INFO324 Operating Systems II

Ahmad FAOUR

Page Replacement

Page Replacement

- Assume a normal page table
- 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
- Local allocation or global allocation?
  - Limit the search to the pages loaded for the process causing the default or all loaded pages
- 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

Terminology

- **Reference string**: the memory reference sequence generated by a program.
- **Paging** – moving pages to (from) disk
- **Optimal** – the best (theoretical) strategy
- **Eviction** – throwing something out
- **Pollution** – bringing in useless pages/lines

Page Replacement Algorithms

Which frame to replace?

Algorithms:

- The Random Algorithm
- The Optimal Algorithm
- First In First Out (FIFO)
- Not Recently Used (NRU)
- Second Chance / Clock
- Least Recently Used (LRU)
- Not Frequently Used (NFU)
- Working Set (WS)
- WSClock

The Random Algorithm

- Choose a page randomly
- Very easy
- Local and global version
- Used for comparison studies

The Optimal Algorithm

Idea:

Select the page that will not be needed for the longest time

Principal of Optimality

- Description:
  - Assume that each page can be labeled with the number of instructions that will be executed before that page is first references, i.e., we would know the future reference string for a program.
  - Then the optimal page algorithm would choose the page with the highest label to be removed from the memory.
- This algorithm provides a basis for comparison with other schemes.
- Impractical because it needs future references
- If future references are known
  - should not use demand paging
  - should use pre paging to allow paging to be overlapped with computation.

Optimal Page Replacement

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 Frames	0	a	**a**	**a**	**a**	**a**						
	1	b	**b**	**b**	**b**	**b**						
	2	c	**c**	**c**	**c**	**c**						
	3	d	**d**	**d**	**d**	**d**						
Page faults							**x**					

Optimal Page Replacement

Select the page that will not be needed for the longest time

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 Frames	0	a	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**a**	
	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**

Optimal Example (2)

12 references,
7 faults

Page Refs	3 Page Frames			
	Fault?	Page Contents		
A	yes	A		
B	yes	B	A	
C	yes	C	B	A
D	yes	D	B	A
A	no	D	B	A
B	no	D	B	A
E	yes	E	B	A
A	no	E	B	A
B	no	E	B	A
C	yes	C	E	B
D	yes	D	C	E
E	no	D	C	E

Optimal Page Replacement

Idea:

Select the page that will not be needed for the longest time

Problem?

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

Always replace the oldest page …

- *Replace the page that has been in memory for the longest time ($t_{loading}$)*

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 Frames	0	a		**a**	**a**	**a**						
	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 Frames	0	a		**a**	**a**	**a**	**a**	**a**	**a**	**a**		
	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 Frames	0	a		**a**	**a**	**a**	**a**	**a**	**a**	**a**	**c**	
	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 Frames	0	a		**a**	**a**	**a**	**a**	**a**	**a**	**a**	**c**	**c**
	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

12 references,
9 faults

Page Refs	3 Page Frames			
	Fault?	Page Contents		
A	yes	A		
B	yes	B	A	
C	yes	C	B	A
D	yes	D	C	B
A	yes	A	D	C
B	yes	B	A	D
E	yes	E	B	A
A	no	E	B	A
B	no	E	B	A
C	yes	C	E	B
D	yes	D	C	E
E	no	D	C	E

Belady's Anomaly (for FIFO)

As the number of page frames increase, so does the fault rate.

12 references, 10 faults

Page Refs	4 Page Frames				
	Fault?	Page Contents			
A	yes	A			
B	yes	B	A		
C	yes	C	B	A	
D	yes	D	C	B	A
A	no	D	C	B	A
B	no	D	C	B	A
E	yes	E	D	C	B
A	yes	A	E	D	C
B	yes	B	A	E	D
C	yes	C	B	A	E
D	yes	D	C	B	A
E	yes	E	D	C	B

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?

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

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

Not Recently Used Algorithm (NRU)

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

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?

Second Chance Example

12 references,
9 faults

Page Refs	3 Page Frames			
	Fault?	Page Contents		
A	yes	A*		
B	yes	B*	A*	
C	yes	C*	B*	A*
D	yes	D*	C	B
A	yes	A*	D*	C
B	yes	B*	A*	D*
E	yes	E*	B	A
A	no	E*	B	A*
B	no	E*	B*	A*
C	yes	C*	E	B
D	yes	D*	C*	E
E	no	D*	C*	E*

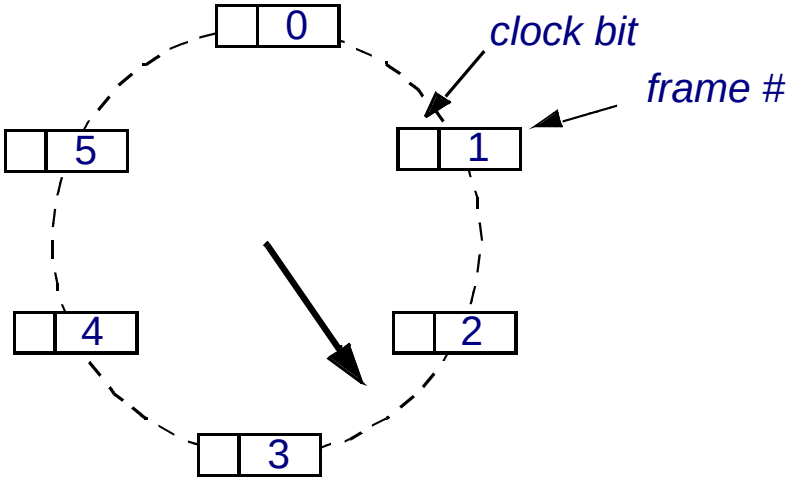
Implementation of Second Chance

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

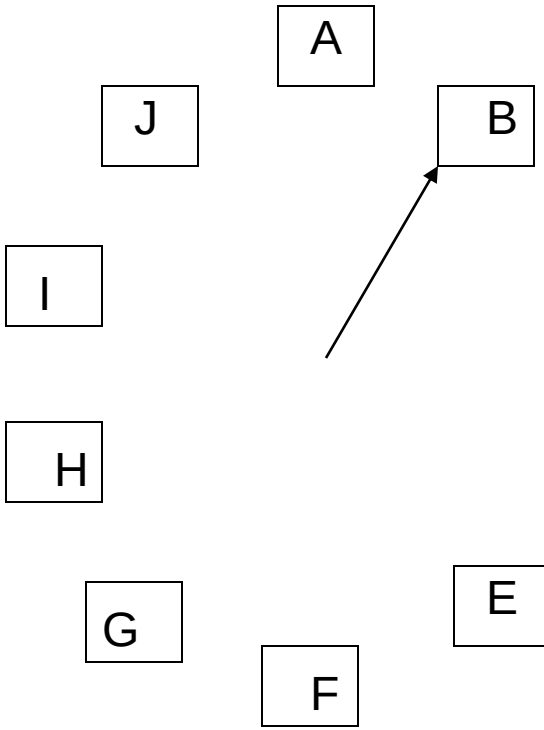
- If the bit is set, clear it and move on to the next page
- Replaces pages that haven't been referenced for one complete clock revolution



Clock Page Replacement Algorithm

Similar to second chance, however

Keep all page frames on a circular list in the form of a clock



Check the R Bit,
if R= 0, evict the page,
if R=1, clear the bit and advance

Least Recently Used Algorithm

A refinement of NRU that orders how recently a page was used

- Keep track of when a page is used ($t_{reference}$)
- 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 Frames	0	a	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**a**
	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**

LRU Example 2

12 references,
10 faults

Page Refs	3 Page Frames			
	Fault?	Page Contents		
A	yes	A		
B	yes	B	A	
C	yes	C	B	A
D	yes	D	C	B
A	yes	A	D	C
B	yes	B	A	D
E	yes	E	B	A
A	no	A	E	B
B	no	B	A	E
C	yes	C	B	A
D	yes	D	C	B
E	yes	E	D	C

Least Recently Used Issues

- Does not suffer from Belady's anomaly
- How to track “recency”?
  - use time
    - record time of reference with page table entry
    - use counter as clock
    - search for smallest time.
  - use stack
    - remove reference of page from stack (linked list)
    - push it on top of stack
- both approaches require large processing overhead, more space, and hardware support.

LRU and Anomalies

12 references,
8 faults

Page Refs	4 Page Frames				
	Fault?	Page Contents			
A	yes	A			
B	yes	B	A		
C	yes	C	B	A	
D	yes	D	C	B	A
A	no	A	D	C	B
B	no	B	A	D	C
E	yes	E	B	A	D
A	no	A	E	B	D
B	no	B	A	E	D
C	yes	C	B	A	E
D	yes	D	C	B	A
E	yes	E	D	C	B

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

Not Frequently Used Algorithm

- Bases decision of frequency of use rather than recency
- Associate a counter with each page
- On every clock interrupt, the OS looks at each page.
  - If the *reference bit* is set increment that page's counter & clear the bit
- The counter approximates how often the page is used
- For replacement, choose the page with lowest counter

Not Frequently Used Algorithm

Problem:

Some page may be heavily used

- Its counter is large

The program's behavior changes

- Now, this page is not used ever again (or only rarely)

This algorithm never forgets!

- *This page will never be chosen for replacement!*

We may want to combine frequency and recency somehow

NFU With Aging

Associate a counter with each page

On every clock tick, the OS looks at each page.

- Shift the counter right 1 bit (divide its value by 2)
- If the *reference bit* is set...
  - Set the most-significant bit
  - Clear the Referenced Bit

T_1 100000 = 32
T_2 010000 = 16
T_3 001000 = 8
T_4 000100 = 4
T_5 100010 = 34

The Working Set

Demand paging

- Pages are only loaded when accessed
- When process begins, all pages marked INVALID

The Working Set

Demand paging

- Pages are only loaded when accessed
- When process begins, all pages marked INVALID

Locality of reference

- Processes tend to use only a small fraction of their pages

The Working Set

Demand paging

- Pages are only loaded when accessed
- When process begins, all pages marked INVALID

Locality of reference

- Processes tend to use only a small fraction of their pages

Working Set

- The set of pages a process needs
- If working set is in memory, no page faults
- What if you can't get working set into memory?

The Working Set

Thrashing

- If you can't get working set into memory page faults occur every few instructions
- Little work gets done
- Most of the CPU's time is going on overhead

Working Set Algorithm

Based on prepaging (prefetching)

- Load pages before they are needed

Main idea:

- Try to identify the process's working set based on time
- Keep track of each page's time since last access
- Assume working set valid for T time units
- Replace pages older than T

Working Set Algorithm

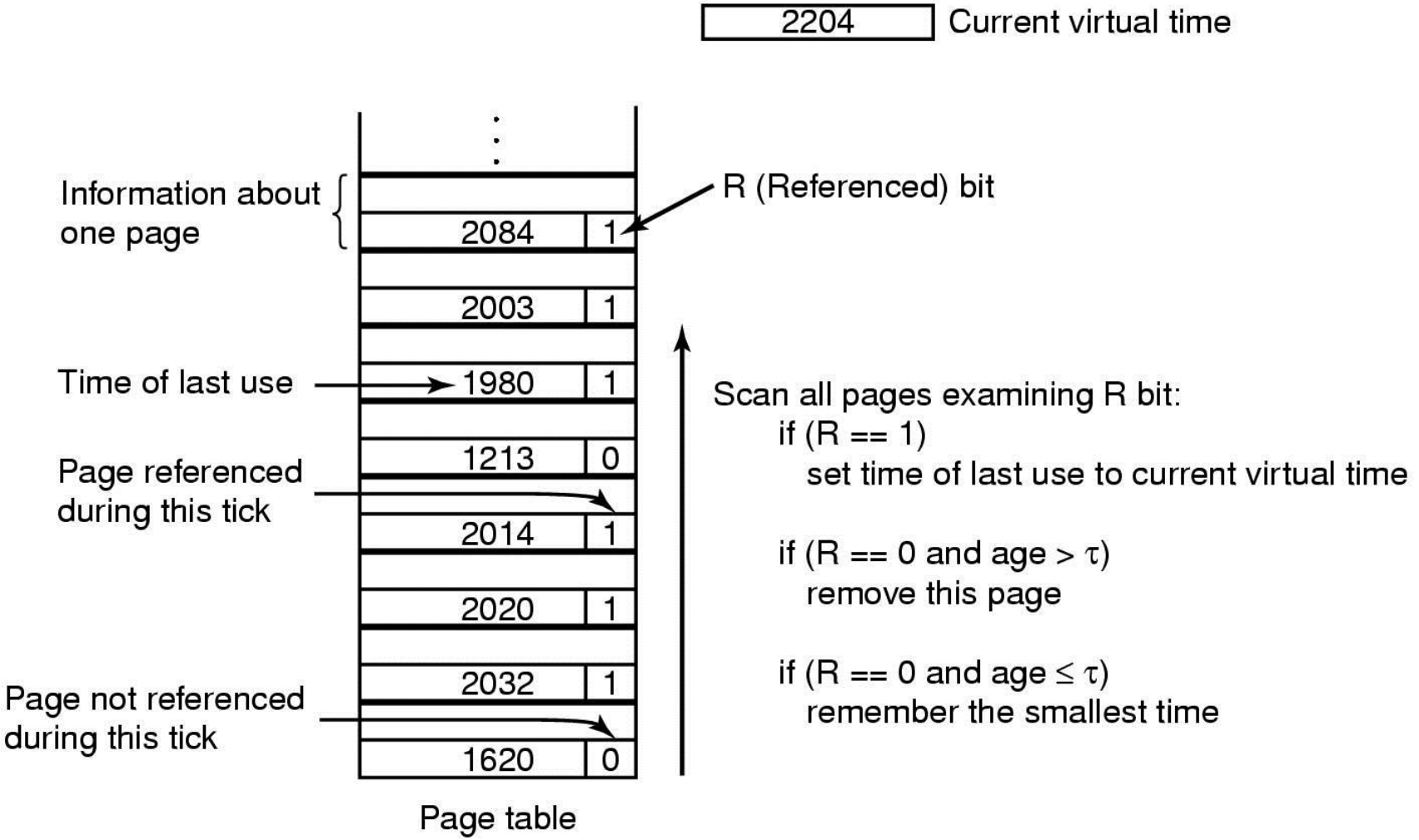
Current Virtual Time

- Only consider how much CPU time this process has seen

Implementation

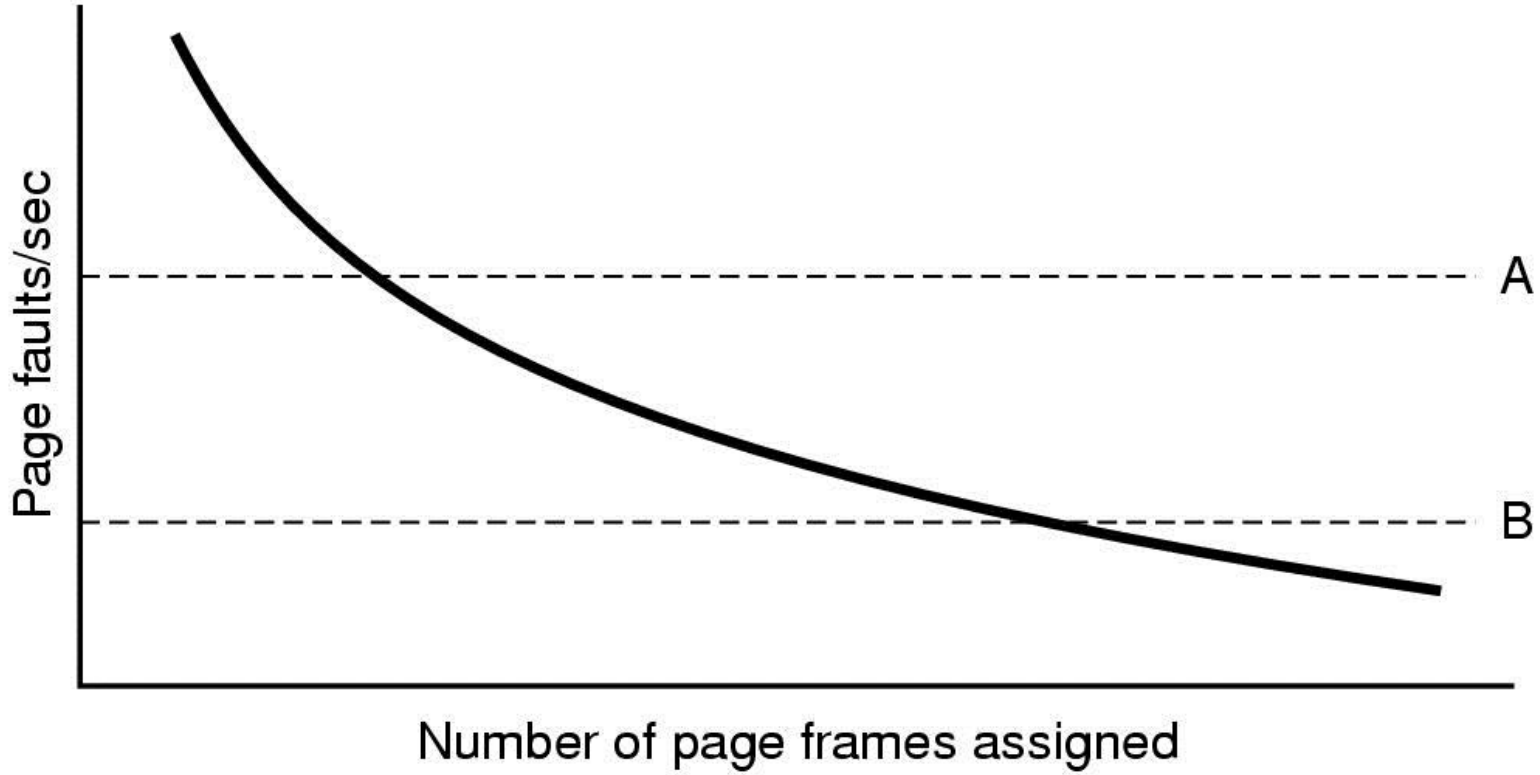
- On each clock tick, look at each page
- Was it referenced since the last check?
  - Yes: make a note of Current Virtual Time
- If a page has not been used in the last T msec,
  - Assume it is not in the working set!
  - Evict it
  - Write it out if it is dirty

Working Set Algorithm

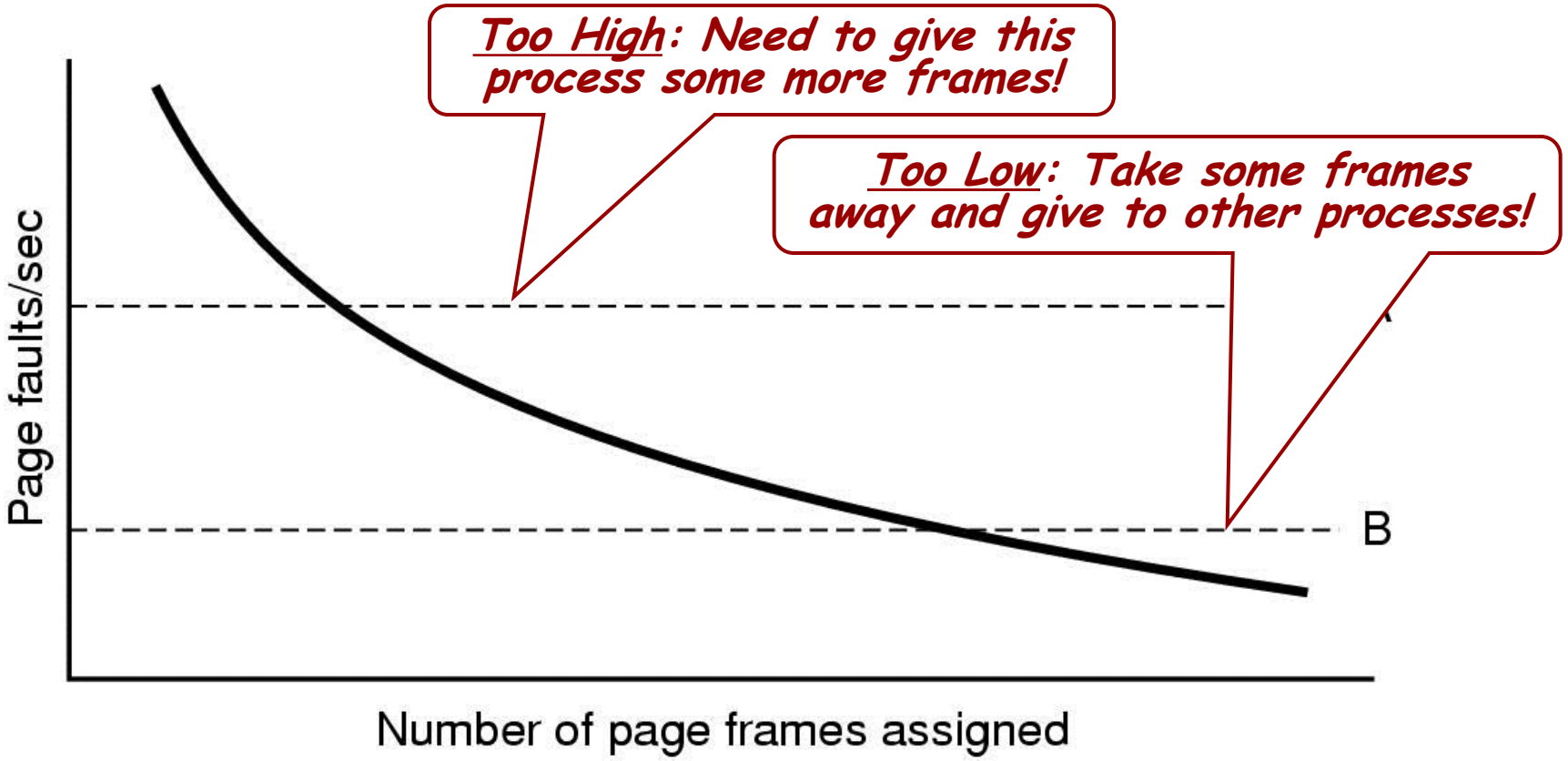


Page Fault Frequency

If T is too small, page fault frequency will be high
If you make it bigger page fault frequency will decline



Page Fault Frequency



Page Fault Frequency

Measure the page fault frequency of each process
Count the number of faults every second

May want to consider the past few seconds as well

Page Fault Frequency

Measure the page fault frequency of each process
Count the number of faults every second

May want to consider the past few seconds as well

Aging:

Keep a running value

Every second

- Count number of page faults
- Divide running value by 2
- Add in the count for this second

Which Algorithm is Best?

Modeling Algorithm Performance

Run a program

- Look at all memory references
- Don’t need all this data
- Look at which pages are accessed

  0000001222333001144440011234444

- Eliminate duplicates

  012301401234

This defines the *Reference String*

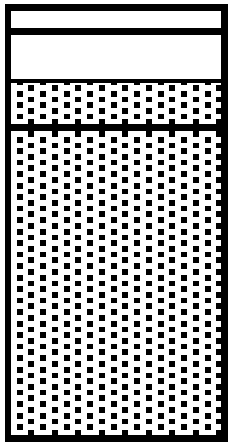
- Use this to evaluate different algorithms
- Count page faults given the same reference string

Proactive Replacement

Replacing victim frame on each page fault typically requires two disk accesses per page fault

Alternative → the O.S. can keep several pages free in anticipation of upcoming page faults.

In Unix: low and high water marks



low < # free pages < high

UNIX Page Replacement

Clock algorithm for page replacement

- If page has not been accessed move it to the free list for use as allocatable page
- If modified/dirty → write to disk (still keep stuff in memory though)
- If unmodified → just move to free list

High and low water marks for free pages

- Pages on the free-list can be re-allocated if they are accessed again before being overwritten

Local vs. Global Replacement

Assume several processes: A, B, C, ...

Some process gets a page fault (say, process A)

Choose a page to replace.

Local page replacement

- Only choose one of A's pages

Global page replacement

- Choose any page

Local vs. Global Replacement

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

Original

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

Local

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

Global

Local vs. Global Replacement

Assume we have

- 5,000 frames in memory and 10 processes

Idea: Give each process 500 frames

Is this fair?

- Small processes do not need all those pages
- Large processes may benefit from even more frames

Idea:

- Look at the size of each process (… but how?)
- Give them a pro-rated number of frames with some minimum

Load Control

Assume:

- The best page replacement algorithm
- Optimal global allocation of page frames

Load Control

Assume:

- The best page replacement algorithm
- Optimal global allocation of page frames

Thrashing is still possible!

Load Control

Assume:

- The best page replacement algorithm
- Optimal global allocation of page frames

Thrashing is still possible!

- Too many page faults!
- No useful work is getting done!
- Demand for frames is too great!

Load Control

Assume:

- The best page replacement algorithm
- Optimal global allocation of page frames

Thrashing is still possible!

- Too many page faults!
- No useful work is getting done!
- Demand for frames is too great!

Solution:

- Get rid of some processes (temporarily swap them out)
- Two-level scheduling (swapping with paging)