Frequently asked questions from the previous class survey

- Page fault: How do you know? Is it a bad thing?
- FIFO: What if the oldest page is the most accessed one?
- Difference between: Pure paging, demand paging, and pure demand paging

Topics covered in this lecture

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

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 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

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

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>01100111</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 Loaded first
Most recently loaded page

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 algorithms that approximate 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
   - 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
- 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

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

- **Global Replacement**
  - Processes A, B, and C

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

**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

**WORKING SETS & THRASHING**
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 effect of multiprogramming on thrashing

![Graph showing CPU Utilization vs. Degree of Multiprogramming]

- 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

Mitigating the effects of thrashing

- Best approach
  1. Track a process’ working set
  2. Make sure the working set is in memory before you let it run

Working set is an approximation of the program’s locality

- Most important property of the working set is **size**

<table>
<thead>
<tr>
<th>Page reference table</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 6 1 5 1 7 7 5 6 2 3 4 4 3 4 4 4 1 2 3 4</td>
</tr>
</tbody>
</table>

- **WSi** = Working set size for process *pi*

- 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

- Increase number of frames
  - Upper Bound
- Decrease number of frames
  - Lower Bound

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