CS 370: OPERATING SYSTEMS
[VIRTUAL MEMORY]

Computer Science
Colorado State University

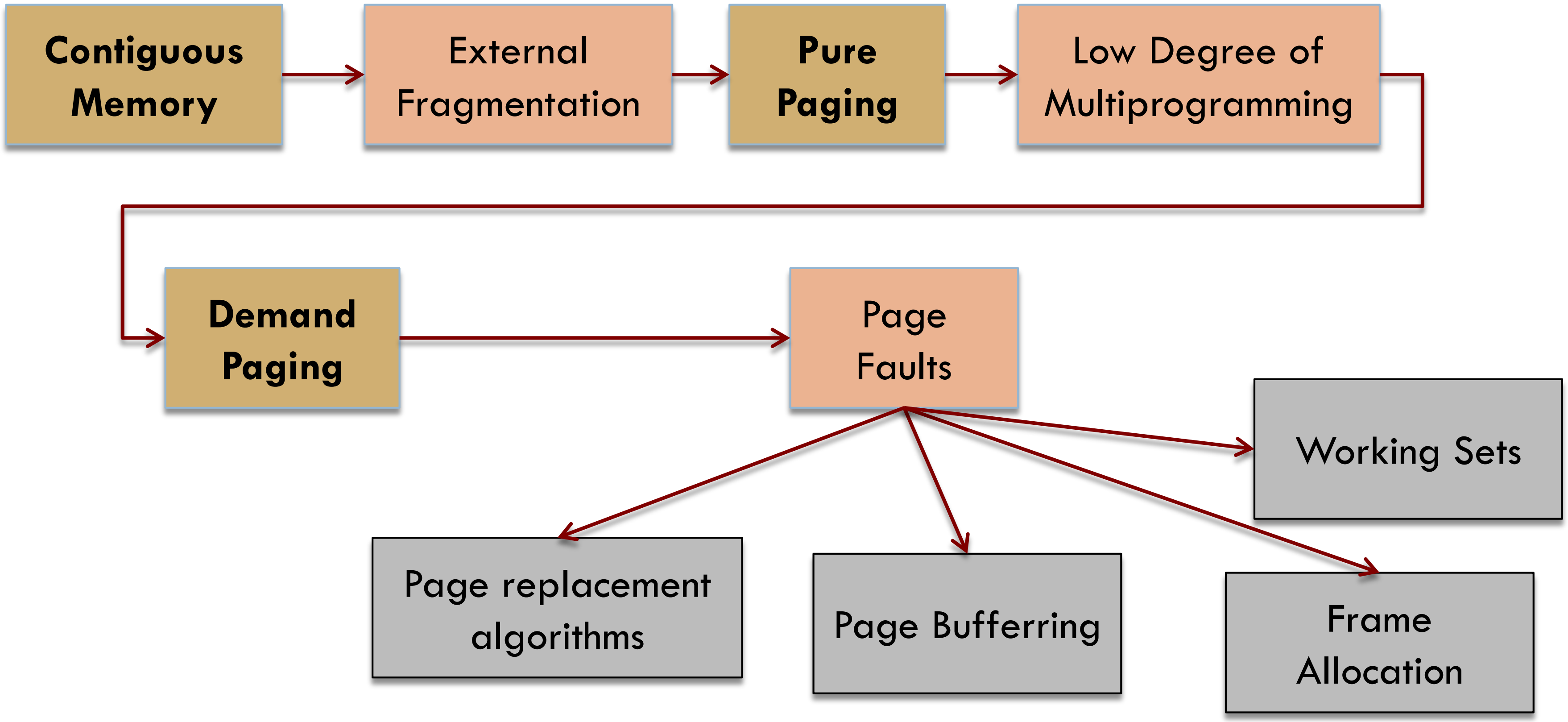
Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: SHRIDEEP PALLICKARA

Topics covered in this lecture

- FIFO Page Replacement Algorithm
- Belady's Anomaly
- Stack Algorithms
- Page Buffering
- Frame Allocations

How we got here …



Factors involved in determining page faults

- **Reference string** of executing process
- Page **replacement algorithm**
- Number of physical memory **frames** available

- Intuitively:
  - Page faults reduce as the number of available frames increase

FIFO Page Replacement Algorithm

FIFO page replacement algorithm: Out with the old; in with the new

- When a page must be replaced
  - Replace the **oldest** one
- OS maintains list of all pages currently in memory
  - Page at head of the list: Oldest one
  - Page at the tail: Recent arrival
- During a page fault
  - Page at the head is removed
  - New page added to the tail

FIFO example: 3 memory frames

Reference String

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Youngest	7	0	1	2	2	3	0	4	2	3	0	0	0	1	2	2	2	7	0	1
		7	0	1	1	2	3	0	4	2	3	3	3	0	1	1	1	2	7	0
Oldest			7	0	0	1	2	3	0	4	2	2	2	3	0	0	0	1	2	7

No page fault

Belady's Anomaly

Intuitively the greater the number of memory frames, the lower the faults

- Surprisingly this is **not always** the case

- In 1969 Belady, Nelson and Shedler discovered counter example* in FIFO
  - FIFO caused more faults with 4 frames than 3

- This strange situation is now called **Belady's anomaly**

* *An anomaly in space-time characteristics of certain programs running in a paging machine.* Belady, Nelson and Shedler.

Belady's anomaly: FIFO
Same reference string, different frames

Numbers in **this color**:
No page fault

	0	1	2	3	0	1	4	0	1	2	3	4
Youngest	0	1	2	3	0	1	4	**4**	**4**	2	3	**3**
		0	1	2	3	0	1	**1**	**1**	4	2	**2**
Oldest			0	1	2	3	0	**0**	**0**	1	4	**4**

9 page faults with 3 frames

	0	1	2	3	0	1	4	0	1	2	3	4
Youngest	0	1	2	3	**3**	**3**	4	0	1	2	3	4
		0	1	2	**2**	**2**	3	4	0	1	2	3
			0	1	**1**	**1**	3	3	4	0	1	2
Oldest				0	**0**	**0**	1	2	3	4	0	1

10 page faults with 4 frames

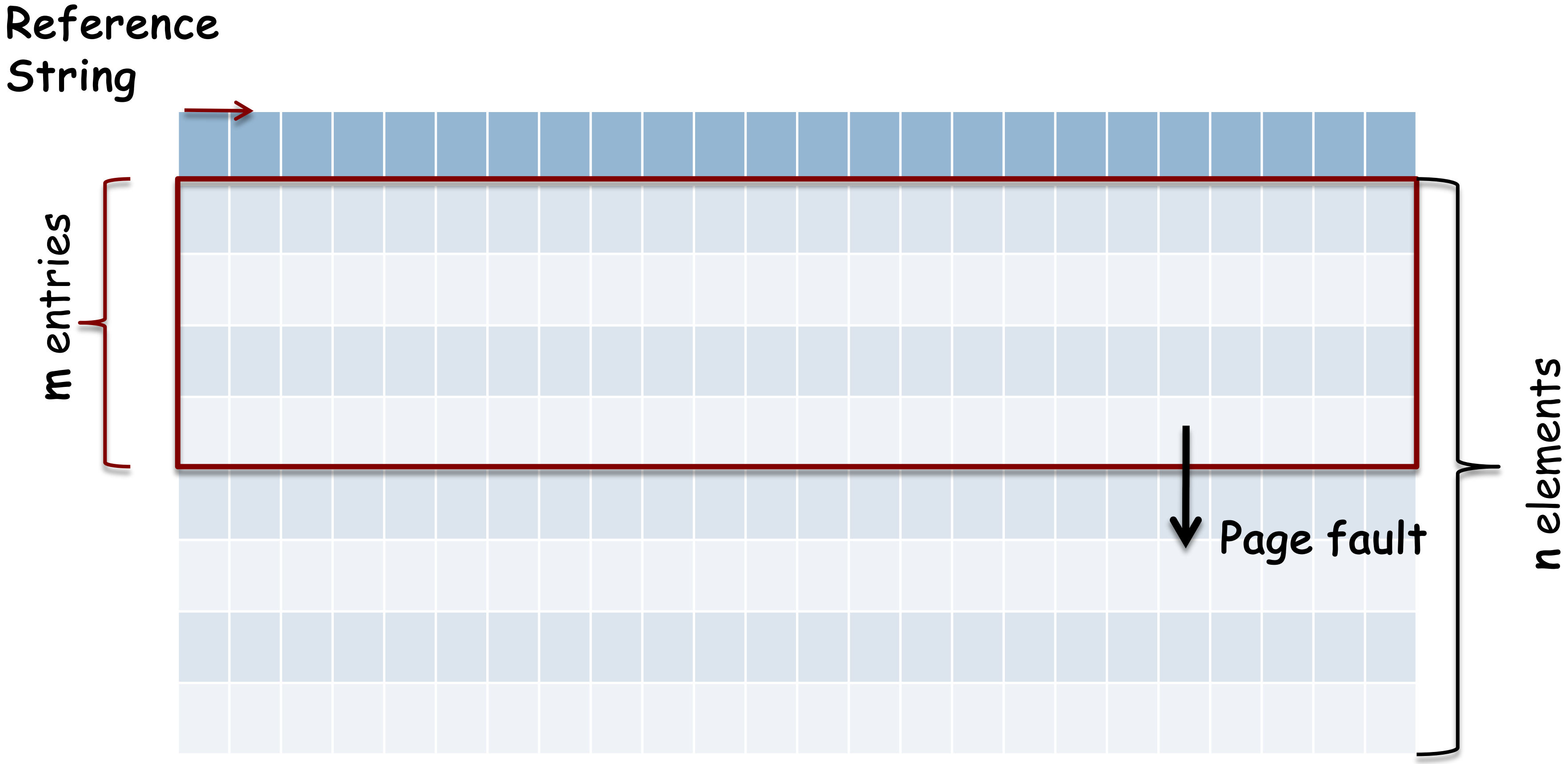
Belady’s anomaly

- Led to a whole theory on paging algorithms and properties
- **Stack algorithms**

The Model

- There is an array ***M***
  - Keeps track of the state of memory
- ***M*** has as many elements as pages of virtual memory
- Divided into two parts
  - Top part: ***m*** entries {Pages currently in memory}
  - Bottom part: ***n-m*** entries
    - Pages that were referenced BUT paged out

The model



Tracking the state of the array M over time

Properties of the model

- When a page is referenced
  - Move to the **top** entry of $\mathbf{M}$
- If the referenced page is already in $\mathbf{M}$
  - All pages above it ***moved down*** one position
  - Pages below it are not moved
- **Transition** from within box to outside of it
  - Page eviction from main memory

The model

0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	4	1
0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	2	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	2	3	4
		0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	1	7	2	3
			0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	3	1	7	2
				0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	5	5	1	7
					0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	4	4	5	5
						0	0	2	2	2	2	2	2	2	2	2	2	2	2	6	6	6	6
								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Properties of the model

- $M(m,r)$
  - The set of pages in the top part of $\mathbf{M}$
  - m page frames
  - r memory references

A property that has some interesting implications

- $M(m, r)$ subset of $M(m+1, r)$

- Set of pages in the top part of $\boldsymbol{M}$ with $\boldsymbol{m}$ frames
  - Also included in $\boldsymbol{M}$ with $\boldsymbol{(m+1)}$ frames

What the subset relationship means

- Execute a process with a set of memory frames
- If we increase memory size by one frame and re-execute **at every point of execution**
  - All pages in the first execution are present in the second run
- Does not suffer from Belady's anomaly
  - **Stack algorithms**

THE OPTIMAL PAGE REPLACEMENT ALGORITHM

The optimal page replacement algorithm

- The best possible algorithm
- Easy to describe but **impossible to implement**
- **Crux:**
  Put off unpleasant stuff for as long as possible
- Idea: evict "Furthest-in-the-future"

The optimal page replacement algorithm description

- When a page fault occurs some set of pages are in memory
- One of these pages will be referenced next
  - Other pages may be not be referenced until 10, 100 or 1000 instructions later
- **Label** each page with the <u>number of instructions</u> to be executed *before* it will be referenced
  - Page with the highest label should be removed

Problem with the optimal page replacement algorithm

- It is **unrealizable**
- During a page fault, OS has no way of knowing ***when*** each of the pages will be referenced next

So why are we looking at it?

- Run a program
  - Track all page references
- Implement optimal page replacement on the second run
  - Based on reference information from the first run
- **Compare** performance of **realizable** algorithms with the best possible one

LRU Page Replacements

The Least Recently Used (LRU) page replacement algorithm

- Approximation of the optimal algorithm
- Observation
  - Pages used heavily in the last few instructions
    - Probably will be used heavily in the next few
  - Pages that have not been used
    - Will probably remain unused for a long time
- When a page fault occurs?
  - Throw out page that has been **unused the longest**

LRU example: 3 memory frames

Reference String

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Recent	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
		7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0
Least Used			7	0	1	2	2	3	0	4	2	2	0	3	3	1	2	0	1	7

Implementing LRU

- Logical clock
- Stacks

Using Logical clocks to implement LRU

- Each page table entry has a **time-of-use** field
  - Entry updated when page is referenced
    - Contents of clock register are copied
- Replace the page with the smallest value
  - Time increases monotonically
    - **Overflows** must be accounted for
- Requires search of page table to find LRU page

Stack based approach

- Keep stack of page numbers
- When page is referenced
  - Move to the top of the stack
- Implemented as a doubly linked list
- No search done for replacement
  - Bottom of the stack is the LRU page

Problems with clock/stack based approaches to LRU replacements

- Inconceivable without hardware support
  - Few systems provide requisite support for true LRU implementations
- Updates of clock fields or stack needed at **every** memory reference
- If we use interrupts and do software updates of data structures things would be very slow
  - Would slow down every memory reference
    - At least 10 times slower

LRU Approximation Page Replacements

LRU Approximation: Reference bit

- **Reference bit** associated with page table entries
- Reference bit is set by hardware when page is referenced
  - Read/write access of the page
- Determine which page has been used and which has not
  - No way of knowing the *order of references* though

LRU Approximation: Additional reference bits

- Maintain 8-bit byte for each page in memory
- OS **shifts** the reference bit for page into the highest order bit of the 8-bit byte
  - Operation performed at *regular intervals*
  - The reference bit is then *cleared*

LRU approximation: Reference bits

Shift Register	Reference bit for the page	Shift Register after the OS timer interrupt
00000000	1	10000000
10010001	1	11001000
01100011	0	00110001

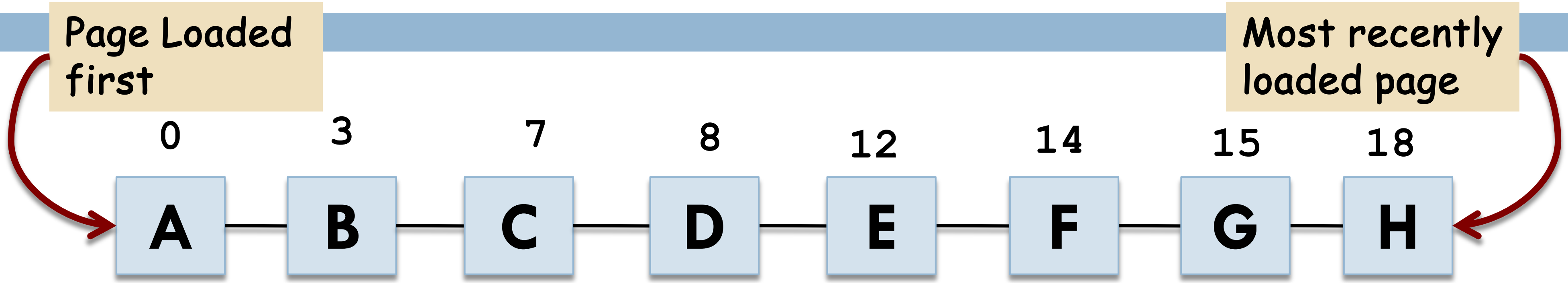
LRU Approximation: Interpreting the reference bits

- Interpret 8-bit bytes as **unsigned integers**
- Page with the lowest number is the LRU page
- 00000000 : Not used in last 8 periods
- 01100101 : Used 4 times in the last 8 periods
- 11000100 used ***more recently*** than 01110111

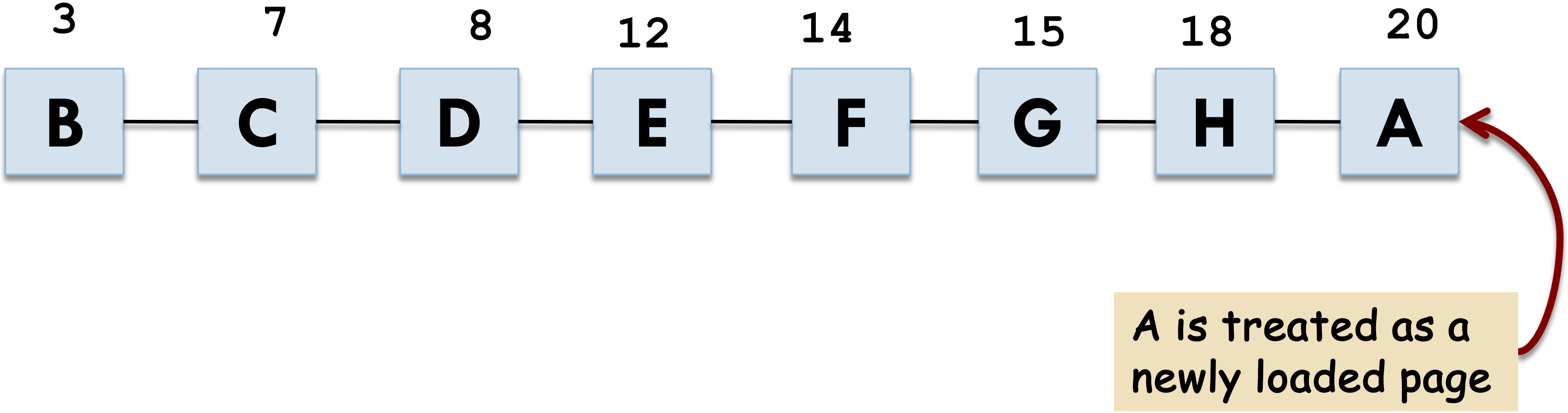
The Second Chance Algorithm

- Simple modification of FIFO
- Avoids throwing out a heavily used page
- Inspect the reference bit of a page
  - If it is **0**: Page is old and unused
    - **Evict**
  - If it is **1**: Page is given a second chance
    - Move page to the end of the list

The Operation of second chance



Page fault occurs at time 20 AND page A's reference bit was set



Second chance

- Reasonable algorithm, but unnecessarily **inefficient**
  - Constantly moving pages around on its list

- Better to keep pages in a circular list
  - In the form of a clock …

Clock Page Replacement

- Keep all frames on a circular list in the form of a clock
  - Hand points to the oldest page
- When a page fault occurs, page being pointed to by the hand is inspected
  - If its R bit is 0:   the page is evicted
    - New page is inserted into the clock in its place
    - Hand is advanced one position
  - If its R bit is 1
    - It is cleared and advanced one position until a page is found with R =0

Counting based page replacements
Most Frequently Used (MFU)

- **Argument**:
  Page with the smallest count was probably just brought in

Summary of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
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
NFU (Not Frequently Used)	Fairly crude approximate to LRU
Aging	Efficient algorithm that approximates LRU well

PAGE BUFFERING ALGORITHMS

Page Buffering

① Maintain a buffer of free frames

② When a page-fault occurs

- Victim frame chosen as before
- Desired page read into free-frame **from buffer**
  - **Before** victim frame is written out
- Process that page-faulted can restart much faster

Page Buffering: Being proactive

- Maintain a list of **modified** pages
- When the paging device is **idle**
  - Write modified pages to disk
- Implications
  - If a page is selected for replacement *increase likelihood* of that page being clean

Page Buffering: Reuse what you can

- Keep pool of free frames as before
  - BUT **remember** which pages they held
- Frame contents are not modified when page is written to disk
- If page needs to come back in?
  - **Reuse** the same frame if it was not used to hold some other page

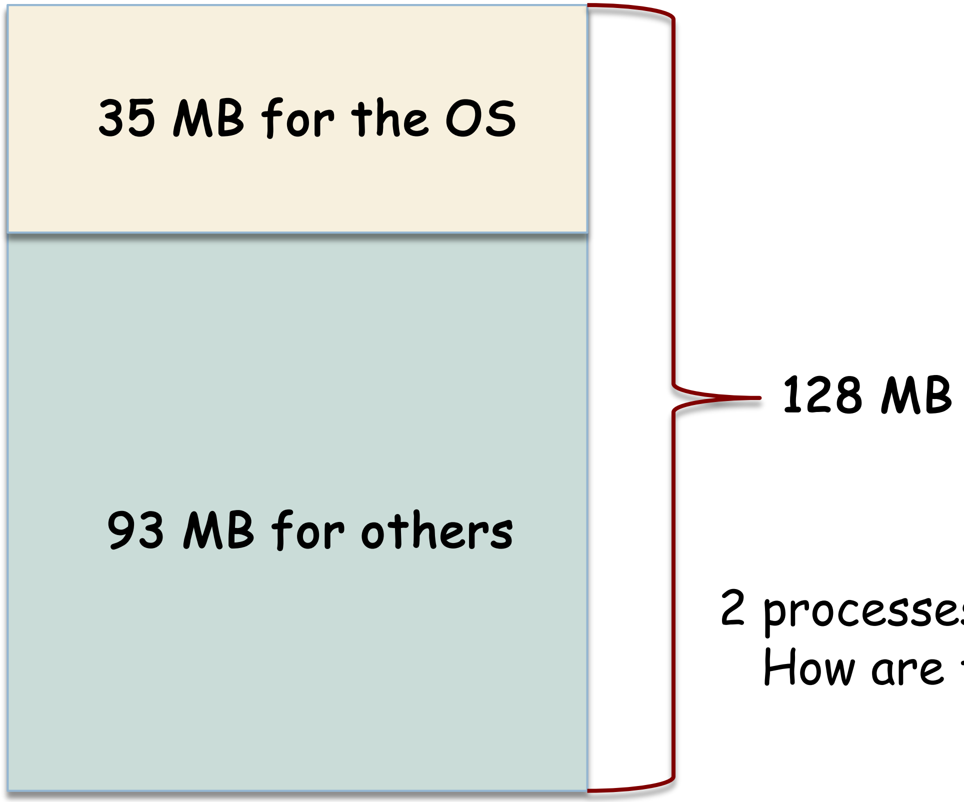
Buffering and applications

- Applications often understand their memory/disk usage <u>better</u> than the OS
  - Provide their own buffering schemes
- If both the OS and the application were to buffer
  - Twice the I/O is being utilized for a given I/O

ALLOCATION OF FRAMES

Frame allocation: How do you divvy up free memory among processes?

Frame size = 1 MB; Total Size = 128 MB



2 processes at T_0

How are frames allocated?

With demand paging all 93 frames would be in the free frame pool

Constraints on frame allocation

- **Max**: Total number of frames in the system
  - Available physical memory
- **Min**: Need to allocate at least a minimum number of frames
  - Defined by the architecture of the underlying system

Minimum number of frames

- As you decrease the number of frames for a process
  - Page fault increases
  - Execution time increases too

- Defined by the **architecture**
  - In some cases instructions and operands (indirect references) straddle page boundaries
    - With 2 operands at least 6 frames needed

Frame Allocation Policies

Global vs Local Allocation

- Global replacement
  - One process can **take** a memory frame from another process
- Local replacement
  - Process can only choose from the set of frames that was allocated to it

Local vs Global replacement: Based on how often a page is referenced

Pages	Usage Count
A1	10
A2	7
A3	5
A4	3
B1	9
B2	4
B3	2
B4	6
C1	3
C2	5
C3	6

Processes A, B and C

Pages
A1
A2
A3
A5
B1
B2
B3
B4
C1
C2
C3

Local Replacement

Pages
A1
A2
A3
A4
B1
B2
A5
B4
C1
C2
C3

Global Replacement

Process ***A*** has page faulted and needs to bring in a page

Global vs Local Replacement

	Local	Global
Number of frames allocated to process	Fixed	Varies dynamically
Can process control its own fault rate?	YES	NO
Can it use free frames that are available?	NO	YES
Increases system throughput?	NO	YES

WORKING SETS & THRASHING

Locality of References

- During any phase of execution a process references a relatively small **fraction** of its pages
- Set of pages that a process is currently using
  - **Working set**
- Working set **evolves** during process execution

Implications of the working set

- If the entire working set is in memory
  - Process will execute without causing many faults
    - Until it moves to *another phase* of execution

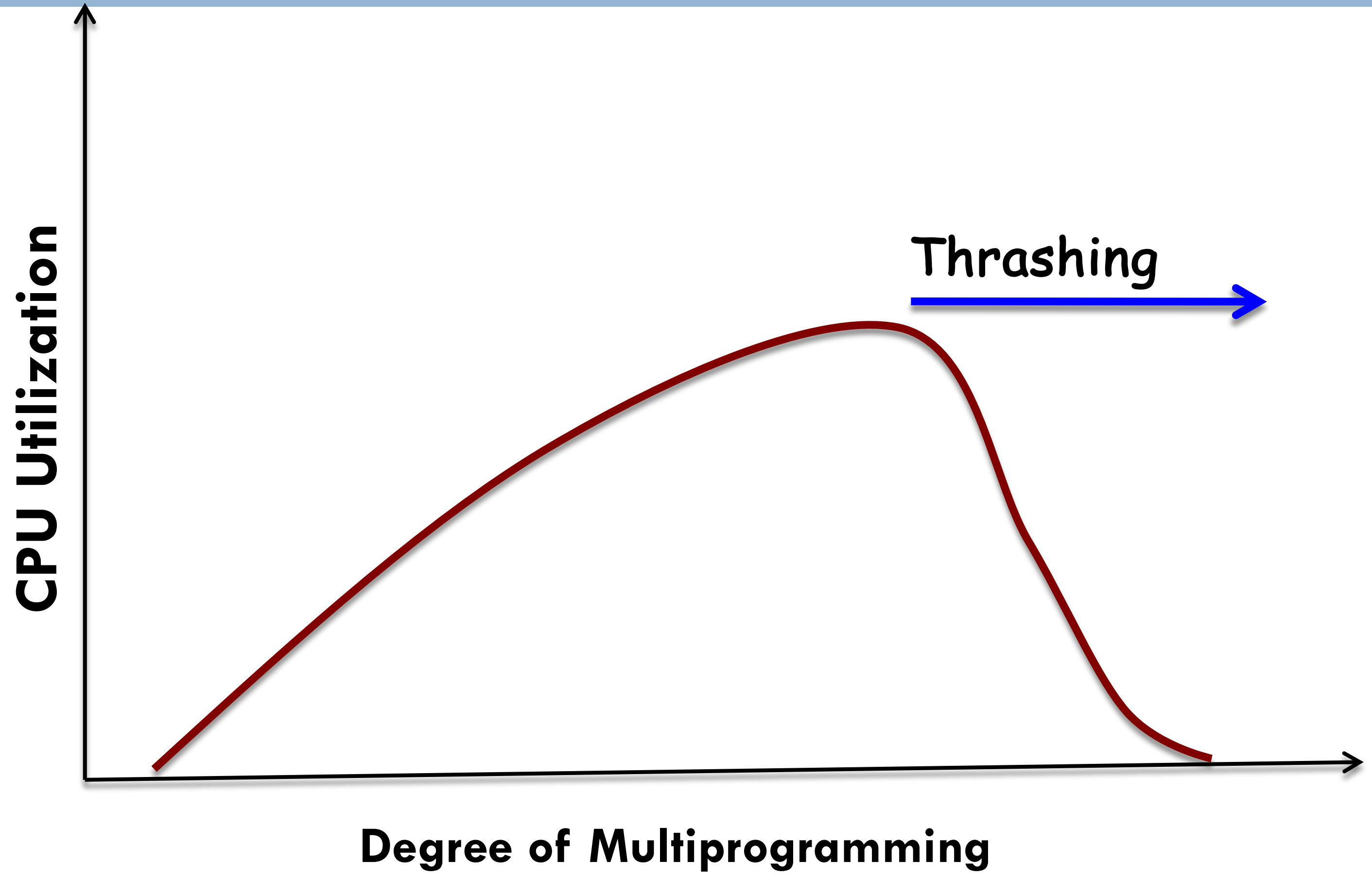
- If the available memory is too small to hold the working set?
  1. Process will cause many faults
  2. Run very slowly

A program causing page faults every few instructions is said to be thrashing

- System throughput **plunges**
  - Processes spend all their time paging

- Increasing the degree of multiprogramming can cause this
  - New process may **steal** frames from another process {*Global Replacement*}
    - Overall page-faults in the system increases

Characterizing the affect of multiprogramming on thrashing



Mitigating the effects of thrashing

- Using a local page replacement algorithm
  - One process thrashing does not cause **cascading thrashing** among other processes
  - BUT if a process is thrashing
    - Average service time for a page fault increases
- Best approach
  1. Track a process' working set
  2. Make sure the working set is in memory **before** you let it run

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 9]
- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 3]*