Chapter 3

Memory Management

Part 2: Page Replacement Algorithms

Outline of Chapter 3

- Basic memory management
- Swapping
- Virtual memory
- Page replacement algorithms
- Modeling page replacement algorithms

in this file

- Design issues for paging systems
- Implementation issues
- Segmentation

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
Remove the page in it
- If it has been modified, must write it back to disk
  - The "dirty" bit

Look at user process and figure out which page was needed
Read the needed page into this frame
Restart the interrupted process
- Retry the same instruction
  - May need to manipulate the machine state

Page Replacement Algorithms

Which frame to replace?

Algorithms:

- **The Optimal Algorithm**
- **FIFO**
- **Not Recently Used**
- **Second Chance**
- **Clock**
- **Least Recently Used (LRU)**
- **Not Frequently Used (NFU)**
- **Working Set**
- **WSClock**

The Optimal Page Replacement Algorithm

Idea:

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

The Optimal Page Replacement Algorithm

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

The Optimal Page Replacement Algorithm

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

However:

Simulation studies

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

The FIFO Page Replacement Algorithm

Always replace the oldest page.

"Replace the page that has been in memory for the longest time."

The FIFO Page Replacement Algorithm

Always replace the oldest page.

> ***"Replace the page that has been in memory for the longest time."***

Maintain a linked list of all pages in memory
Keep in order of when they came into memory
The page at the front of the list is oldest
Add new page to end of list

The FIFO Page Replacement Algorithm

Always replace the oldest page.

"Replace the page that has been in memory for the longest time."

Maintain a linked list of all pages in memory
Keep in order of when they came into memory
The page at the front of the list is oldest
Add new page to end of list

Disadvantage:

The oldest page may be needed again soon
Some page may be important
It will get old, but replacing it
will cause an immediate Page Fault

Page Table: Referenced and Dirty Bits

Each page has a...

Valid Bit - checked when page is read or written

ReadOnly Bit - checked when page is written
BLITZ calls it a "Writable Bit" (0=readonly)

Referenced Bit - set by MMU when page read / written

Dirty Bit - set when page is written
Sometimes called "Modified Bit"

Page Table: Referenced and Dirty Bits

Each page has a...

Valid Bit - checked when page is read or written

ReadOnly Bit - checked when page is written
BLITZ calls it a “Writable Bit” (0=readonly)

Referenced Bit - set by MMU when page read / written

Dirty Bit - set when page is written
Sometimes called “Modified Bit”

This algorithm will use these bits

Page Table: Referenced and Dirty Bits

Unfortunately, some hardware has
only a ReadOnly Bit but no Dirty Bit

Idea:

- **Software sets the ReadOnly bit for all pages**
- **When program tries to update the page...**
  A trap occurs
- **Software sets the Dirty Bit**
  and clears the ReadOnly bit
- **Resumes execution of the program**

The Not Recently Used Page Replacement Alg.

Use the Referenced Bit and the Dirty Bit

Initially, all pages have

Referenced Bit = 0

Dirty Bit = 0

Periodically...

(e.g. whenever a clock tick (timer interrupt) occurs)

Clear the Referenced Bit

The Not Recently Used Page Replacement Alg.

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 page from class 1.
If none, choose a page from class 2.
If none, choose a page from class 3.
If none, choose a page from class 4.

The Second Chance Page Replacement Alg.

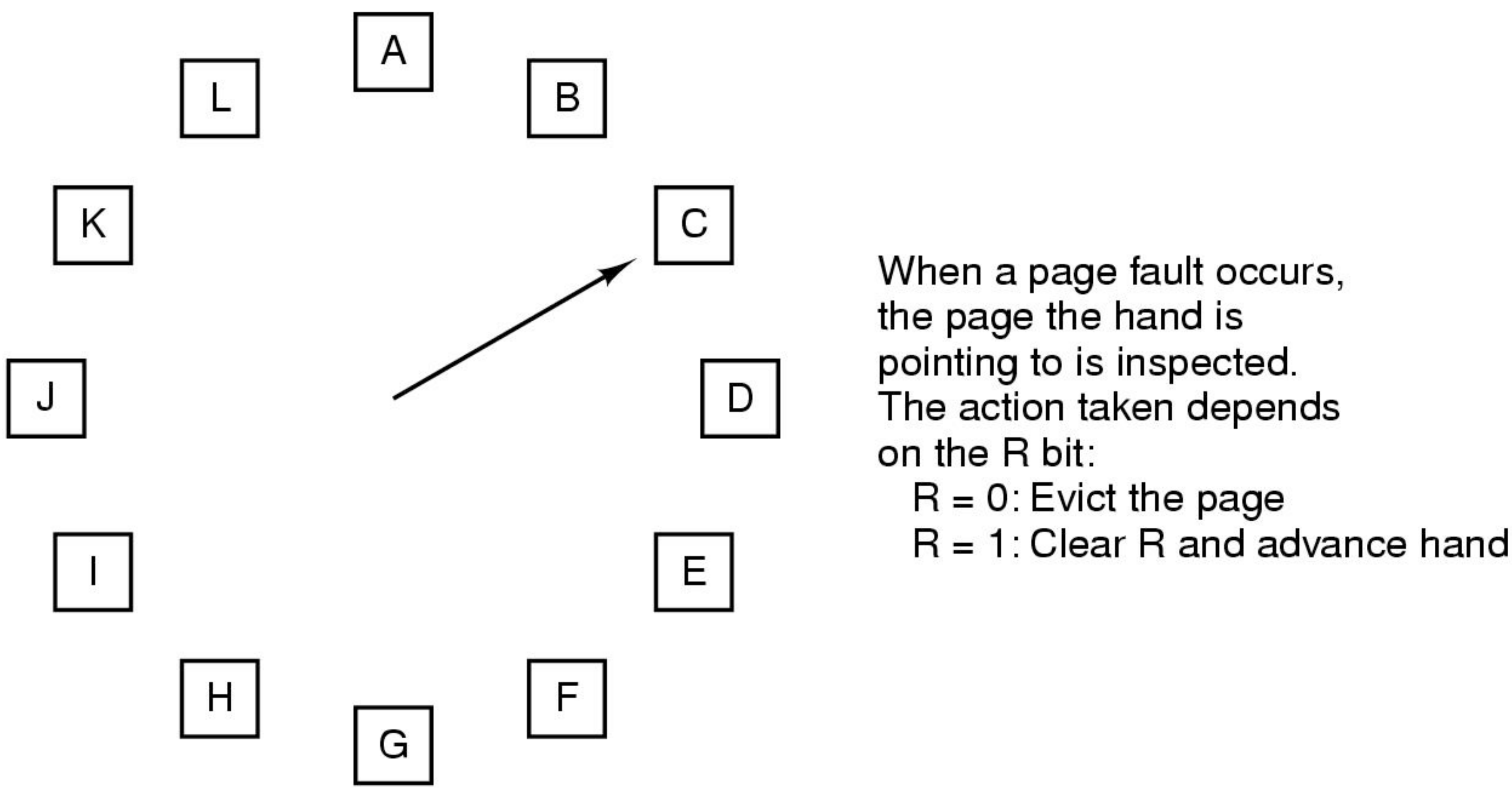
- Modification to FIFO
- Pages kept in a linked list
  - Oldest is at the front of the list
- Look at the oldest page
  - If its “referenced bit” is 0...
    - Select it for replacement
  - Else
    - It was used recently; don’t want to replace it
    - Clear its “referenced bit”
    - Move it to the end of the list
    - Repeat
- Everything was used in last clock tick?
- Eventually we will get back to the oldest page
  - This time, its ref. bit will be 0 and we’ll select it.

The Clock Page Replacement Alg.

Same as "second chance" algorithm

Keep the pages in a circular list

- Current position



The Least Recently Used Algorithm (LRU)

Keep track of when a page is used.
Replace the page that has been used least recently.

The Least Recently Used Algorithm (LRU)

Keep track of when a page is used.
Replace the page that has been used least recently.

Implementation #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.**

"on every memory reference..."

- **Not feasible in software**

The Least Recently Used Algorithm (LRU)

Keep track of when a page is used.
Replace the page that has been used least recently.

Implementation #2:

- **MMU maintains a counter**
- **Incremented on every clock cycle**
- **Every time a page table entry is used**
  - **MMU writes the value to the entry**
  - **"*timestamp*" / "*time-of-last-use*"**
- **When a page fault occurs**
  - **Software looks through the page table**
  - **Idenitifies the entry with the oldest timestamp**

The Least Recently Used Algorithm (LRU)

Keep track of when a page is used.
Replace the page that has been used least recently.

Implementation #3:

- **No hardware support**
- **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**

The Not Frequently Used (NFU) Algorithm

- Associate a counter with each page
- On every timer 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.

The Not Frequently Used (NFU) Algorithm

- **Associate a counter with each page**
- **On every timer 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.**

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!

Aging

Given:

A series of numbers, being produced over time.

$x_0, x_1, x_2, \ldots, x_i$ (${x_i}$ is the most recent value)

Goal:

Compute an average value...

with most recent values getting greater weights.

Really want a "running average"

$T_0, T_1, T_2, \ldots, T_i$

with most recent values getting greater weights.

a = the weight of current value ($0 < a < 1$)

Formula:

$$T_i = (a)\, x_i + (1-a)\, T_{i-1}$$

Aging

Given:

$x_0, x_1, x_2, \ldots, x_i$

Example:

Let $a = ½$

$T_0 = x_0$

$T_1 = 1/2\ x_1 + 1/2\ x_0$

$T_2 = 1/2\ x_2 + 1/4\ x_1 + 1/4\ x_0$

$T_3 = 1/2\ x_3 + 1/4\ x_2 + 1/8\ x_1 + 1/8\ x_0$

$T_3 = 1/2\ x_3 + 1/2\ \ (1/2\ x_2 + 1/4\ x_1 + 1/4\ x_0)$

$T_3 = 1/2\ x_3 + 1/2\ \ (T_2)$

$T_i = 1/2\ x_i + 1/2\ \ (T_{i-1})$

Formula:

$T_i = (a)\ x_i + (1\text{-}a)\ T_{i-1}$

Aging

Assume $a = \frac{1}{2}$

$$T_i = \tfrac{1}{2} x_i + \tfrac{1}{2} T_{i-1}$$
$$T_i = \tfrac{1}{2} (x_i + T_{i-1})$$

This can be computed efficiently!

To divide by two... Just shift right 1 bit.
On each iteration:
- **Add in the new value**
- **Shift everything right 1 bit**

NFU Modification: Aging

- Associate a counter with each page
- On every timer interrupt, 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.

100000 =32
010000 = 16
001000 = 8
000100 = 4
100010 = 34
111111 = 63

Working Set Page Replacement

Demand Paging

Pages are only loaded when accessed

When process begins, all pages marked INVALID

Working Set Page Replacement

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 Page Replacement

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?**

Working Set Page Replacement

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?

Thrashing

- Pages faults every few instructions
- No work gets done

Working Set Page Replacement

Prepaging

Load pages before they are needed

Main Idea:

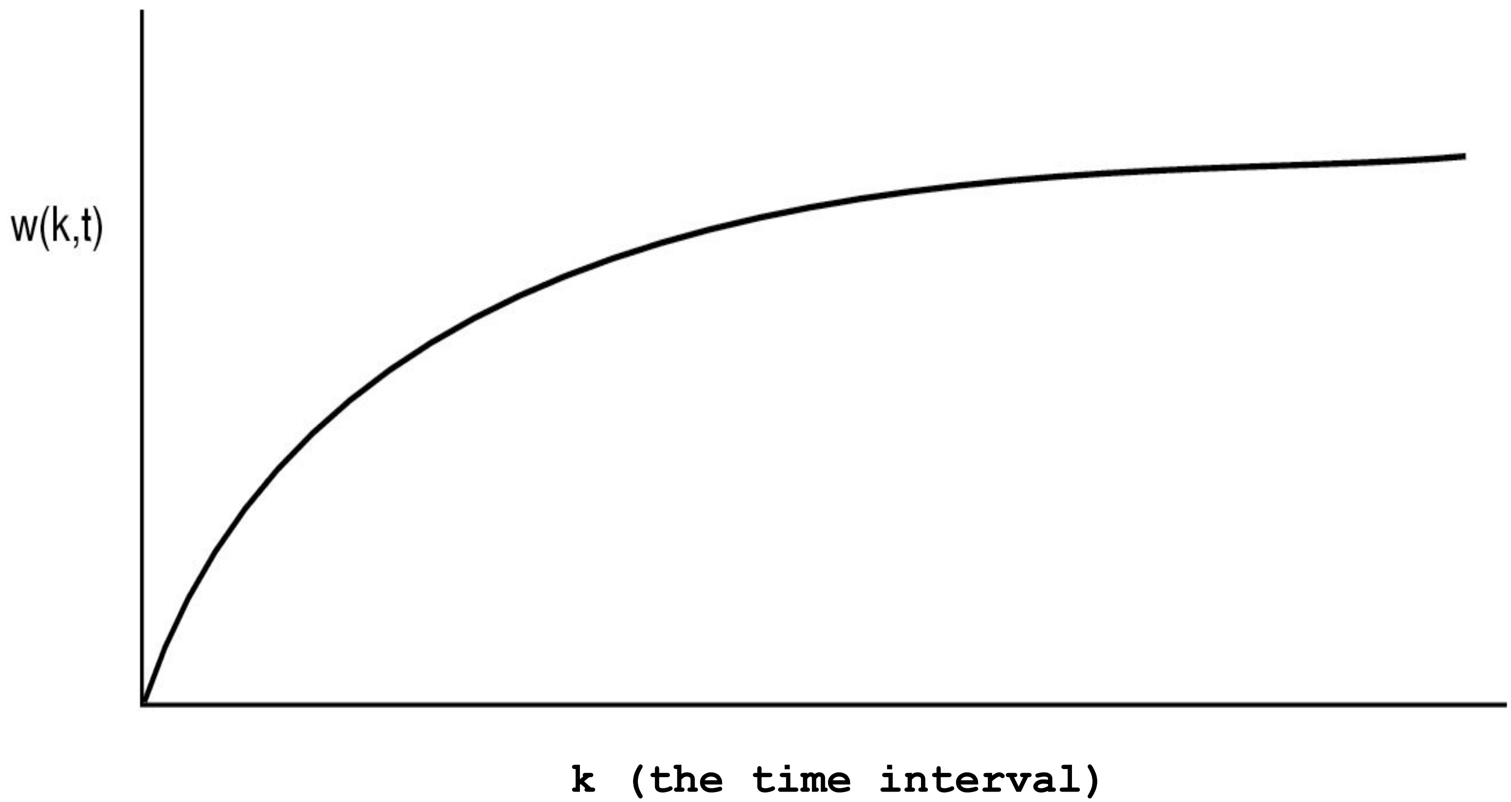
Identify the process's "working set"

How big is the working set?

Look at the last K memory references
As K gets bigger, more pages needed.
In the limit, all pages are needed.

Working Set Page Replacement

The size of the working set:



Working Set Page Replacement

Idea:

Look back over the last T msec of time

Which pages were referenced?

This is the working set.

Current Virtual Time

Only care about how much CPU time this process has seen.

Implementation

On each timer interrupt, look at each page

Was it referenced?

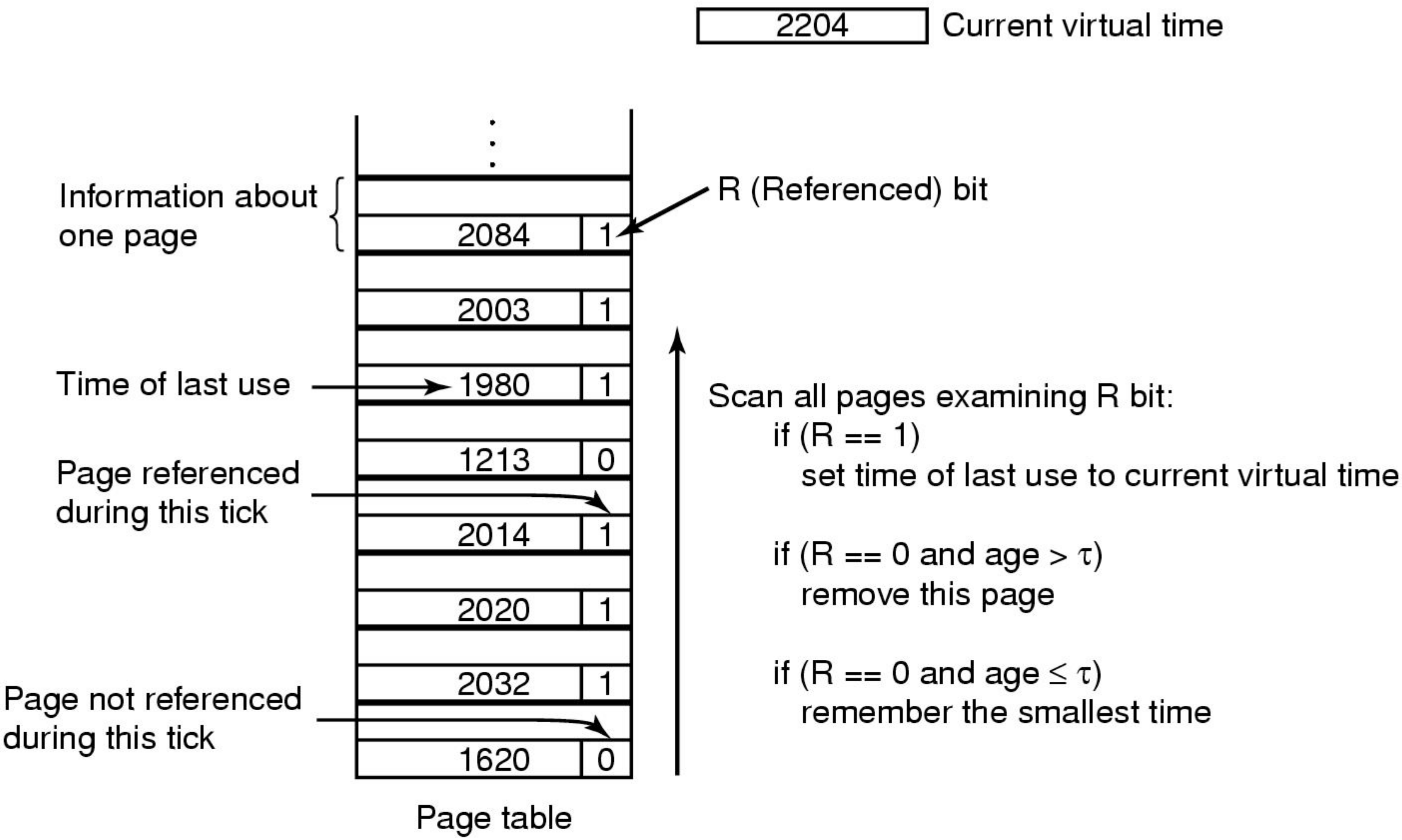
Yes: Make a note of Current Virtual Time

If a page has not been used in the last T msec,

It is not in the working set!

Evict it; write it out if it is dirty.

Working Set Page Replacement



The WSClock Page Replacement Algorithm

All pages are kept in a circular list.
As pages are added, they go into the ring.
The “clock hand” advances around the ring.
Each entry contains “time of last use”.
Upon a page fault...
- If Reference Bit = 1...
  - Page is in use now. Do not evict.
  - Clear the Referenced Bit.
  - Update the “time of last use” field.

The WSClock Page Replacement Algorithm

If Reference Bit = 0
- **If the age of the page is less than T...**
  - **This page is in the working set.**
  - **Advance the hand and keep looking**
- **If the age of the page is greater than T...**
  - **If page is clean**
    - **Reclaim the frame and we are done!**
  - **If page is dirty**
    - **Schedule a write for the page**
    - **Advance the hand and keep looking**

Summary

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Modelling Page Replacement

Run a program
- Look at all memory references
- Don't need all this data
- Look at which pages are accessed
  - 0000001222333300114444001123444
- Eliminate duplicates
  - 012301401234

Reference String
- Use this to evaluate different page replacement algorithms

Belady's Anomaly

If you have more page frames (i.e., more memory)...

You will have fewer page faults, right???

Belady's Anomaly

If you have more page frames (i.e., more memory)...
You will have fewer page faults, right???

Not always!

Belady's Anomaly

If you have more page frames (i.e., more memory)...
You will have fewer page faults, right???

Not always!

Consider FIFO page replacement
Look at this reference string:
012301401234

Belady's Anomaly

If you have more page frames (i.e., more memory)...
You will have fewer page faults, right???

Not always!

Consider FIFO page replacement
Look at this reference string
012301401234

Case 1:
3 frames available --> 9 page faults
Case 2:
4 frames available --> 10 page faults

Belady's Anomaly

All pages frames initially empty

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	0	1	4	4	4	2	3	3	
			0	1	2	3	0	1	1	1	4	2	2	
Oldest page				0	1	2	3	0	0	0	1	4	4	
		P	P	P	P	P	P	P			P	P		9 Page faults

FIFO with 3 page frames

Belady's Anomaly

All pages frames initially empty

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	0	1	4	4	4	2	3	3	
			0	1	2	3	0	1	1	1	4	2	2	
Oldest page				0	1	2	3	0	0	0	1	4	4	
		P	P	P	P	P	P	P			P	P		9 Page faults

FIFO with 3 page frames

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	3	3	4	0	1	2	3	4	
			0	1	2	2	2	3	4	0	1	2	3	
Oldest page				0	1	1	1	2	3	4	0	1	2	
					0	0	0	1	2	3	4	0	1	
		P	P	P	P			P	P	P	P	P	P	10 Page faults

FIFO with 4 page frames