CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Fall 2017 Lecture 23

Virtual memory

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
FAQ

• Is a page replaces when the process is finished with it?
• Dirty bit: indicates that a page has been modified, hence no longer a copy of the page on the disk.
• Why LRU and OPT not affected by Belady’s anomaly? Stack type. When frames are less they are a subset of pages when frame are more.
• Is OPT used in real world?
• When Belady’s Anomaly is present in an algorithm, will adding an additional frame always cause more page faults?
• More RAM means less frequent need for replacement?
• Paging algorithms seem to assume both temporal and spatial locality... how true is this?
Page Replacement Algorithms

Algorithms

• FIFO
• “Optimal”
• The Least Recently Used (LRU)
  – Exact Implementations
    • Time of use field, Stack
  – Approximate implementations
    • Reference bit
    • Reference bit with shift register
    • Second chance: clock
    • Enhanced second chance: dirty or not?

• Other
Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time (4th access – page 7 is least recently used ...)
- Associate time of last use with each page

12 faults – better than FIFO (15) but worse than OPT (9)
- Generally good algorithm and frequently used
- But how to implement it by tracking the page usage?
LRU Algorithm: Implementations

Possible implementations

• 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
  – Each update expensive
  – No search for replacement needed (bottom is least recently used)

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

Least recently used

Too slow if done in software
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)
    • 0 implies not used since initialization
    • We do not know the order, however.

• Advanced schemes using more bits: preserve more information about the order
Ref bit + history shift register

LRU approximation
Ref bit: 1 indicates used, Shift register records history

<table>
<thead>
<tr>
<th>Ref Bit</th>
<th>Shift Register</th>
<th>Shift Register after OS timer interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0000 0000</td>
<td>1000 0000</td>
</tr>
<tr>
<td>1</td>
<td>1001 0001</td>
<td>1100 1000</td>
</tr>
<tr>
<td>0</td>
<td>0110 0011</td>
<td>0011 0001</td>
</tr>
</tbody>
</table>

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

• **Second-chance algorithm**
  
  – Generally FIFO, plus hardware-provided reference bit
  
  – Avoid throwing out a heavily used page
  
  – **Clock** replacement (using circular queue): hand as a pointer
  
  – Consider next page
    
    • Reference bit = 0 -> replace it
    
    • reference bit = 1 then:
      
      – set reference bit 0, leave page in memory
      
      – consider next page, subject to same rules
Second-Chance (clock) Page-Replacement Algorithm

- **Clock replacement**: hand as a pointer
- Consider next page
  - Reference bit = 0 -> replace it
  - Reference bit = 1 then:
    - set reference bit 0, leave page in memory
    - consider next page, subject to same rules
      - (a) change to 0
      - (b) replace page
Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
  clean page: better replacement candidate
- 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

- Least 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
Clever Techniques for enhancing Perf

• Keep a buffer (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

• Keep list of modified pages
  – When backing store is otherwise idle, write pages there and set to non-dirty (being proactive!)

• Keep free frame previous 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
Some applications (like databases) 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
- OS may provide “raw access” disk to special programs without file system services.
Allocation of Frames
How to allocate frames to processes?

- Each process needs *minimum* number of frames
  Depending on specific needs of the process
- *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 (need based)
  – Dynamic as degree of multiprogramming, process sizes change

\[
s_i = \text{size of process } p_i
\]
\[
S = \sum s_i
\]
\[
m = \text{total number of frames}
\]
\[
a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m
\]

\[
m = 64
\]
\[
s_1 = 10
\]
\[
s_2 = 127
\]
\[
a_1 = \frac{10}{137} \quad 62 \quad 4
\]
\[
a_2 = \frac{127}{137} \quad 62 \quad 57
\]
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 or
  – 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
Problem: 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, leading to
    • Operating system thinking that it needs to increase the degree of multiprogramming leading to
    • Another process added to the system

• Thrashing ≡ a process is busy swapping pages in and out
Thrashing (Cont.)

The diagram illustrates the relationship between CPU utilization and the degree of multiprogramming, highlighting the phenomenon of thrashing. As the degree of multiprogramming increases, the CPU utilization also increases until a point where the system experiences thrashing, indicated by the graph approaching and then dropping significantly.
Demand Paging and Thrashing

• Why does demand paging work?
  **Locality model**
  – Process migrates from one locality to another
  – Localities may overlap

• Why does thrashing occur?

  \[ \sum \text{size of locality} > \text{total memory size} \]

  – Limit effects by using local or priority page replacement
Working-Set Model

- \( \Delta \equiv \text{working-set window} \equiv \text{a fixed number of page references} \)
  - Example: 10,000 instructions

\[ \Delta = 10 \]

**Page reference table**

\[ \ldots 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 \ldots \]

- 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
- Approximation of locality

\( \text{WS}(t_1) = \{1,2,5,6,7\} \quad \text{WS}(t_2) = \{3,4\} \)

- \( WSS_i \) (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
  - Approximation of locality

- \( D = \sum WSS_i \equiv \text{total demand frames} \)
  - If \( D > m \) \( \Rightarrow \) Thrashing
  - **Policy** if \( D > m \), then suspend or swap out one of the processes
Page-Fault Frequency Approach

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

Peaks occur at locality changes: 3 working sets
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- File is then in memory instead of disk
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when the pager scans for dirty pages
Memory Mapped Files

Disk File uses 6 blocks
Page tables used for mapping
Allocating Kernel Memory
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
    - Process descriptors, semaphores, file objects etc.
    - Often much smaller than page size
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O
  - approaches (skipped)
Other Considerations
Other Considerations -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume $s$ pages are prepaged and $\alpha$ of the pages is used
    - Is cost of $s \times \alpha$ saved pages faults $>$ or $<$ than the cost of prepaging $s \times (1 - \alpha)$ unnecessary pages?
    - $\alpha$ near zero $\Rightarrow$ greater prepaging loses
Other Issues – Page Size

• Sometimes OS designers have a choice
  – Especially if running on custom-built CPU

• Page size selection must take into consideration:
  – Fragmentation
  – Page table size
  – Resolution
  – I/O overhead
  – Number of page faults
  – Locality
  – TLB size and effectiveness

• Always power of 2, usually in the range $2^{12}$ (4,096 bytes) to $2^{22}$ (4,194,304 bytes)

• On average, growing over time
Page size issues – TLB Reach

• TLB Reach - The amount of memory accessible from the TLB

• TLB Reach = (TLB Size) X (Page Size)

• Ideally, the working set of each process is stored in the TLB
  – Otherwise there is a high degree of page faults

• Increase the Page Size
  – This may lead to an increase in fragmentation as not all applications require a large page size

• Provide Multiple Page Sizes
  – This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation
Other Issues – Program Structure

• Program structure
  – int[128,128] data;  i: row, j: column
  – Each row is stored in one page
  – Program 1
    
    ```c
    for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
    ```

    128 x 128 = 16,384 page faults

  – Program 2 inner loop = 1 row = 1 page
    
    ```c
    for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
    ```

    128 page faults
Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory.

- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

- **Pinning** of pages to lock into memory.
Virtual Memory

- Address spaces and demand paging
- Page Replacement Algorithms
- Page Buffering
- Frame Allocation
- Page size issues
Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum