Frequently asked questions from the previous class survey

- Finding an empty frame: How long does this typically take?
- When a process swap takes place between the disk and the RAM, is the OS selective about which pages it copies over? Are there redundant copies?
- How does an OS guess which pages will be used?
Topics covered in this lecture

- Page replacement algorithms
- Page Buffering
- Frame Allocations
- Working Sets
- TLB Reach

How we got here ...

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

Demand Paging → Page Faults → Working Sets

Page replacement algorithms → Page Buffering → Frame Allocation
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
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
  - When there is a page-fault, the 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

**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: 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

<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>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 [Multiple reference bits]</td>
<td>Efficient algorithm that approximates LRU well</td>
</tr>
</tbody>
</table>
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
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 T₀
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**: We 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>

Local Replacement

Global Replacement

Processes A, B and C have page faults and need to bring in pages.

Global vs Local Replacement

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of frames</td>
<td>Fixed</td>
<td>Varies</td>
</tr>
<tr>
<td>allocated to process</td>
<td></td>
<td>dynamically</td>
</tr>
<tr>
<td>Can process control its</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>own fault rate?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can it use free frames</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>that are available?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Increases system</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>throughput?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 \( \text{Global Replacement} \)
  - 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
Working sets & Thrashing

Working set is an approximation of the program’s locality

- Most important property of the working set is its size

Page reference table
\[ \begin{array}{cccccccccccccccccccccccccccc}
\ldots & 2 & 6 & 1 & 5 & 7 & 7 & 7 & 5 & 1 & 6 & 2 & 3 & 4 & 4 & 3 & 4 & 3 & 4 & 4 & 1 & 2 & 3 & 4 & 8 \\
\end{array} \]

\[ \text{WS} = \{1, 2, 5, 6, 7\} \quad \text{WS} = \{3, 4\} \]

- \( WSS_i \) = Working set size for process \( p_i \)
- If total demand exceeds available frames
  - Thrashing will occur
Working sets and page fault rates

- The peak in page-fault rate happens when a new locality is being demand-paged.
- Once working set is in memory:
  - Page fault rate falls.
- When process moves towards a new working set window?
  - Fault rate rises again.

The page fault frequency approach to reducing thrashing

- When the page fault rate is high:
  - Process needs more frames.
- When the page fault rate is too low:
  - Process may have too many frames.
Using page fault frequencies to control thrashing: Establish bounds

Using page fault frequencies to control thrashing: Establish bounds

Using page fault frequencies to control thrashing: Establish bounds

Using page fault frequencies to control thrashing: Establish bounds

OTHER CONSIDERATIONS
Prepaging: Loading pages BEFORE letting a process run

- Bring into memory -- at one time -- all the pages that will be needed
  - Prepage frames for small files

- With the working set model
  - Ensure that the entire working set is in memory before the process is resumed

TLB Reach is the amount of memory accessible from the TLB

- TLB-Reach = Number of TLB entries x Page Size

- Approaches to increasing TLB reach
  - Double the entries
    - Expensive
  - Increase page size
    - Increases (internal) fragmentation
  - Support multiple page sizes
    - OS not hardware manages the TLB
    - Increase reach and hit ratio
      - Current trend
Select data structures and program structures efficiently

- Increase locality
  - Reduce page fault rates

- Loops
  - If data is stored in row-major format, but program reads it as column-major format

- Loader should avoiding placing routines across page boundaries

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