CS 370: OPERATING SYSTEMS

[VIRTUAL MEMORY]

Computer Science
Colorado State University

** Lecture slides created by: SHRIDEEP PALICKARA

Instructor: Louis-Noel Pouchet
Spring 2024
Topics covered in this lecture

- FIFO Page Replacement Algorithm
- Belady’s Anomaly
- Stack Algorithms
- Page Buffering
- Frame Allocations
How we got here …

Contiguous Memory → External Fragmentation → Pure Paging → Low Degree of Multiprogramming

Demand Paging

Page Faults

Page replacement algorithms → Page Bufferring → Frame Allocation

Working Sets
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

<table>
<thead>
<tr>
<th>Reference String</th>
<th>Youngest</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Youngest</th>
<th>0 1 2 3 0 1 4 0 1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oldest</td>
<td>0 1 2 3 0 0 0 1 4 4</td>
</tr>
</tbody>
</table>

Numbers in this color: No page fault
9 page faults with 3 frames

<table>
<thead>
<tr>
<th>Youngest</th>
<th>0 1 2 3 3 3 4 0 1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oldest</td>
<td>0 1 2 2 2 3 4 0 1 2</td>
</tr>
</tbody>
</table>

10 page faults with 4 frames

<table>
<thead>
<tr>
<th>Youngest</th>
<th>0 1 1 1 3 3 4 0 1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oldest</td>
<td>0 0 0 1 2 3 4 0 1</td>
</tr>
</tbody>
</table>
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

Reference String

$\text{Tracking the state of the array } M \text{ over time}$

Page fault

$m$ entries

$n$ elements

$n$ elements

$m$ entries

$\text{Reference String}$

$\text{Page fault}$

$\text{Tracking the state of the array } M \text{ over time}$
Properties of the model

- When a page is referenced
  - Move to the top entry of $M$

- If the referenced page is already in $M$
  - 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
Properties of the model

- $M(m,r)$
  - The set of pages in the top part of $M$
  - $m$ 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 $M$ with $m$ frames
  - Also included in $M$ with $(m+1)$ 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 **number of instructions** 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

<table>
<thead>
<tr>
<th>Reference String</th>
<th>Recent</th>
<th>Least Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td>7 0 1 2 2 3 0 4 2 2 0 3 3 1 2 0 1 7</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Shift Register</th>
<th>Reference bit for the page</th>
<th>Shift Register after the OS timer interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>1</td>
<td>10000000</td>
</tr>
<tr>
<td>10010001</td>
<td>1</td>
<td>11001000</td>
</tr>
<tr>
<td>01100011</td>
<td>0</td>
<td>00110001</td>
</tr>
</tbody>
</table>
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

A is treated as a newly loaded page
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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude approximation of LRU</td>
</tr>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out important pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but difficult to implement</td>
</tr>
<tr>
<td>NFU (Not Frequently Used)</td>
<td>Fairly crude approximate to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm that approximates LRU well</td>
</tr>
</tbody>
</table>
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 better 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

35 MB for the OS

93 MB for others

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

<table>
<thead>
<tr>
<th>Pages</th>
<th>Usage Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>A2</td>
<td>7</td>
</tr>
<tr>
<td>A3</td>
<td>5</td>
</tr>
<tr>
<td>A4</td>
<td>3</td>
</tr>
<tr>
<td>B1</td>
<td>9</td>
</tr>
<tr>
<td>B2</td>
<td>4</td>
</tr>
<tr>
<td>B3</td>
<td>2</td>
</tr>
<tr>
<td>B4</td>
<td>6</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
</tr>
<tr>
<td>C2</td>
<td>5</td>
</tr>
<tr>
<td>C3</td>
<td>6</td>
</tr>
</tbody>
</table>

Processes A, B and C

<table>
<thead>
<tr>
<th>Pages</th>
<th>Usage Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
</tbody>
</table>

Local Replacement

<table>
<thead>
<tr>
<th>Pages</th>
<th>Usage Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
</tbody>
</table>

Global Replacement

<table>
<thead>
<tr>
<th>Pages</th>
<th>Usage Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
</tbody>
</table>

Process A has page faulted and needs to bring in a page.
Global vs Local Replacement

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of frames allocated to process</td>
<td>Fixed</td>
<td>Varies dynamically</td>
</tr>
<tr>
<td>Can process control its own fault rate?</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Can it use free frames that are available?</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Increases system throughput?</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
WORKING SETS & THRASHING
Locality of References

- During any phase of execution a process references a relatively small \textit{fraction} of its pages

- Set of pages that a process is currently using
  - Working set

- Working set \textit{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

![Graph showing CPU utilization vs. degree of multiprogramming with a peak representing thrashing.](image)

CPU Utilization

Degree of Multiprogramming

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
