Page Replacement Algorithms

- MIN, OPT (optimal)
- RANDOM
  - evict random page
- FIFO (first-in, first-out)
  - give every page equal residency
- LRU (least-recently used)
- MRU (most-recently used)
First-In-First-Out (FIFO) Algorithm

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)
  
  ![Page fault Illustration]

  - 15 page faults
  - Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
    - Adding more frames can cause more page faults!
      - **Belady’s Anomaly**
  - How to track ages of pages?
    - Just use a FIFO queue
FIFO Illustrating Belady’s Anomaly

The graph shows the relationship between the number of page faults and the number of frames.

- The y-axis represents the number of page faults, ranging from 2 to 16.
- The x-axis represents the number of frames, ranging from 1 to 7.

The graph illustrates how the number of page faults decreases as the number of frames increases, highlighting the FIFO page replacement algorithm's behavior in the context of Belady's Anomaly.
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs

reference string

7 0 1 2 0 3 0 4 2 3 0 1 2 0 1 7 0 1

page frames

7 7 7 2 2 2 2 7
0 0 0 0 4 0 0 0
1 1 3 3 3 1 1
Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

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

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?
LRU Algorithm (Cont.)

- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to find smallest value.
    - Search through table needed.

- **Stack implementation**
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - Move it to the top.
    - Requires 6 pointers to be changed.
  - But each update more expensive.
  - No search for replacement.

- LRU and OPT are cases of **stack algorithms** that don’t have Belady’s Anomaly.
Use Of A Stack to Record Most Recent Page References

reference string

4  7  0  7  1  0  1  2  1  2  7  1  2

stack before a

2
1
0
7
4

stack after b

7
2
1
0
4

a  b
LRU Approximation Algorithms

- **LRU needs special hardware and still slow**

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 \(\rightarrow\) replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules
Second-Chance (clock) Page-Replacement Algorithm
Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert

- Take ordered pair (reference, modify)
  1. (0, 0) neither recently used not modified – best page to replace
  2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
  3. (1, 0) recently used but clean – probably will be used again soon
  4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times
Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common

- **Lease Frequently Used (LFU) Algorithm**: replaces page with smallest count

- **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected
Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - Raw disk mode
- Bypasses buffering, locking, etc
Allocation of Frames

- Each process needs \textit{minimum} number of frames
  - Defined by the computer architecture
- \textbf{Maximum} of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations
Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool

- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change
    - $s_i = \text{size of process } p_i$
    - $S = \sum s_i$
    - $m = \text{total number of frames}$
    - $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

\[
\begin{align*}
& m = 64 \\
& s_1 = 10 \\
& s_2 = 127 \\
& a_1 = \frac{10}{137} \times 62 \approx 4 \\
& a_2 = \frac{127}{137} \times 62 \approx 57
\end{align*}
\]
Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process $P_i$ generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common

- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** ⇔ a process is busy swapping pages in and out
Thrashing (Cont.)
Demand Paging and Thrashing

- Why does demand paging work?
  **Locality model**
  - A locality is a set of pages actively used together
  - Process migrates from one locality to another
  - Localities may overlap
  - Localities are defined by the program structure and its data structure

- Why does thrashing occur?
  \[ \sum \text{size of locality} > \text{total memory size} \]
  - Limit effects by using local or priority page replacement
Locality In A Memory-Reference Pattern
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  - Example: 10,000 instructions
- $WSS_i$ (working set of Process $P_i$) = total number of pages referenced in the most recent $\Delta$ (varies in time)
  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

Page reference table

- $WS(t_1) = \{1, 2, 5, 6, 7\}$
- $WS(t_2) = \{3, 4\}$
Approximate the working-set model with interval timer + a reference bit

Example: $\Delta = 10,000$

- Timer interrupts after every 5000 time units
- Keep in memory 2 bits for each page
- Whenever a timer interrupts copy and sets the values of all reference bits to 0
- If one of the bits in memory $= 1 \Rightarrow$ page in working set

Why is this not completely accurate?

Improvement = 10 bits and interrupt every 1000 time units
Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” page-fault frequency (PFF) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

![Graph showing the relationship between page-fault rate and number of frames]
Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time