R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه شانزدهم – الگوریتم‌های جایگزینی صفحه (۱)

جلسه‌ی گذشته

مدیریت حافظه – حافظه‌ی مجازی

Virtual Memory

- **Virtual memory** – separation of user logical memory from physical memory
  - *Only part of the program needs to be in memory for execution*
  - *Logical address space can therefore be much larger than physical address space*
  - *Allows address spaces to be shared by several processes*
  - *Allows for more efficient process creation*
  - *More programs running concurrently*
  - *Less I/O needed to load or swap processes*

Page Sharing in Fork System Call

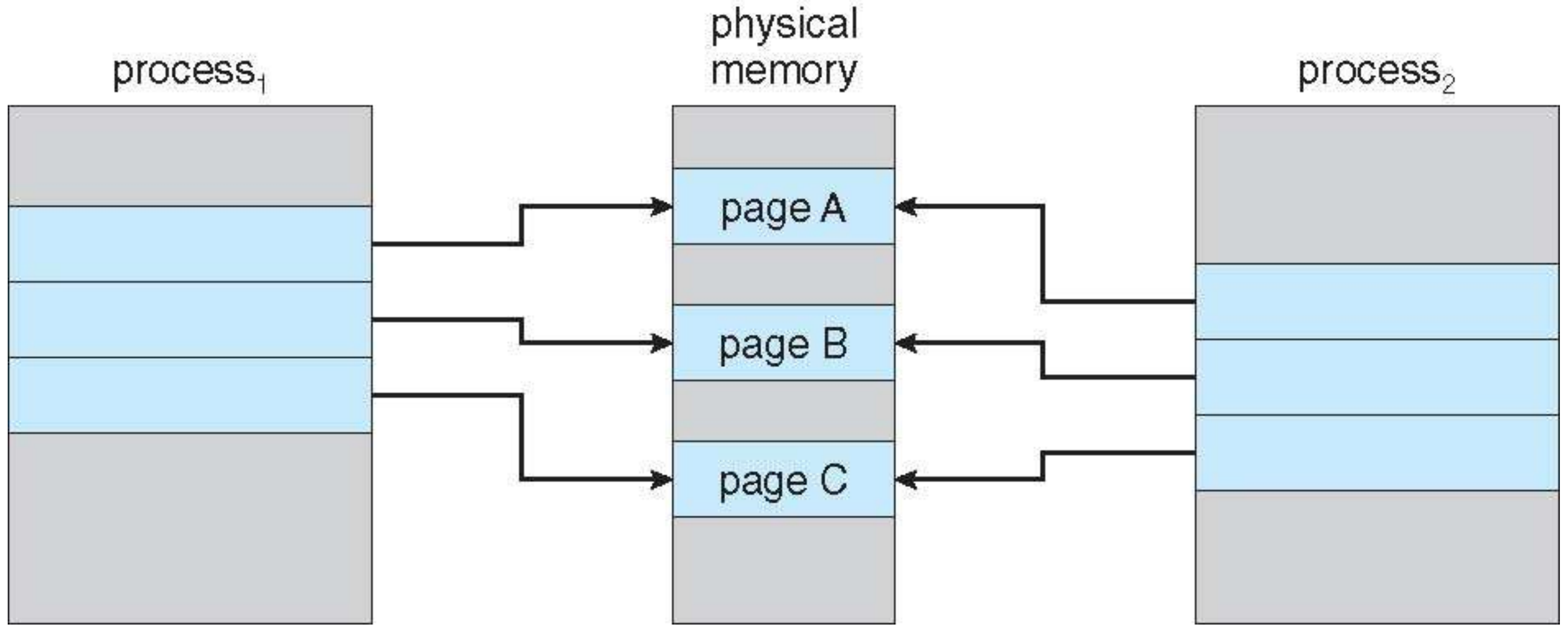
- Normal usage: copy the parent’s virtual address space and immediately do an “Exec” system call
  - *Exec overwrites the calling address space with the contents of an executable file (ie a new program)*
- Desired Semantics:
  - *Pages are copied, not shared*
- Observations
  - *Copying every page in an address space is expensive!*
  - *Processes can’t notice the difference between copying and sharing unless pages are modified!*

Copy-on-Write Page Sharing

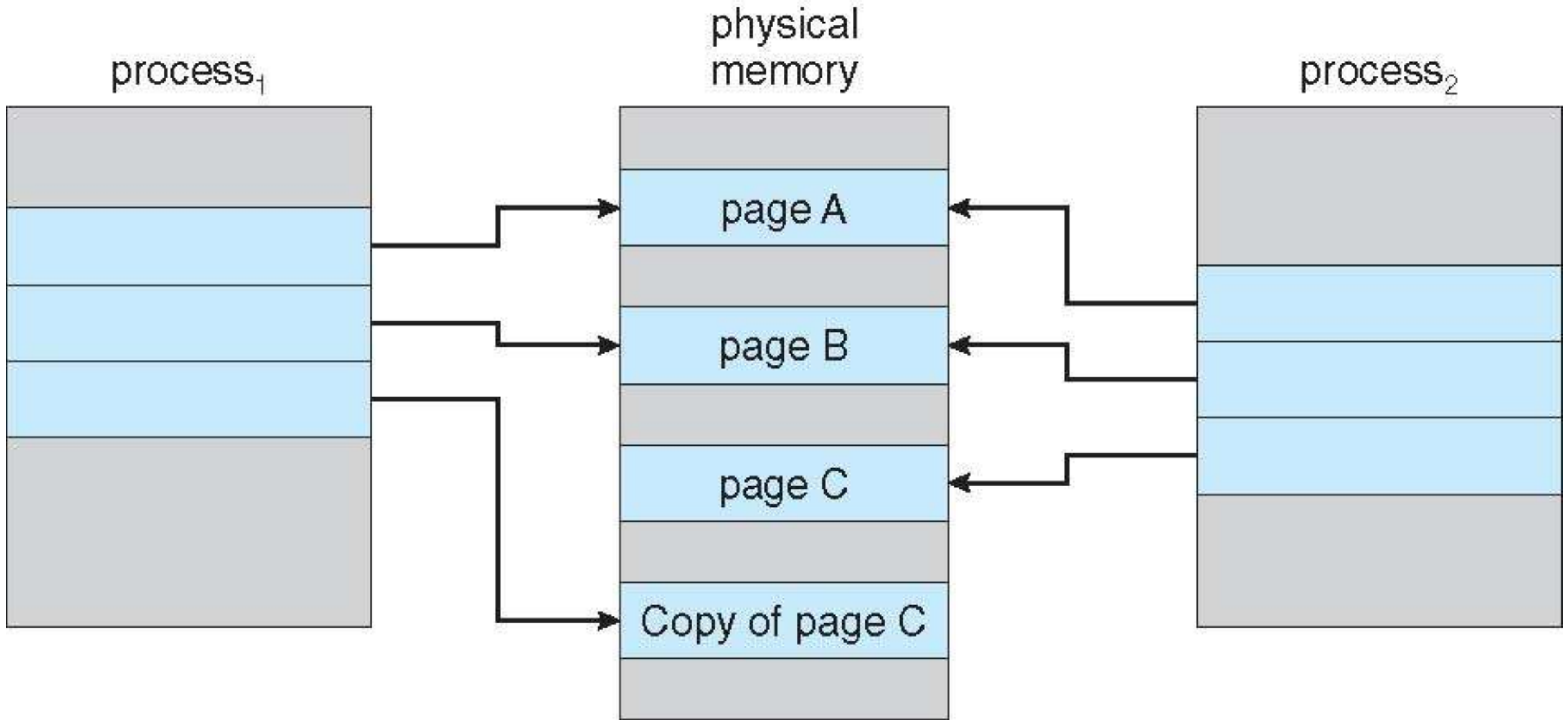
- Initialize new page table, but point entries to existing page frames of parent, i.e. share pages
- Temporarily mark all pages "read-only"
- Continue to share all pages until a protection fault occurs
- Protection fault (copy-on-write fault):
  - *Is this page really read only or is it writable but temporarily protected for copy-on-write?*
  - *If it is writable, copy the page, ~~mark both copies writable~~, resume execution as if no fault occurred*

فقط copy مخصوص پردازه‌ای که تلاش کرده بنویسه writable می‌شود.
همچنین سیستم عامل در یک داده‌ساختار مجزا خارج از page table اطلاعات مختص به page sharing را نگه می‌دارد،
در این داده‌ساختار یک شمارنده که هر صفحه‌ی اشتراک گذاشته‌شده، متعلق به چند پردازه است نیز وجود دارد و اگر
این عدد ۱ بود، صفحه را کپی نمی‌کند.

Before Process 1 Modifies Page C



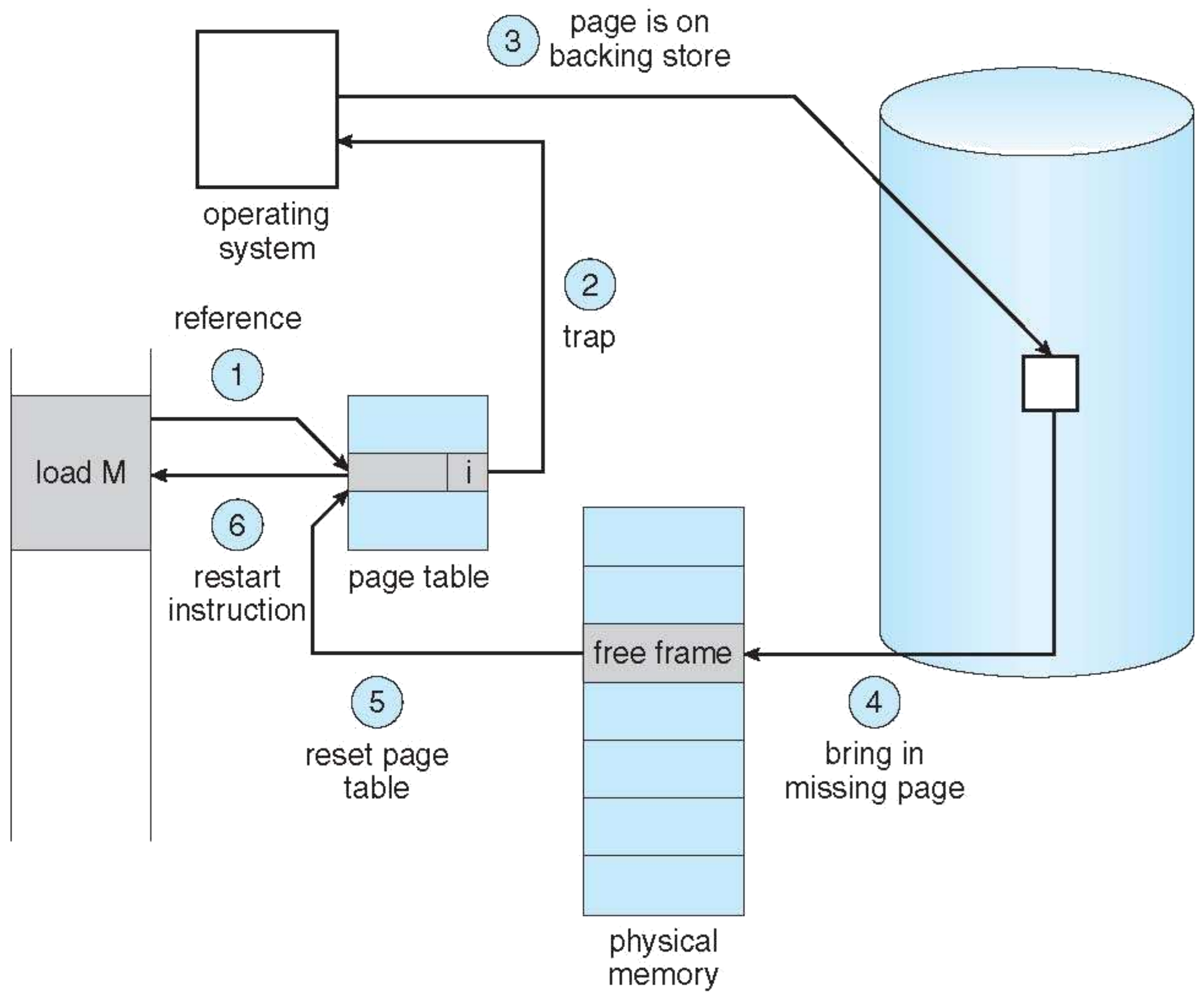
After Process 1 Modifies Page C



Virtual Memory Implementation

- When is the kernel involved?
  - *Process creation*
  - *Process is scheduled to run*
  - *A fault occurs*
  - *Process termination*

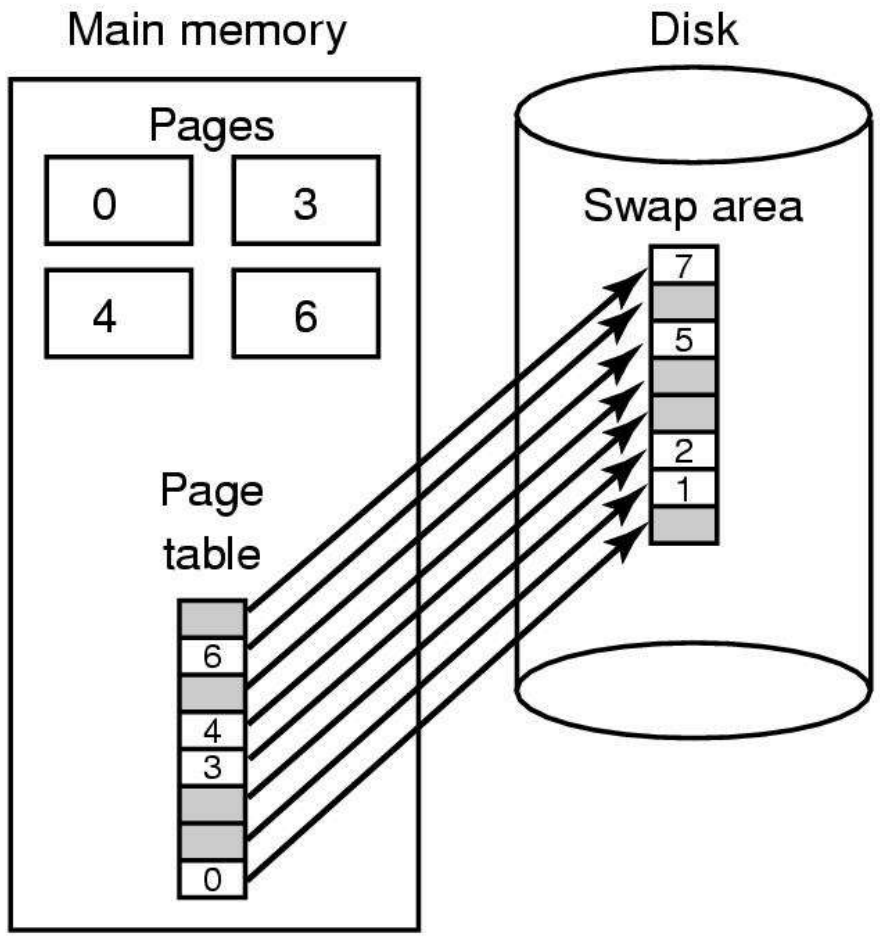
Handling a Page Fault



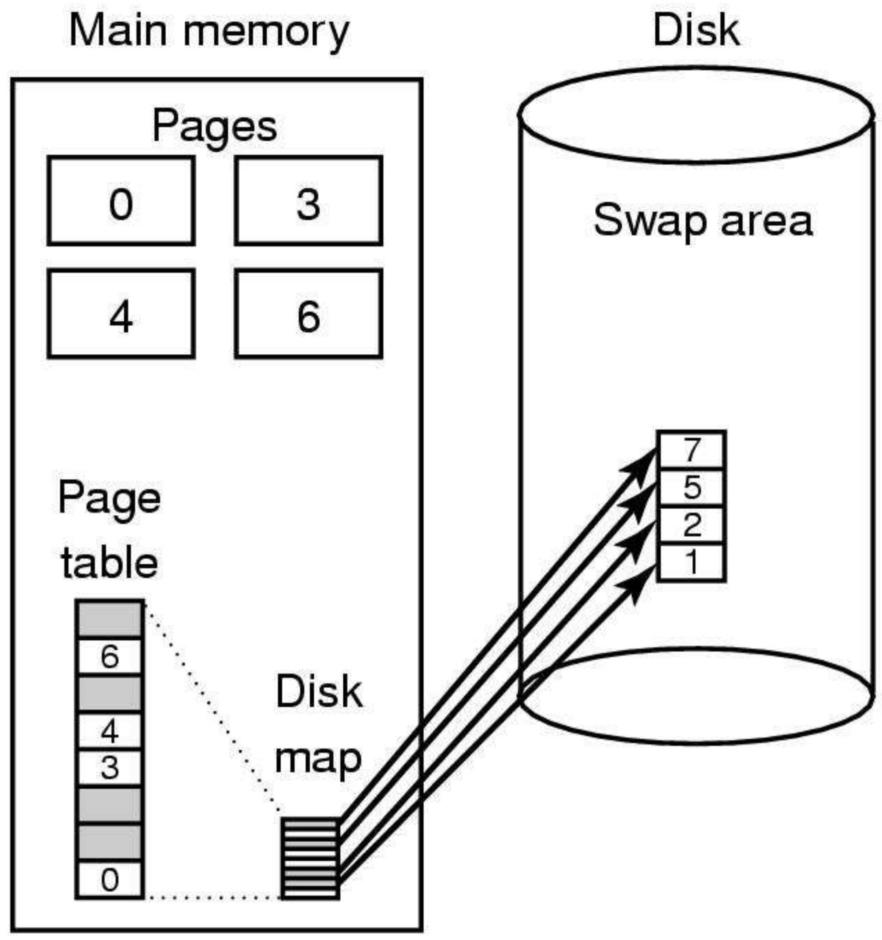
Managing the Swap Area on Disk

- ***Approach #1:***
- A process starts up
  - *Assume it has N pages in its virtual address space*
- A region of the swap area is set aside for the pages
- There are N pages in the swap region
- The pages are kept in order
- For each process, we need to know:
  - *Disk address of page 0*
  - *Number of pages in address space*
- Each page is either...
  - *In a memory frame*
  - *Stored on disk*

Approach #2



(a)



(b)

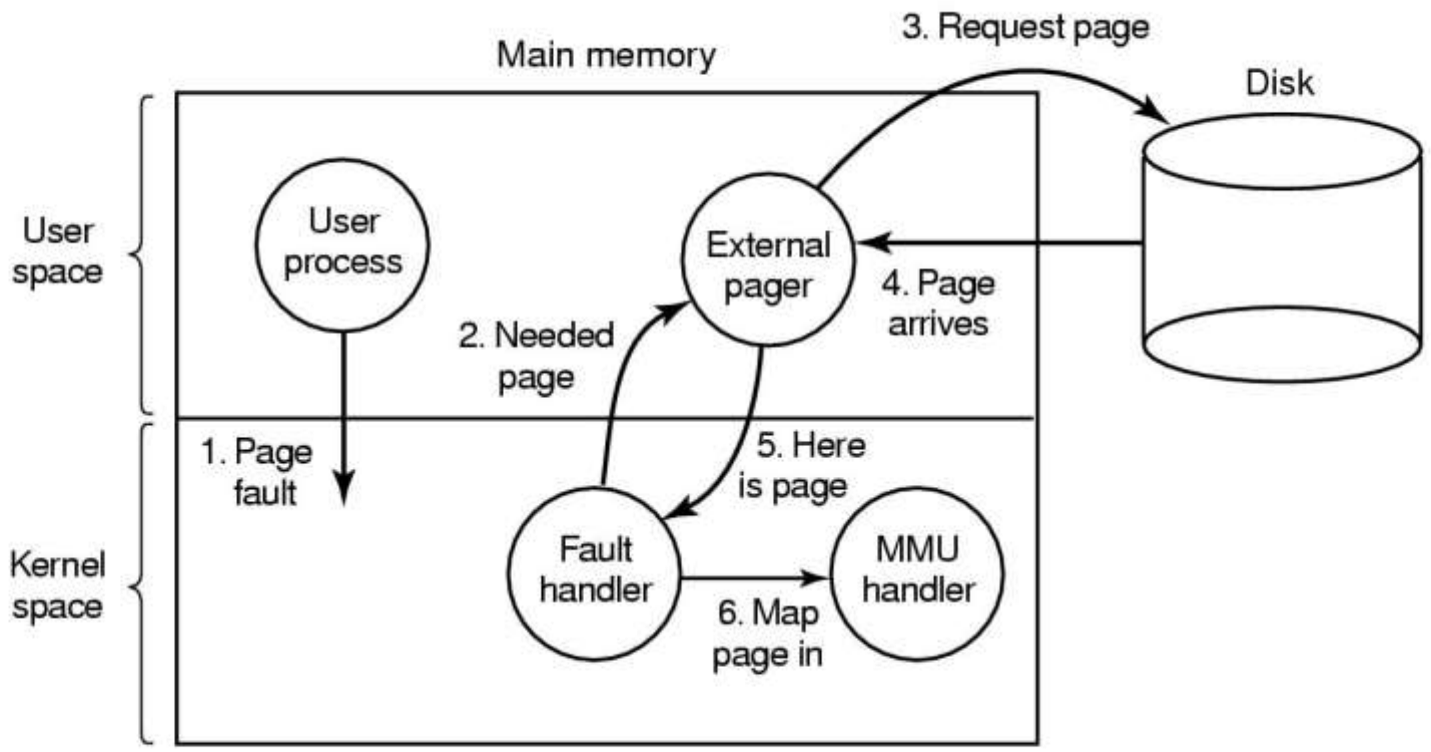
Approach #3

- Swap to a file
  - *Each process has its own swap file*
  - *File system manages disk layout of files*

Approach #4

- Swap to an external pager process (object)
- A user-level external pager determines policy
  - *Which page to evict* α
  - *When to perform disk I/O*
  - *How to manage the swap file*
- When the OS needs to read in or write out a page it sends a message to the external pager
  - *Which may even reside on a different machine*

Approach #4


Main memory
3. Request page
Disk
User space
Kernel space
User process
External pager
4. Page arrives
2. Needed page
1. Page fault
5. Here is page
Fault handler
MMU handler
6. Map page in

Mechanism vs Policy

- Kernel contains
  - *Code to interact with the MMU*
    - This code tends to be *machine dependent*
    - *Mechanism*
  - *Code to handle page faults*
    - This code tends to be *machine independent* and may embody generic operating system policies
    - Policy

جلسه‌ی جدید

Page replacement algorithms

کارایی صفحەبندی

Paging Performance

- Paging works best if there are plenty of free frames
- If all pages are full of dirty pages…
  - *we must perform 2 disk operations for each page fault*
  - *This doubles page fault latency*
- It can be a good idea to periodically write out dirty pages in order to speed up page fault handling delay

Paging Daemon

- Paging daemon
  - *A kernel process*
  - *Wakes up periodically*
  - *Counts the number of free page frames*
  - *If too few, run the page replacement algorithm...*
    - Select a page & write it to disk
    - Mark the page as clean
  - *If this page is needed later... then it is still there*
  - *If an empty frame is needed then this page is evicted*

جایگزینی صفحه

Page Replacement

- Assume a normal page table (e.g., BLITZ)
- User-program is executing
- A *PageInvalidFault* occurs!
  - *The page needed is not in memory*
- Select some frame and remove the page in it
  - *If it has been modified, it must be written back to disk*
    - the “dirty” bit in its page table entry tells us if this is necessary
- Figure out which page was needed from the faulting addr
- Read the needed page into this frame
- Restart the interrupted process by retrying the same instruction

Page Replacement Algorithms

- Which frame to replace?
- *Algorithms?*
  - ورودی: لیست صفحه‌هایی که در حافظه هستند.
  - خروجی: کدام صفحه را از حافظه خارج کنیم
  - هدف: کاهش تعداد page faultها

الگوریتمی با کمینه تعداد PAGE FAULT!

The optimal page replacement algorithm

- **Idea:** Given all the data, how to find the optimal page replacement?

```
Time           0    1    2    3    4    5    6    7    8    9    10
Requests            c    a    d    b    e    b    a    b    c    d
_____________________________________________________________________
Page    0      a
Frames  1      b
        2      c
        3      d

Page faults
```

The optimal page replacement algorithm

- **Idea:** Given all the data, how to find the optimal page replacement?

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page	0	a	a	a	a	a						
Frames	1	b	b	b	b	b						
	2	c	c	c	c	c						
	3	d	d	d	d	d						
Page faults							X					

The optimal page replacement algorithm

- **Idea:** Given all the data, how to find the optimal page replacement?
- **Longest Forward Distance (LFD):** Select the page that will not be needed for the longest time

LFD

- Replace the page that will not be needed for the longest
- Example:

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page	0	a	a	a	a	a						
Frames	1	b	b	b	b	b						
	2	c	c	c	c	c						
	3	d	d	d	d	d						
Page faults							X					

LFD

- Replace the page that will not be needed for the longest
- Example:

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page	0	a	a	a	a	a	a	a	a	a	a	
Frames	1	b	b	b	b	b	b	b	b	b	b	
	2	c	c	c	c	c	c	c	c	c	c	
	3	d	d	d	d	d	e	e	e	e	e	
Page faults							X				X	

Is LFD Optimal?

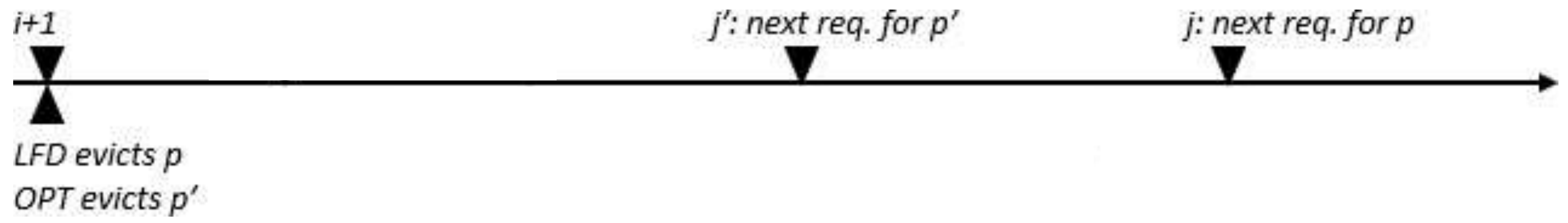
- Try to prove by contradiction…

Is LFD Optimal?

- **OPT:** Optimum with longest prefix equal to LFD
  - *First non-equal poistion: i+1*
- Case 1) i+1 is not a page fault.
  - *Is it possible?*
- Case 2) i+1 is page fault
  - *LFD evicts p*
  - *OPT evicts p'*

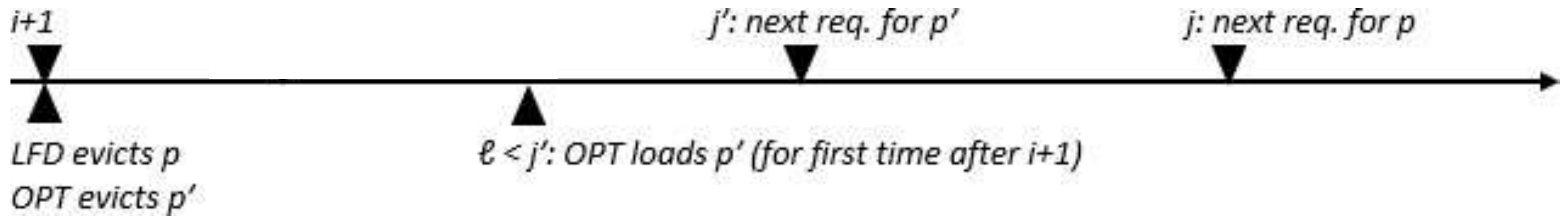
Is LFD Optimal?

- **OPT:** Optimum with longest prefix equal to LFD
  - *First non-equal poistion: i+1*
- Case 2) i+1 is page fault
  - *LFD evicts p*
  - *OPT evicts p'*



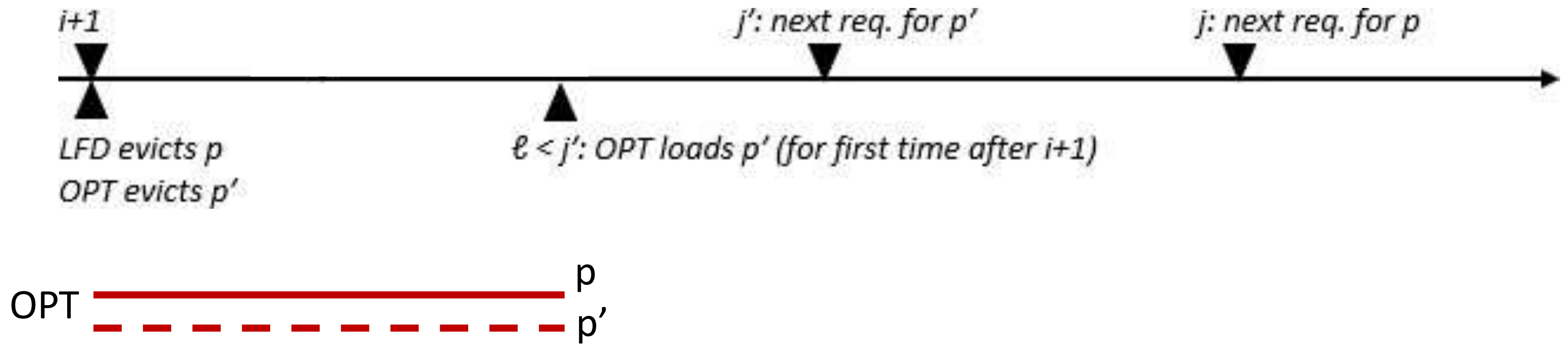
Is LFD Optimal?

- Proof (by contradiction):
  - **OPT**: Optimum with longest prefix equal to LFD Case
  - 2) i+1 is page fault
    - Case 2-A) OPT keeps p until l
    - Case 2-B) OPT evicts p at l' < l



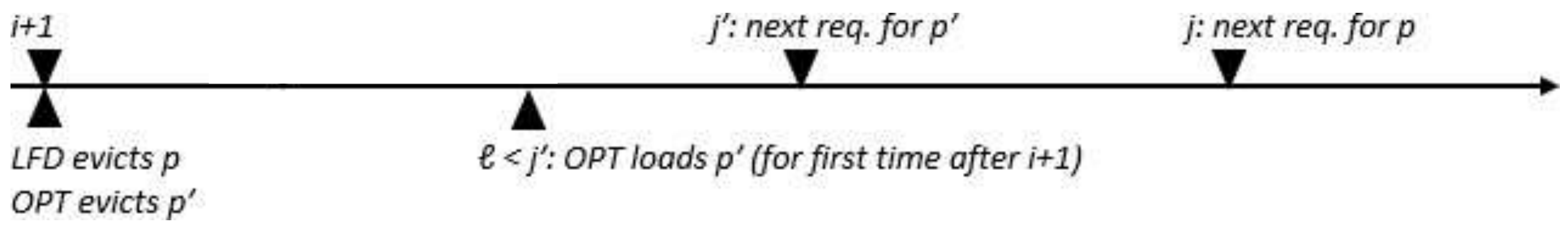
Is LFD Optimal?

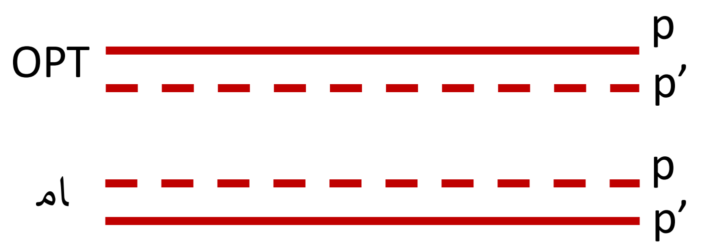
- Proof (by contradiction):
  - **OPT**: Optimum with longest prefix equal to LFD Case
  - 2) i+1 is page fault
    - Case 2-A) OPT keeps p until l
    - Case 2-B) OPT evicts p at l' < l



Is LFD Optimal?

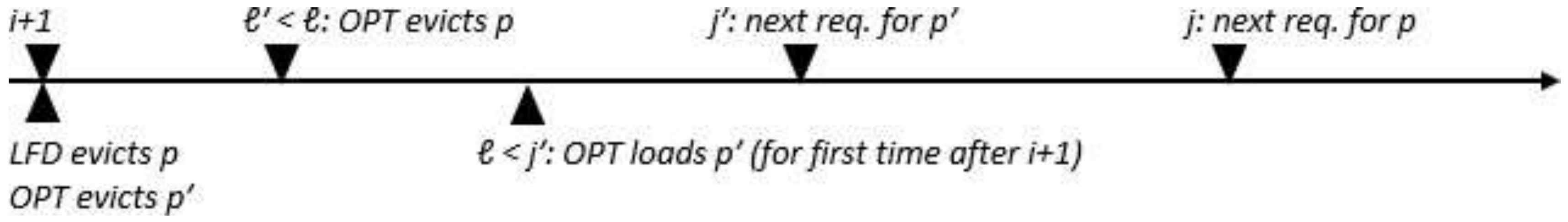
- Proof (by contradiction):
  - **OPT**: Optimum with longest prefix equal to LFD Case
  - 2) i+1 is page fault
    - Case 2-A) OPT keeps p until l
    - Case 2-B) OPT evicts p at l' < l





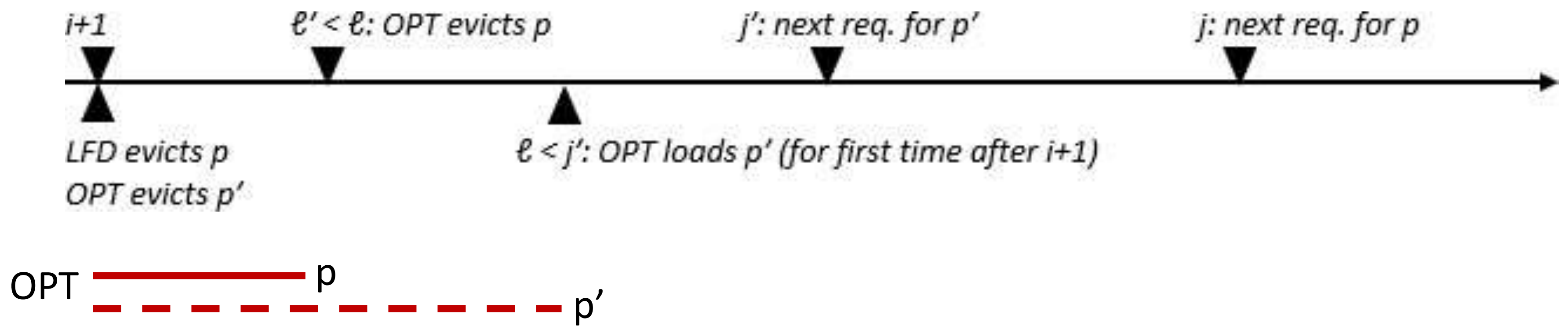
Is LFD Optimal?

- Proof (by contradiction):
  - **OPT**: Optimum with longest prefix equal to LFD Case
  - 2) i+1 is page fault
    - Case 2-A) OPT keeps p until l
    - Case 2-B) OPT evicts p at l’ < l



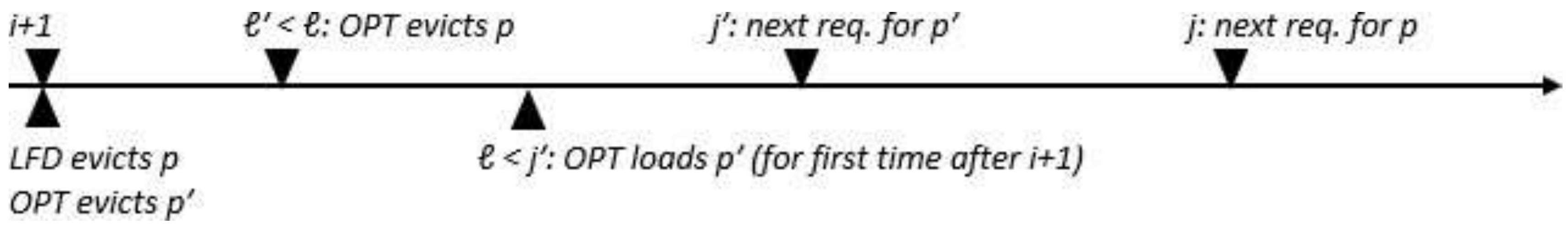
Is LFD Optimal?

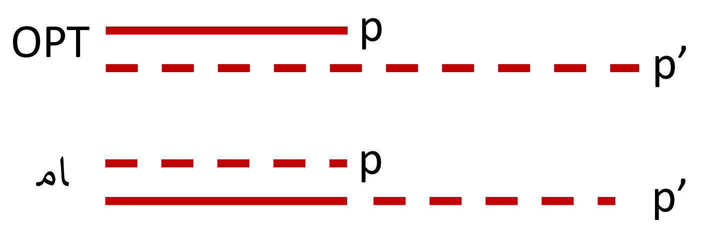
- Proof (by contradiction):
  - **OPT**: Optimum with longest prefix equal to LFD Case
  - 2) i+1 is page fault
    - Case 2-A) OPT keeps p until l
    - Case 2-B) OPT evicts p at l' < l



Is LFD Optimal?

- Proof (by contradiction):
  - **OPT**: Optimum with longest prefix equal to LFD Case
  - 2) i+1 is page fault
    - Case 2-A) OPT keeps p until l
    - Case 2-B) OPT evicts p at l’ < l





LFD = OPT

Optimal Page Replacement

- Idea:
  - *Select the page that will not be needed for the longest time*
- Problem:
  - *Can't know the future of a program*
  - *Can't know when a given page will be needed next*
  - *The optimal algorithm is unrealizable*

Optimal Page Replacement

- However:
  - *We can use it as a control case for simulation studies*
    - Run the program once
    - Generate a log of all memory references
    - Do we need all of them?
    - Use the log to simulate various page replacement algorithms
    - Can compare others to “optimal” algorithm

FIFO ALGORITHM

FIFO Algorithm

- Always replace the oldest page …
  - *Replace the page that has been in memory for the longest time*

FIFO Algorithm

- Replace the page that was first brought into memory
- Example: Memory system with 4 frames:

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	a
Page	0	a		**a**	**a**	**a**						
Frames	1	b				**b**						
	2	c	**c**	**c**	**c**	**c**						
	3	d			**d**	**d**						
Page faults							**x**					

FIFO Algorithm

- Replace the page that was first brought into memory

- Example: Memory system with 4 frames:

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	a
Page	0	a		a	a	a	a	a	a	a		
Frames	1	b				b	b	b	b	b		
	2	c	c	c	c	c	e	e	e	e		
	3	d			d	d	d	d	d	d		
Page faults							X				X	

FIFO Algorithm

- Replace the page that was first brought into memory

- Example: Memory system with 4 frames:

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	a
Page	0	a		a	a	a	a	a	a	a	c	
Frames	1	b				b	b	b	b	b	b	
	2	c	c	c	c	c	e	e	e	e	e	
	3	d			d	d	d	d	d	d	d	
Page faults							x				x	x

FIFO Algorithm

- Replace the page that was first brought into memory

- Example: Memory system with 4 frames:

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	a
Page	0	a		a	a	a	a	a	a	a	c	c
Frames	1	b				b	b	b	b	b	b	b
	2	c	c	c	c	c	e	e	e	e	e	e
	3	d			d	d	d	d	d	d	d	a
Page faults							x				x	x

FIFO Algorithm

- Always replace the oldest page.
  - *Replace the page that has been in memory for the longest time*

- Implementation
  - *Maintain a linked list of all pages in memory*
  - *Keep it in order of when they came into memory*
  - *The page at the tail of the list is oldest*
  - *Add new page to head of list*

FIFO Algorithm

- Disadvantage?
  - *The oldest page may be needed again soon*
  - *Some page may be important throughout execution*
  - *It will get old, but replacing it will cause an immediate page fault*

NRU ALGORITHM

How Can We Do Better?

- Need an approximation of how likely each frame is to be accessed in the future
  - *If we base this on past behavior we need a way to track past behavior*
  - *Tracking memory accesses requires hardware support to be efficient*

Referenced and Dirty Bits

- Each page table entry (and TLB entry!) has a
  - *Referenced bit - set by TLB when page read / written*
  - *Dirty / modified bit - set when page is written*
  - *If TLB entry for this page is valid, it has the most up to date version of these bits for the page*
  - *OS must copy them into the page table entry during fault handling*
- Idea: use the information contained in these bits to drive the page replacement algorithm

Referenced and Dirty Bits

- Some hardware does not have support for the dirty bit
- Instead, memory protection can be used to emulate it
- Idea:
  - *Software sets the protection bits for all pages to “read only”*
  - *When program tries to update the page...*
    - A trap occurs
    - Software sets the Dirty Bit in the page table and clears the ReadOnly bit
    - Resumes execution of the program

Not Recently Used Algorithm

- Uses the Referenced Bit and the Dirty Bit
- Initially, all pages have
  - *Referenced Bit = 0*
  - *Dirty Bit = 0*
- Periodically... (e.g. whenever a timer interrupt occurs)
  - *Clear the Referenced Bit*
  - *Referenced bit now indicates "recent" access*

Not Recently Used Algorithm

- When a page fault occurs...
- Categorize each page...

Class 1:	*Referenced = 0*	*Dirty = 0*
Class 2:	*Referenced = 0*	*Dirty = 1*
Class 3:	*Referenced = 1*	*Dirty = 0*
Class 4:	*Referenced = 1*	*Dirty = 1*

- Choose a victim page from …
  - *class 1 … why?*
- If none, choose a page from …
  - *class 2 … why?*
- If none, choose a page from …
  - *class 3 … why?*
- If none, choose a page from …
  - *class 4 … why?*

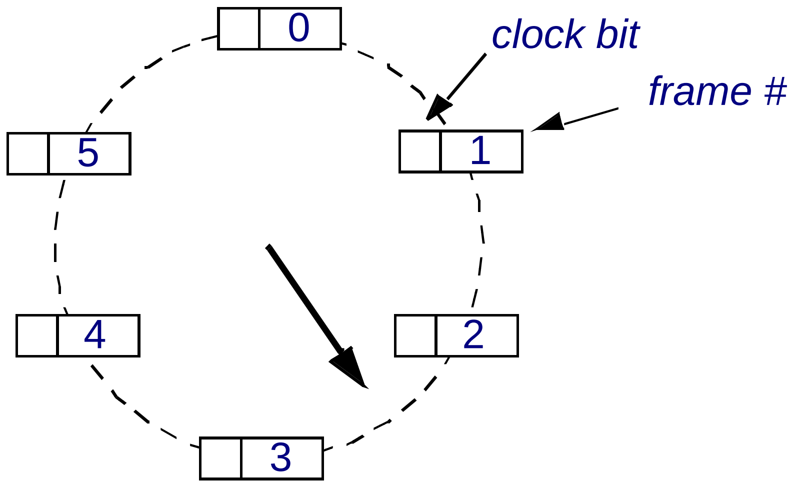
SECOND CHANCE ALGORITHM

Second Chance Algorithm

- An implementation of NRU based on FIFO
- Pages kept in a linked list (oldest at the front)
- Look at the oldest page
  - *If its "referenced bit" is 0...*
    - Select it for replacement
  - *Else*
    - It was used recently; don't want to replace it
    - Clear its "referenced bit"
    - Move it to the end of the list
  - *Repeat*
- What if every page was used in last clock tick?

Implementation of Second Chance

- Maintain a circular list of pages in memory
- Set a bit for the page when a page is referenced
- Search list looking for a victim page that does not have the referenced bit set
  - *If the bit is set, clear it and move on to the next page*
  - *Replaces pages that haven't been referenced for one complete clock revolution*



LRU

Least Recently Used Algorithm

- A refinement of NRU that orders how recently a page was used
  - *Keep track of when a page is used*
  - *Replace the page that has been used least recently*

Least Recently Used Algorithm

- Replace the page that hasn’t been referenced in the longest time

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page	0	a	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**a**
Frames	1	b	**b**	**b**	**b**	**b**	**b**	**b**	**b**	**b**	**b**	**b**
	2	c	**c**	**c**	**c**	**c**	**e**	**e**	**e**	**e**	**e**	**d**
	3	d	**d**	**d**	**d**	**d**	**d**	**d**	**d**	**d**	**c**	**c**
Page faults							**x**				**x**	**x**

Least Recently Used Algorithm

- But how can we implement LRU?

Least Recently Used Algorithm

- But how can we implement LRU?

- Idea #1:
  - *Keep a linked list of all pages*
  - *On every memory reference, Move that page to the front of the list*
  - *The page at the tail of the list is replaced*

Least Recently Used Algorithm

- But how can we implement LRU?
  - *… without requiring every access to be recorded?*
- Idea #2:
  - *MMU (hardware) maintains a counter*
  - *Incremented on every clock cycle*
  - *Every time a page table entry is used*
    - MMU writes the value to the page table entry
    - This *timestamp* value is the *time-of-last-use*
  - *When a page fault occurs*
    - OS looks through the page table
    - Identifies the entry with the oldest timestamp

Least Recently Used Algorithm

- What if we don't have hardware support for a counter?
- Idea #3:
  - *Maintain a counter in software*
  - *One every timer interrupt...*
    - Increment counter
    - Run through the page table
    - For every entry that has "ReferencedBit" = 1
      - ** Update its timestamp*
      - ** Clear the ReferencedBit*
  - *Approximates LRU*
  - *If several have oldest time, choose one arbitrarily*