CS510 Operating System Foundations

Bruce Irvin (thanks to Jon Walpole for the original version of the slides)

Portland State
UNIVERSITY

Page Replacement

User-program is executing

A Page Fault occurs!

- Indicates that the needed page 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:

- 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 Optimal Algorithm

Idea:

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

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 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 for comparison purposes

- Run the program once
- Generate a log of all memory references
- 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*

Simple to implement

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						
	1					b						
	2		c	c	c	c						
	3				d	d						
Page faults			x	x	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		
	1					b	b	b	b	b		
	2		c	c	c	c	e	e	e	e		
	3				d	d	d	d	d	d		
Page faults			x	x	x	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	c	
	1					b	b	b	b	b	b	
	2		c	c	c	c	e	e	e	e	e	
	3				d	d	d	d	d	d	d	
Page faults			X	X	X	X	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	c	c
	1					b	b	b	b	b	b	b
	2		c	c	c	c	e	e	e	e	e	e
	3				d	d	d	d	d	d	d	a
Page faults			x	x	x	x	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 list of all pages in memory

Keep it in order of when they came into memory

Add new page to head of list

The page at the tail of the list is oldest

FIFO Algorithm

Always replace the oldest page.

- *Replace the page that has been in memory for the longest time*

Implementation

Maintain a list of all pages in memory

Keep it in order of when they came into memory

Add new page to head of list

The page at the tail of the list is oldest

Advantages

Simple to implement and needs no extra hardware support

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 evicting it will cause an immediate page fault

Temporal locality: that which was needed recently probably will be needed again in the near future.

FIFO ignores temporal locality

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 (actually TLB entry!) has a

- *Referenced bit* - set by TLB when page read or updated
- *Dirty / modified bit* - set when page is updated
- 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. i.e., TLB flushing becomes more complicated

Idea: use the information contained in these bits to drive the page replacement algorithm

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

A mix of NRU and FIFO

Sometimes (unfortunately) called the "Clock" algorithm

OS maintains circular list of valid (in-memory) pages

Algorithm: look at the next page in the circular list

If its "referenced bit" is 0 and dirty bit is 0

- Then select it for replacement

Else

- It was used recently; don't want to replace it
- Clear its "referenced bit"
- If dirty, then schedule it for write to swap file
- Move on to next frame in the list

Repeat

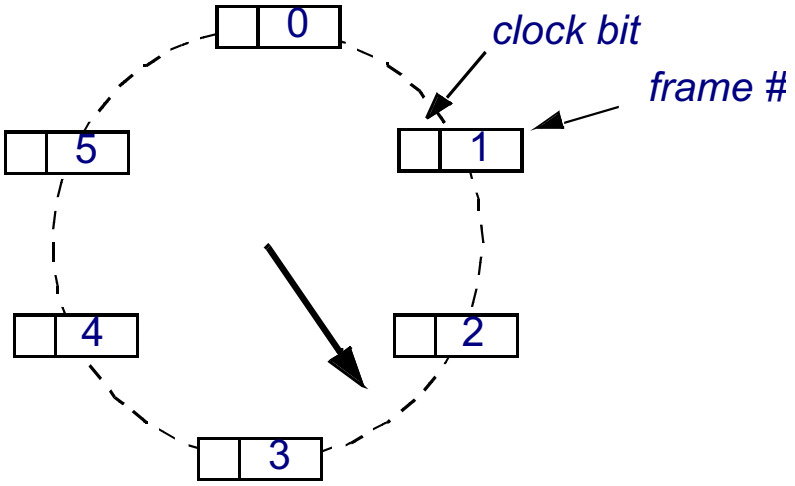
Implementation of Second Chance

Maintain a circular list (called "the clock") of pages in memory

Set ref bit for the page whenever the page is referenced

Search list looking for a victim page that does not have the referenced bit set (and is not dirty)

- 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



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 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

Least Recently Used Algorithm

But how can we implement LRU?

Least Recently Used Algorithm

But how can we implement LRU?

Idea #1:

- Keep a 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 CPU clock cycle
- Every time a TLB entry is used
  - MMU writes the value to the TLB entry
  - This *timestamp* value is the *time-of-last-use*
  - Save TLB bookkeeping data to page table when needed
- When a page fault occurs
  - OS looks through the page table
  - Identifies the entry with the oldest timestamp

Not Frequently Used Algorithm

Bases decision of frequency of use rather than recency

Add a reference "counter" to the reference "bit"

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 might be heavily used

- Its counter is large

The program's behavior changes to a new phase of execution

- Now, this page is rarely used

NFU never forgets!

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

Fix: 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
- It's all overhead all of the time!

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

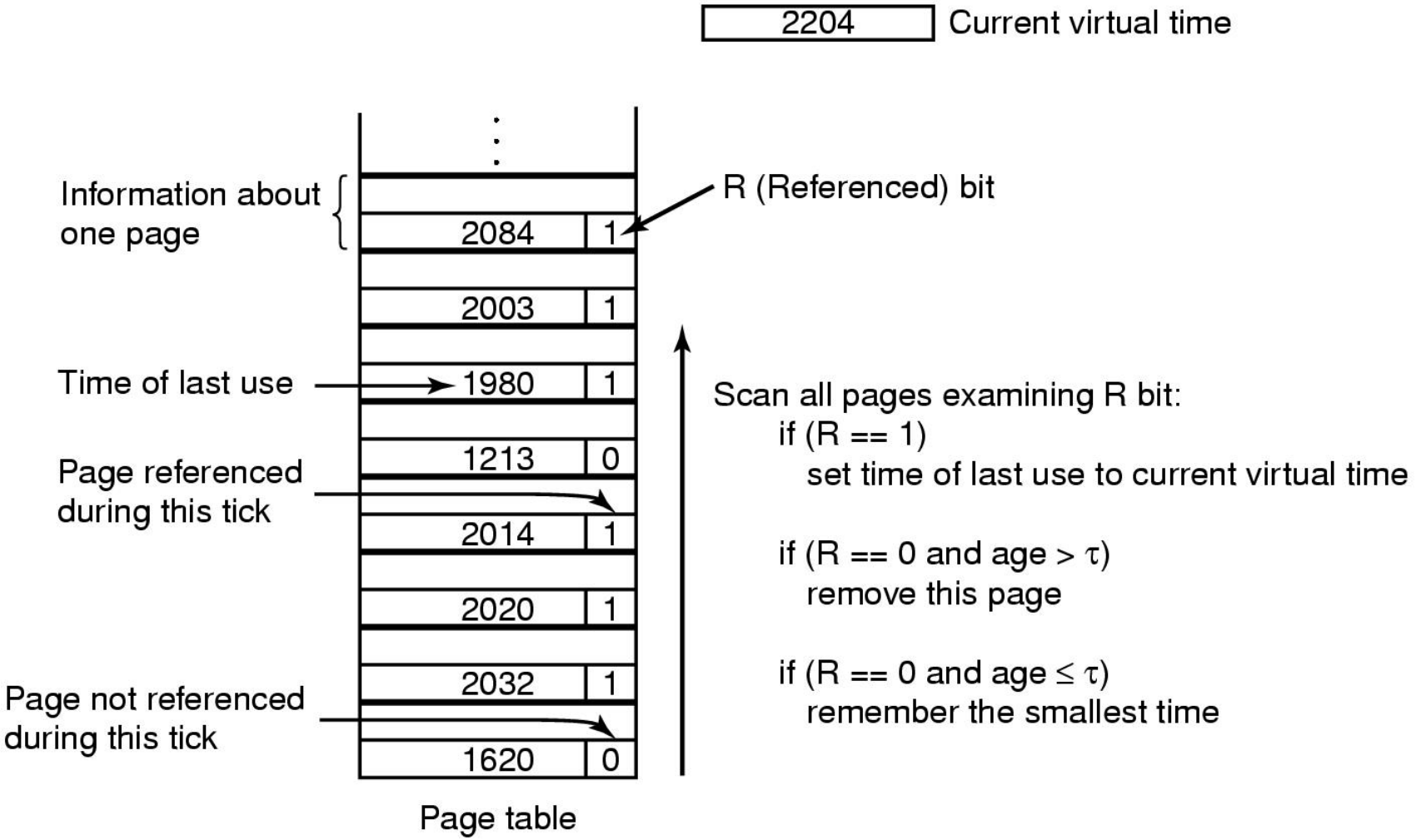
Current Virtual Time

- Only consider how much CPU time this process has seen

Implementation

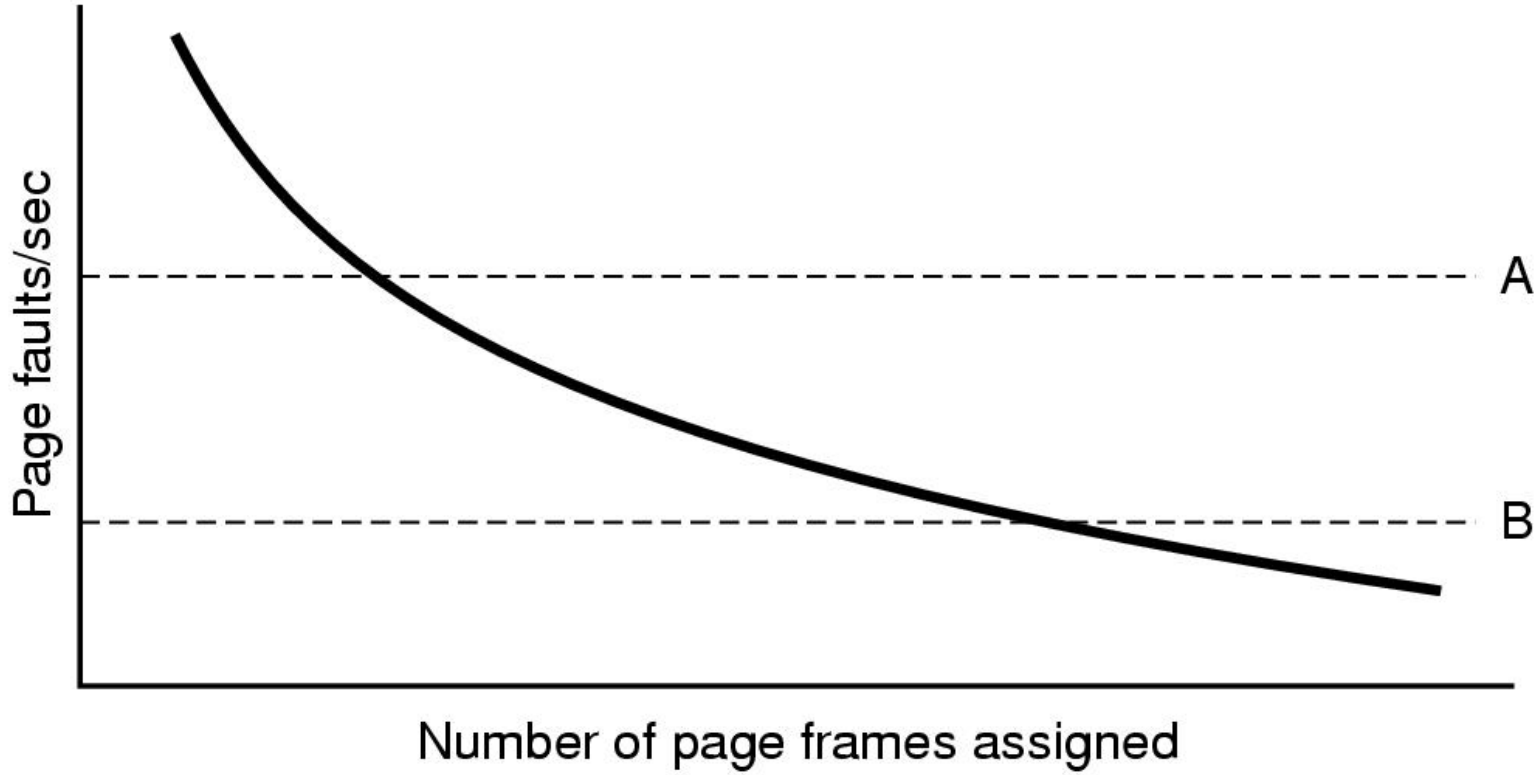
- On each clock tick, look at each page
- Was it referenced since the last check?
  - Yes: make a note of Current Virtual Time
- If a page has not been used in the last T msec,
  - Assume it is not in the working set!
  - Evict it
  - Write it to swap space if it is dirty

Working Set Algorithm

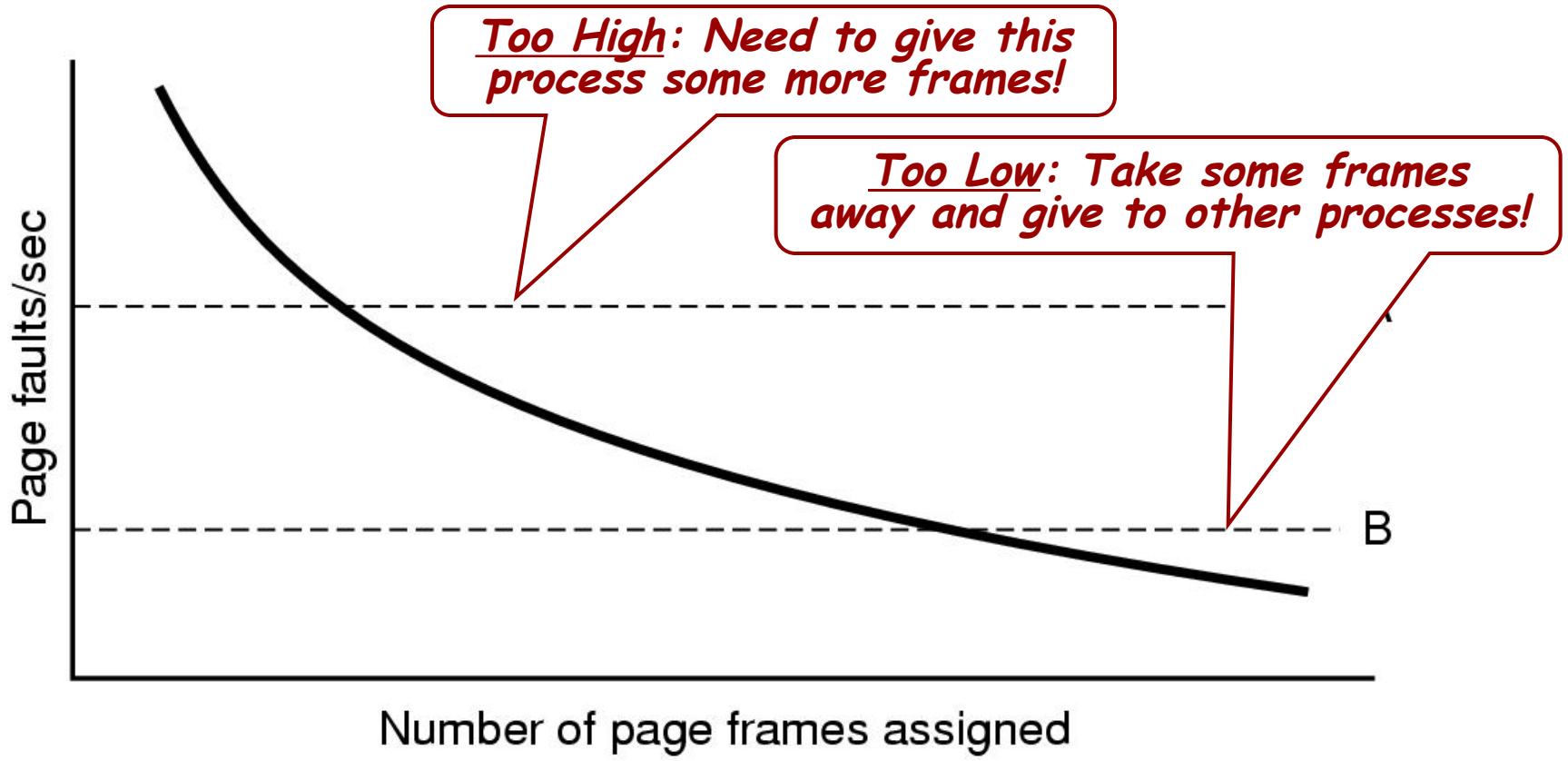


Page Fault Frequency

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
- Look at which pages are accessed
  00000012223333001144440011234444
- Eliminate duplicates
  012301401234

This defines the *Reference String*

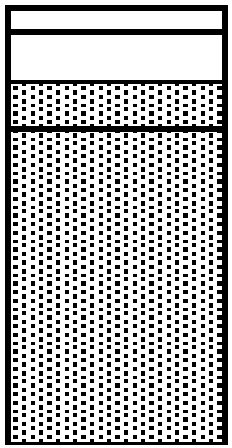
- Use this to evaluate various algorithms
- Count page faults given the same reference string

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 modified/dirty → write to disk (still keep it in memory though)

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

Load Control: if page replacement is too frequent, then suspend/swap entire processes

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)