Page Replacement

Operating Systems
Baochun Li
University of Toronto

Introduction

We have seen that the OS allocates memory frames to programs on demand (i.e., page fault)

- If no frame is available, then OS needs to evict another page to free a frame
- Which page should be evicted?

A page "cache" miss is similar to a TLB miss or a memory cache miss

- However, a miss may require accessing the disk
- So miss handling can be very expensive
- Disk access times are at least 1000x memory access times

When will paging work well?

Paging can only work well if page replacement occurs rarely

Paging schemes depend on the locality of reference

Spatial locality

- Programs tend to use a small fraction of their memory, or
- Memory accesses are close to memory accessed recently

Temporal locality

- Programs use same memory over short periods of time, or
- Memory accessed recently will be accessed again

Programs normally have both kinds of locality, and the overall cost of paging is not very high

Why not just evict a random page?

If a page evicted is used again in the near future, it needs to be brought back into memory

Challenge: How to find a page that is least used to evict?

- Same problem applies to other cache systems (such as memory cache and web cache)

The Optimal Page Replacement Algorithm

The page that is not needed for the longest time in the future should be evicted

Assuming that the future can be perfectly predicted

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

The Optimal Page Replacement Algorithm

Problem

We cannot accurately predict the future

So we do not know when a given page will be next needed

The optimal algorithm is not realizable

However it can be used in simulation studies

Run the program once

Generate a log of all page references

Use the log in the **second run** to simulate optimal algorithm

Use the "optimal" algorithm as a control case for evaluating other algorithms

First In First Out (FIFO)

Replace the page that has been in memory for the longest time (oldest page)

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

First In First Out (FIFO)

Implementation for replacing the oldest page

- Maintain a linked list of all pages in memory
- Keep the list in order of when pages came into memory
- Add the new page to the end of list

On a page fault

- Choose page at the front of the list (oldest page)

Problem

- The oldest page may be needed again soon
  - Some page may be important throughout execution
  - When it gets old, replacing it may cause immediate page fault

FIFO suffers from Bélády's anomaly

Bélády's anomaly —

Increasing the number of page frames results in an increase in the number of page faults

This is very bad!

Can We Do Better than FIFO?

- **Need to predict page access pattern in the future**
  - But we can only learn from the past
- **One idea — pages used in the recent past should not be evicted**
  - Assumption: pages used recently are likely to be used in the near future
  - Need a way to track past page references
  - Requires hardware support!

Page Table Bits Revisited

Each page table entry has:

- **Referenced** bit
  - Set by CPU when the page is read or written
  - Cleared by OS software (never cleared by hardware)
- **Modified (dirty)** bit
  - Set by CPU when the page is written
  - Cleared by OS software (never cleared by hardware)

TLB may have the most recent copy of them

- Hardware/OS must synchronize it with page table entry bits

Can we use page table bits to estimate the page access pattern in the past?

Second Chance

FIFO, but give a second chance to referenced pages

Maintain a linked list of all pages in memory

- New pages are added to the end of list

On a page fault

- Look at the first page in the list (oldest page)
- If its referenced bit is 0, select it for replacement
- Else, **clear** referenced bit, move page to end as if it is a **new** page, repeat

If every page was referenced, then second chance reverts back to FIFO

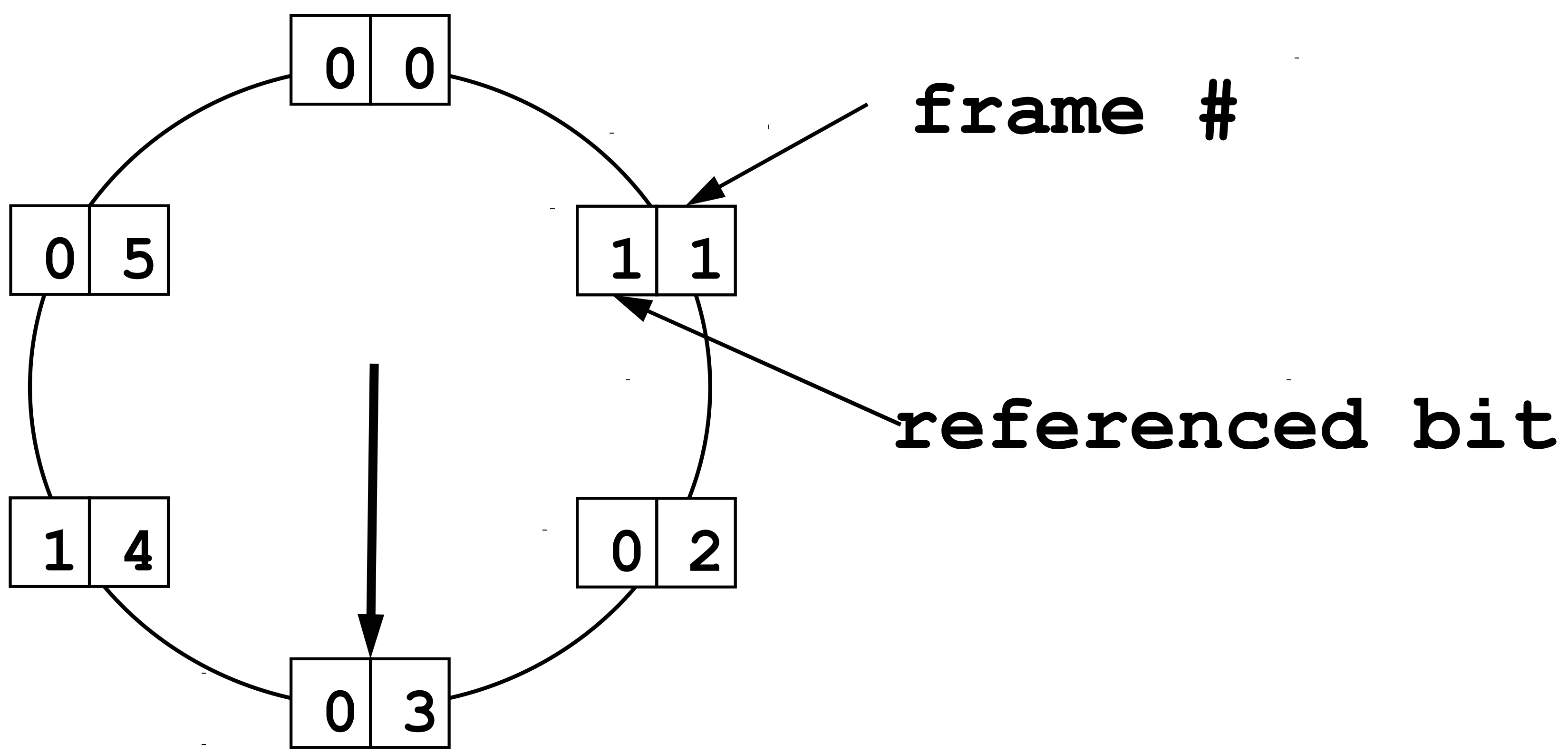
Clock Algorithm

An implementation of second chance

Maintain a circular list of pages in memory

On a page fault

- The "hand of the clock" sweeps over circular list
- Looks for a page that does not have the referenced bit set (instead of moving pages in FIFO list)



Enhanced Second Chance

Also called Not Recently Used (NRU)

Replace the page that is not recently used

Initially, all pages have

- Referenced bit = 0
- Dirty bit = 0

Periodically (e.g., every timer interrupt) clear the referenced bit of all pages

- Then, the referenced bit indicates that a page was recently accessed

Enhanced Second Chance

On a page fault, pages are in 4 classes

Class	Referenced	Dirty
1	0	0
2	0	1
3	1	0
4	1	1

How is this class possible?

Choose a random page from the lowest non-empty class to evict

Least Recently Used (LRU)

A refinement of NRU that replaces the page that has **not been used for the longest period of time**

Needs to track how recently a page was used

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 Implementation — Option 1

Keep a stack of all pages

On each memory reference

- Move corresponding page to the top of the stack
- Best implemented as a doubly linked list

On a page fault

- Choose page at the bottom of the stack (least recently used)

LRU Implementation — Option 1

Move referenced page to the top of the stack

c	a	d	b	e	b	a	b	c	d
a	c	a	d	b	e	b	a	b	c
b	b	c	a	d	d	e	e	a	b
d	d	b	c	a	a	d	d	e	a

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 Implementation — Option 1

Problems

Requires moving list elements on each memory access

Each memory access becomes several accesses!

Not implementable with hardware

LRU Implementation — Option 2

MMU (hardware) maintains a counter that is incremented for every memory reference

Every time a page table entry is used

MMU writes the value of the counter to the page table entry

This timestamp value is the "time of last use"

On a page fault

OS software looks through the page table, and identifies the entry with the oldest timestamp

Problem

Updating of the timestamps must be done for every memory reference

LRU Implementation — Option 3

Additional-Reference-Bits Algorithm

Maintain an 8-bit counter for each page in software

- initially zero

At each timer interrupt (or any regular interval) —

- The referenced bit is shifted into the high-order bit of the counter
- The other bits are shifted to the right
- The low-order bit is discarded
- The referenced bit is then cleared

On a page fault, evict the page with the lowest counter

- arbitrarily evict one if more than one candidate

The Additional-Reference-Bits algorithm

	(a)	(b)	(c)	(d)	(e)
	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
Page					
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

Problems with Additional-Reference-Bits

Granularity of record-keeping is limited by the frequency of the timer interrupts

- Records only one bit per interval
- Lost the ability to distinguish references early in the clock interval from those occurring later

Counters have a finite number of bits

- Limits its past time horizon
- All we can do is to pick one of them at random

Comparison of Page Replacement Algorithms

Optimal	Not implementable, useful as a benchmark
FIFO	May evict important pages
Enhanced SC	Simple, but crude approximation of LRU
Second Chance/Clock	Major improvements over FIFO, Clock is realistic
LRU	Excellent, but difficult to implement exactly
ARB	Efficient algorithm that approximates LRU well
Working Set	Good, if an appropriate time horizon T is used

Working Set Model

Locality of reference revisited

Spatial locality: processes tend to use small a fraction of their memory

Temporal locality: processes tend to use the same memory over short periods of time

Working Set

The set of pages a process needs currently

If working set is in memory, no page faults occur

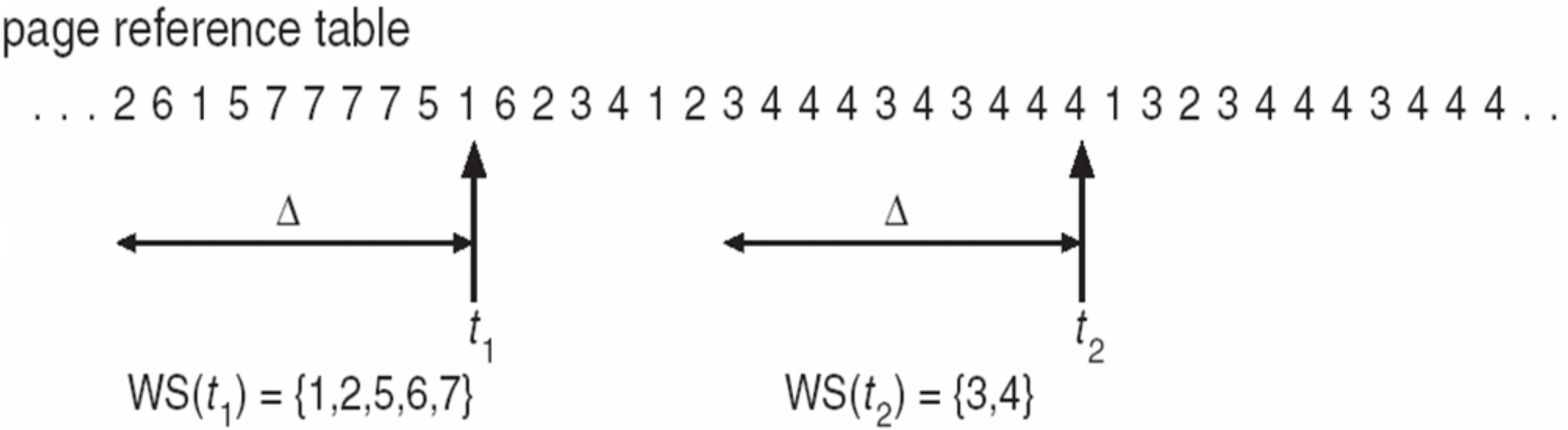
What if you can't get the working set into memory?

Thrashing — not enough frames to accommodate the WS

Page faults occur every few instructions

The user-level program makes little progress

The Working Set: Examples



Demand Paging vs. Prefetching

Demand paging: loading pages on demand initially when a process starts to run

Prefetching: load the working set of the process **before** letting it run, minimizes page faults

But how does the OS determine the working set?

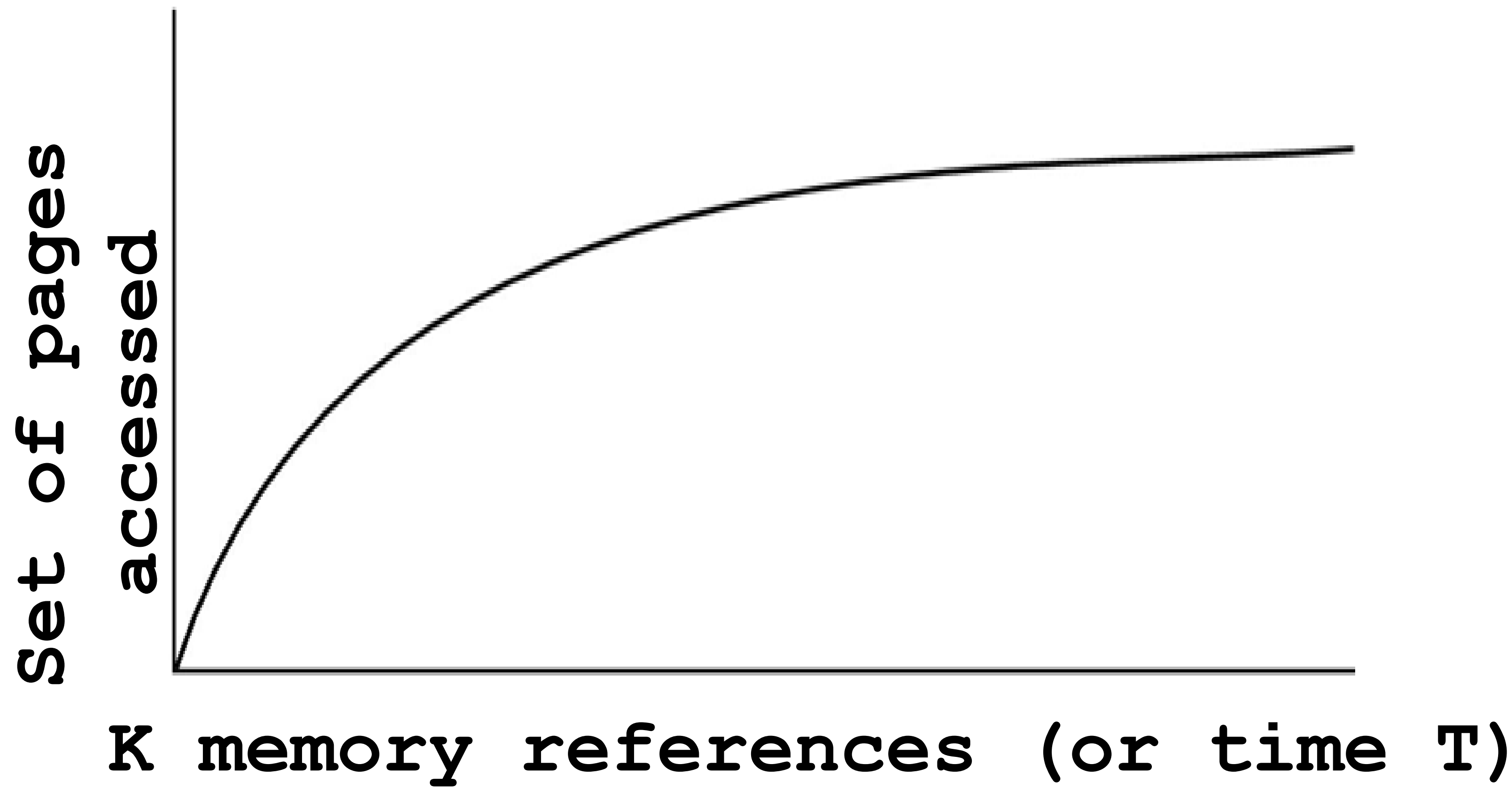
How Big Is the Working Set?

Look at the last K memory references

Alternatively, look back over last time T, the working set time interval

As K (or T) gets larger, more pages are needed

Goal: Design a page replacement algorithm that keeps this working set in memory



Approximating the working set model

Approximate with interval timer + a reference bit

Example: T = 10,000 time units

Timer interrupts after every 5000 time units

Keep in page table: 2 bits for each page

On a timer interrupt, copy and set the value of the Referenced bit to 0

If one of the bits is 1 => the page is in the WS

The Working Set Algorithm

Hardware sets the R bit when a page is referenced

The virtual time is the time that the process has run on the CPU

On a timer interrupt —

If R is 1, it is cleared, and the current virtual time is written to "Time of Last Use"

The Working Set Algorithm: On a Page Fault

On a page fault —

- If R is 1, the current virtual time is written to "Time of Last Use"
- If R is 0, and the age (current virtual time - time of last use) > T, evict
- If R is 0, and age <= T, record the page with the greatest age

If the entire page table has been scanned

- If one or more pages has R = 0, evict the one with the greatest age
- Otherwise, all pages have been referenced, evict one at random, preferably a clean page (dirty = 0)

Local vs. Global Replacement

Say a process gets a page fault and a page needs to be replaced

Which process's page should be replaced?

Policy 1: Local page replacement

- Choose a page of the same process

Policy 2: Global page replacement

- Choose a page of any process

Some algorithms can be used with either policy

- e.g., LRU can be used with both local or global replacement
- But not the Working Set algorithms

Local vs. Global Replacement

Example: Process A has a page fault

Original	Time last accessed	Local	Global
A0	10	A0	A0
A1	7	A1	A1
A2	5	A2	A2
A3	4	A3	A3
A4	6	A4	A4
A5	3	A6	A5
B0	9	B0	B0
B1	4	B1	B1
B2	6	B2	B2
B3	2	B3	A6
B4	5	B4	B4
B5	6	B5	B5
B6	12	B6	B6
C1	3	C1	C1
C2	5	C2	C2
C3	6	C3	C3

Replace earliest local page

Replace earliest global page

Original **Local** **Global**

Problem With Local Page Replacement

Suppose there are 10 processes and 5,000 frames in memory

Should each process get 500 frames?

No

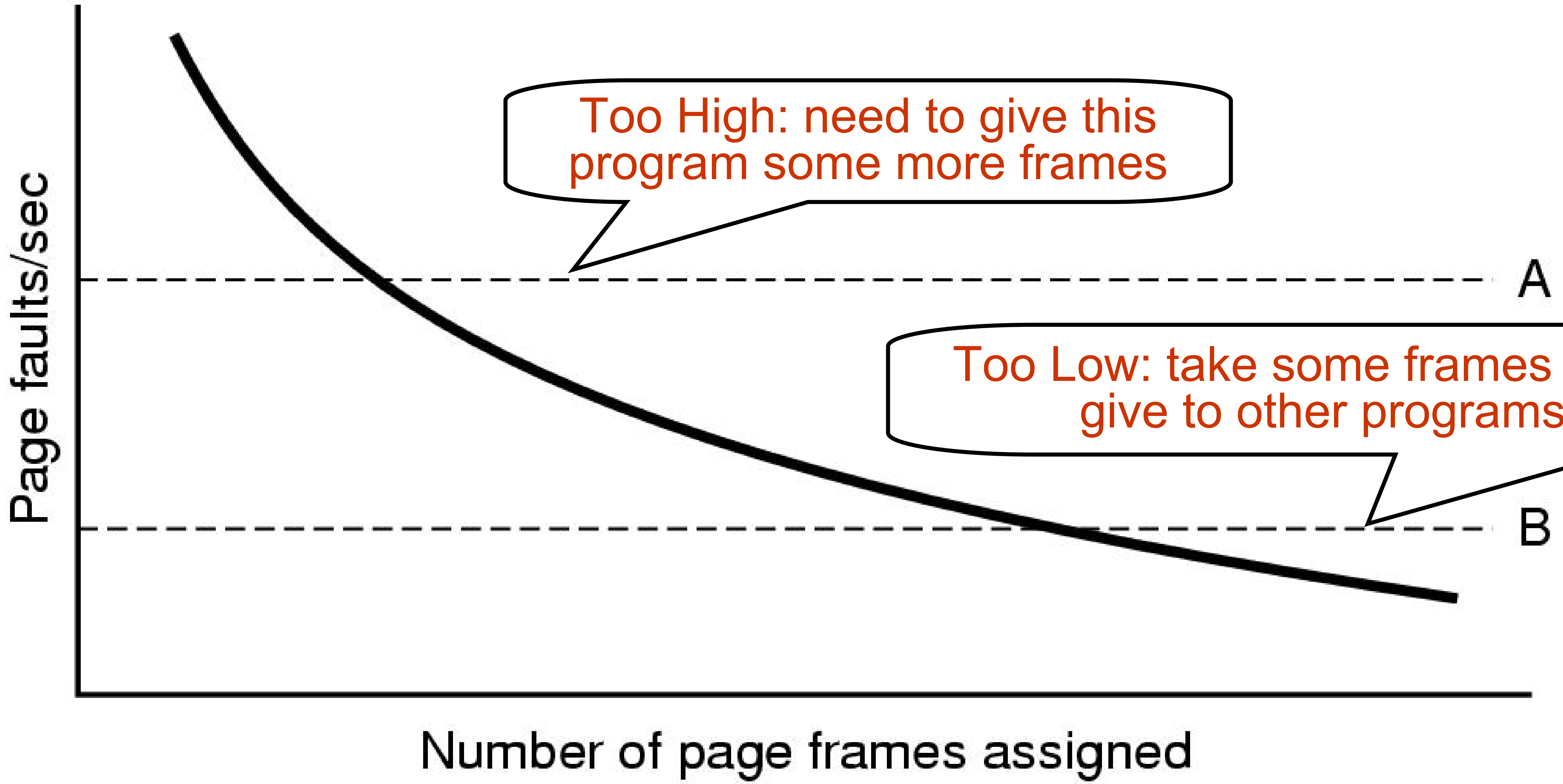
- Small processes do not need all those pages
- Large processes may benefit from more frames

Idea

- Look at the needs of each process and give each an adequate number of frames to prevent thrashing
- But how?

Page Fault Frequency

Page fault frequency declines as a process is assigned more pages



Page Fault Frequency

The page fault frequency provides an estimate of the working set needs of a process

Goal: Allocate frames so that the page fault frequency is roughly equal for all processes

How should the page fault frequency be measured?

For each process

- On each fault, increment a counter **f**
  - f = f + 1
- Every second, update Fault Frequency (**ff**, in faults/second) via **aging**
  - ff = (1 - a) * ff + a * f, f = 0 (0 < a < 1, when a -> 1, history is ignored)

This global page allocation algorithm can then be combined with a local page replacement algorithm

Paging Daemon Revisited

It is expensive to run replacement algorithm on each page fault

Instead, OS can use a paging daemon to maintain a pool of free frames

- Runs replacement algorithm when pool reaches low watermark
- Writes out dirty pages and frees them
- Frees enough pages until pool reaches high watermark

Frames in pool still hold previous contents

- Can be rescued if page is referenced before reallocation

Example: Windows

Demand paging

Working-set minimum and working-set maximum (usually 50 and 345 pages)

If page fault occurs for a process below its working-set maximum:

allocates a page from a list of free pages

If at the working-set maximum, select a page for replacement using a local page replacement policy (variation of the Clock algorithm)

If the amount of free pages falls below a threshold —

Evaluate the number of pages allocated to a process

If it is more than the working-set minimum, evict pages until it reaches the minimum

What we've covered so far

Three Easy Pieces: Chapter 22 (Beyond Physical Memory: Policies)