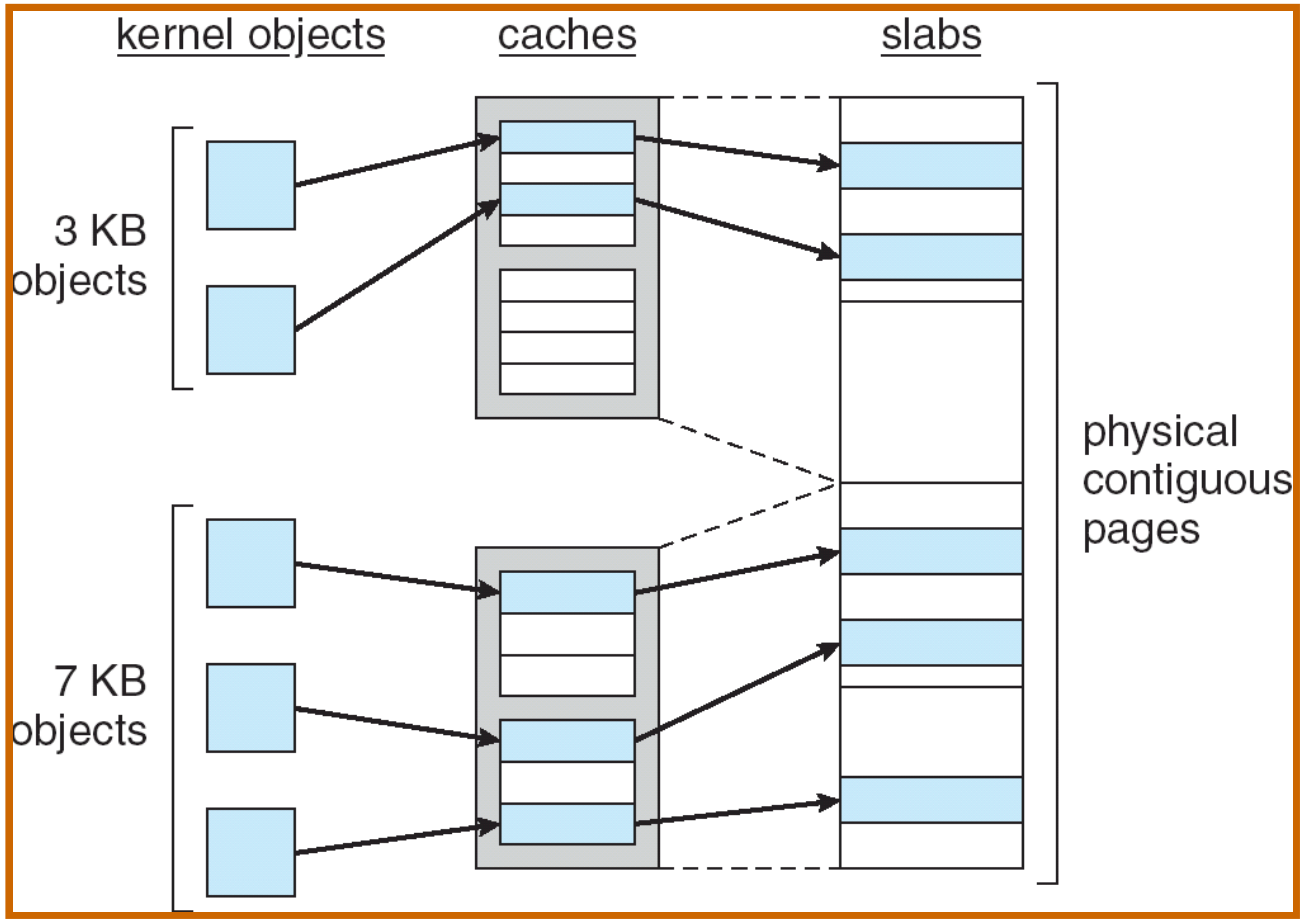
SLAB ALLOCATOR

- Jumlah objek yang bisa ditampung tergantung dari besaran slab dalam cache
  - Misalnya 12Kb *slab* (terdiri dari 3 page dengan ukuran page masing 4kb) dapat menampung enam objek (satu objek besarnya 2kb)
- Objek diberi tanda bit **used**, jika sedang digunakan dan bit **free** jika tidak digunakan
- Tiga status slab dalam linux
  - Full, semua objek dalam slab used
  - Empty, semua objek dalam slab *free*
  - Partial, slab terdiri dari objek yang used dan free
- Pencarian slab, mulai dari slab :
  - Partial → empty → buat slab baru

SLAB ALLOCATOR

- Keuntungan
  - Fragmentasi internal dapat diminimalisir
    - Setiap struktur data kernel mempunyai *dedicated cache* masing-masing
  - Permintaan memori dapat dipenuhi dengan cepat
    - Objek dibuat diawal pembuatan cache
    - Proses selesai → object diset free

SLAB ALLOCATION



TOPIK TAMBAHAN

- Prepaging
- Ukuran page
- TLBReach
- Struktur Program

PREPAGING

- Prepaging → memindahkan sejumlah page ke memori fisik sebelum proses dieksekusi.
- Tujuan : mencegah tingginya page fault seperti yang terjadi *pure demand paging*.
- Page yang dipindahkan adalah page yang berada pada rentang working set

PREPAGING

- Jika :
  - s adalah jumlah page yang dipilih untuk dipindahkan ke memori fisik (prepaging)
  - α adalah perbandingan jumlah page yang diakses dengan s, dimana nilai a berada pada rentang $0 \leq \alpha \leq 1$

  - Jika $s * \alpha > s * (1- \alpha) \Rightarrow$ cost-efektif dalam mengurangi page fault
  - Jika $s * \alpha < s * (1- \alpha) \Rightarrow$ biaya prepaging lebih tinggi dibanding tanpa prepaging
  - α mendekati $0 \Rightarrow$ prepaging gagal
  - α mendekati $1 \Rightarrow$ prepaging efektif

UKURAN PAGE

- Faktor yang menjadi pertimbangan untuk menentukan ukuran page
  - Fragmentasi internal
  - Besar page table
  - Waktu I/O
    - Waktu I/O = Seek time + Latency Time + Tranfer time
  - *Locality of Reference*
  - Jumlah memori yang dipakai
  - Jumlah Page fault
  - Eksekusi Parsial

UKURAN PAGE

- Contoh perhitungan waktu I/O untuk menulis dan membaca page.
  - Diketahui:
    - Transfer rate dari memory ke HD dan sebaliknya = 2MBps (artinya, butuh 0.2 ms untuk mentransfer 512 bytes)
    - seek time=20 ms, latency time= 8ms
    - Ukuran 1 page = 512 byte
    - total waktu I/O = 28,2ms → 512bytes
    - Waktu I/O untuk transfer 1 page = seek time + latency time + tranfer time = 20ms + 8ms + 0.2ms = 28.2ms
    - Waktu I/O untuk 2 page dengan ukuran yang sama?
    - Waktu I/O untuk 1page dengan ukuran = 1024 byte?

UKURAN PAGE

- Locality of Reference by example
  - Besar proses = 200 kb; yang dieksekusi=100 kb
  - Jika ukuran page > 200 Kb, eksekusi parsial (100kb) tidak dapat dilakukan
    - Terdapat bagian page yang tidak digunakan
    - Butuh memori besar
  - Jika ukuran page < 200 Kb (misal = 1 byte), eksekusi parsial (100kb) dapat dilakukan
    - Page fault rate tinggi

PAGE KECIL VS PAGE BESAR

- Ukuran page kecil
  - Keuntungan : butuh sedikit memori untuk menampung page proses, fragmentasi internal rendah
  - Kerugian : frekuensi *page-fault* tinggi, waktu I/O tinggi, butuh memori besar untuk menampung *page table*
- Ukuran page besar
  - Keuntungan: butuh sedikit memori untuk menampung page table, waktu I/O sedikit dan *page-fault* rendah
  - Kerugian : fragmentasi internal tinggi, butuh memori besar untuk menampung page proses

TLB REACH

- TLB Reach – Jumlah memory yang dapat diakses melalui TLB
- TLB Reach =(jumlah entri TLB) X (Ukuran Page)
- Idealnya *working set* setiap proses disimpan dalam TLB
  - Jika tidak, frekuensi akses ke page table lebih tinggi dibanding ke TLB

TLB REACH

- Cara meningkatkan TLBReach :
  - Menambah ukuran page
    - Fragmentasi bertambah.
    - Beberapa aplikasi tidak membutuhkan ukuran page yang besar
  - Menggunakan ukuran page yang bervariasi (*multiple page size*)
    - Aplikasi yang membutuhkan ukuran page lebih besar dapat menggunakan tanpa menambah fragmentasi

STRUKTUR PROGRAM

- Struktur Program *by example*
  - `Int[128,128] data;`
  - Jumlah frame yang tersedia adalah 1 frame
  - Satu baris matriks disimpan pada satu page
  - Program 1

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

128 x 128 = 16,384 page fault

  - Program 2

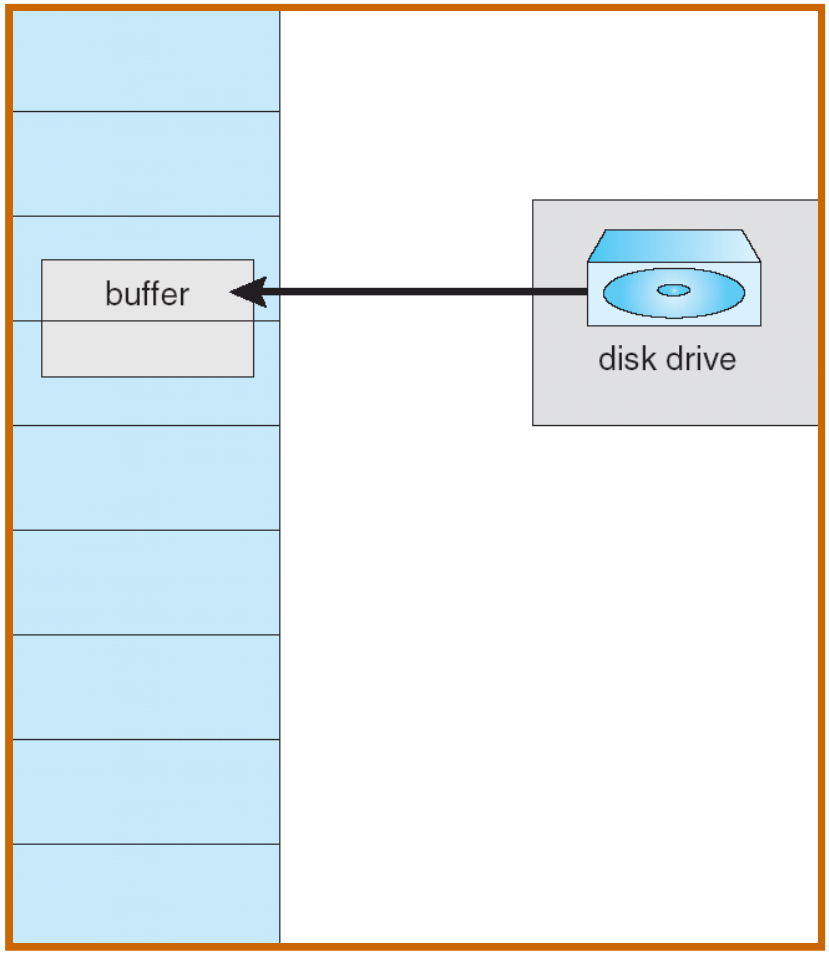
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

I/O INTERLOCK

- **I/O Interlock** – Kadang page harus di *lock* didalam memory dan tidak boleh diswap.
- Kenapa?
  - Salah satu alasan: page sedang digunakan sebagai buffer

I/O INTERLOCK



Selesai