Memory Management II

Virtual Memory
Virtual memory idea

- OS and hardware produce illusion of disk as fast as main memory, or main memory as large as disk

- Process runs when not all pages are loaded in memory
  - Only keep referenced pages in main memory
  - Keep unreferenced pages on slower, cheaper backing store (disk)
  - Bring pages from disk to memory when necessary
Page table with virtual memory

Virtual Memory

Page 0
Page 1
Page 2
Page 3

Physical Memory

Page 0
Page 1

Page 2
Page 3

Disk

Page table
Handling page fault by demand paging

1. Trap
2. Bring in missing page
3. Page is on backing store
4. Physical memory
5. Reset page table
6. Restart instruction

Load M

Operating system

Reference
Page fault handler

- Handles both swapped-out pages and illegal access

- Illegal access
  - User mode accessing kernel space
  - Write access on read-only region
  - SIGSEGV or possibly Copy-On-Write

- Legal but page currently swapped out
  - Demand paging
Paging strategies

- **Demand paging**: load page on page fault
  - Process starts with no pages loaded

- **Request paging**: user specifies which pages are needed
  - Requires users to manage memory by hand

- **Pre-paging**: load page before it is referenced
  - When one page is referenced, bring in next one
Working set

- With pure demand paging:

- Pre-paging tries to smooth out bursts
Thrashing

- What if we need more pages regularly than we have?
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back

- Leads to:
  - High page fault rate
  - Lots of I/O wait
  - Low CPU utilization
  - No useful work done

- Thrashing: system busy just swapping pages
Page replacement

- When no free pages available, must select victim page in memory and throw it out to disk

- Page replacement algorithms
  - **Optimal**: throw out page that won’t be used for longest time in future
  - **Random**: throw out a random page
  - **FIFO**: throw out page that was loaded in first
  - **LRU**: throw out page that hasn’t been used in longest time
Evaluating page replacement algorithms

- **Goal**: fewest number of page faults

- **A method**: run algorithm on a particular string of memory references (reference string) and computing the number of page faults on that string

- **In all our examples, the reference string is**

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Optimal algorithm

- Throw out page that won’t be used for longest time in future

6 page faults

Problem: difficult to predict future!
Fist-In-First-Out (FIFO) algorithm

- Throw out page that was loaded in first

1 2 3 4 1 2 5 1 2 3 4 5
1 1 1 1 1 5 5 5 5 5 4 4
2 2 2 2 2 1 1 1 1 1 5
3 3 3 3 3 2 2 2 2 3 3
4 4 4 4 4 4 4 3 3 3

10 page faults

Problem: ignores access patterns
Fist-In-First-Out (FIFO) algorithm (cont.)

- Results with 3 physical pages

9 page faults

Problem: fewer physical pages ➔ fewer faults!

➔ Known as Belady’s Anomaly
Ideal curve of # of page faults v.s. # of physical pages

![Graph showing the ideal curve of number of page faults vs. number of physical pages. The curve starts high at one frame and gradually decreases as the number of frames increases, approaching a constant value.](image)
Belady’s Anomaly in FIFO algorithm

![Graph showing number of page faults vs number of frames]
Least-Recently-Used (LRU) algorithm

- Throw out page that hasn’t been used in longest time. Can use FIFO to break ties

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

8 page faults

Advantage: with locality, LRU approximates Optimal
Implementing LRU: hardware

- A counter for each page

- Every time page is referenced, save system clock into the counter of the page

- Page replacement: scan through pages to find the one with the oldest clock

- **Problem**: have to search all pages!
Implementing LRU: software

- A doubly linked list of pages
- Every time page is referenced, move it to the front of the list
- Page replacement: remove the page from back of list  
  - Avoid scanning of all pages
- **Problem**: too expensive  
  - Requires 6 pointer updates for each page reference  
  - High contention on multiprocessor
LRU: concept vs. reality

- LRU is considered to be a reasonably good algorithm.

- Problem is in implementing it efficiently.

- In practice, settle for efficient approximate LRU:
  - Find a not recently accessed page, but not necessarily the least recently accessed.
  - LRU is approximation anyway, so why not approximate even more.
Clock (second-chance) algorithm

- **Goal**: remove a page that has not been referenced recently
  - good LRU-approximate algorithm

- **Idea**
  - A reference bit per page
  - Memory reference: hardware sets bit to 1
  - Page replacement: OS finds a page with reference bit cleared
  - OS traverses all pages, clearing bits over time
Clock algorithm implementation

- If ref bit is 1, clear it, and advance hand
- Else return current page as victim
Clock algorithm example

10 page faults

Advantage: simple to implement!
Clock algorithm extension

- Problem of clock algorithm: does not differentiate dirty v.s. clean pages

- Dirty page: pages that have been modified and need to be written back to disk
  - More expensive to replace dirty than clean pages
  - One extra disk write (about 5 ms)
Clock algorithm extension (cont.)

- Use **dirty** bit to give preference to dirty pages

- On page reference
  - Read: hardware sets reference bit
  - Write: hardware sets dirty bit

- Page replacement
  - reference = 0, dirty = 0 → victim page
  - reference = 0, dirty = 1 → skip (don’t change)
  - reference = 1, dirty = 0 → reference = 0, dirty = 0
  - reference = 1, dirty = 1 → reference = 0, dirty = 1
  - advance hand, repeat
  - If no victim page found, run swap daemon to flush unreferenced dirty pages to the disk, repeat
Summary of page replacement algorithms

- **Optimal**: throw out page that won’t be used for longest time in future
  - Best algorithm if we can predict future
  - Good for comparison, but not practical
- **Random**: throw out a random page
  - Easy to implement
  - Works surprisingly well. Why? Avoid worst case
  - **Cons**: random
- **FIFO**: throw out page that was loaded in first
  - Easy to implement
  - Fair: all pages receive equal residency
  - Ignore access pattern
- **LRU**: throw out page that hasn’t been used in longest time
  - Past predicts future
  - With locality: approximates Optimal
  - Simple approximate LRU algorithms exist (Clock)
Current trends in memory management

- Virtual memory is less critical now
  - Personal computer v.s. time-sharing machines
  - Memory is cheap ➔ Larger physical memory

- Virtual to physical translation is still useful
  - “All problems in computer science can be solved using another level of indirection” David Wheeler

- Larger page sizes (even multiple page sizes)
  - Better TLB coverage
  - Smaller page tables, less page to manage
  - Internal fragmentation: not a big problem

- Larger virtual address space
  - 64-bit address space
  - Sparse address spaces

- File I/O using the virtual memory system
  - Memory mapped I/O: mmap()