Memory Management II

Virtual Memory
Virtual memory motivation

- Previous approach to memory management
  - Must completely load user process in memory
  - One large AS or too many AS → out of memory

- Observation: locality of reference
  - Temporal: access memory location accessed just now
  - Spatial: access memory location adjacent to locations accessed just now

- Implication: process only needs a small part of address space at any moment!
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
Virtual memory illustration

Virtual Memory:
- Page 0
- Page 1
- Page 2
- Page 3

Page table:
- Page 0
- Page 1
- Page 2
- Page 3

Physical Memory:
- Page 0
- Page 1

Disk:
- Page 2
- Page 3
Detect reference to page on disk and recognize disk location of page

- Overload the present bit of page table entries

- If a page is on disk, clear present bit in corresponding page table entry and store disk location using remaining bits

- **Page fault**: if bit is cleared then referencing resulting in a trap into OS

- In OS **page fault handler**, check page table entry to detect if page fault is caused by reference to true invalid page or page on disk
Steps in handling a page fault

1. Trap
2. Bring in missing page
3. Page is on backing store
4. Reset page table
5. Free frame
6. Restart instruction
OS decisions

- **Page selection**
  - When to bring pages from disk to memory?

- **Page replacement**
  - When no free pages available, must select *victim* page in memory and throw it out to disk
Page selection algorithms

- **Demand paging**: load page on page fault
  - Start up process with no pages loaded
  - Wait until a page absolutely must be in memory

- **Request paging**: user specifies which pages are needed
  - Requires users to manage memory by hand
  - Users do not always know best
  - OS trusts users (e.g., one user can use up all memory)

- **Prepaging**: load page before it is referenced
  - When one page is referenced, bring in next one
  - Do not work well for all workloads
    - Difficult to predict future
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 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

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

10 page faults

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

- Results with 3 physical pages

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

9 page faults

Problem: fewer physical pages $\Rightarrow$ fewer faults!

belady anomaly
Ideal curve of # of page faults v.s. # of physical pages

number of frames

number of page faults

1 2 3 4 5 6

1 2 3 4 5 6
FIFO illustrating belady’s anomaly
Least-Recently-Used (LRU) algorithm

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

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

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/counters!
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
  - Hardware implementation: counter per page, copied per memory reference, have to search pages on page replacement to find oldest
  - Software implementation: no search, but pointer swap on each memory reference, high contention

- 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 approximate 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

- Combining FIFO with LRU: give the victim page that FIFO selects a **second chance**

- Keep pages in a circular list = **clock**

- Pointer to next victim = clock **hand**

- To replace a page, OS examines the page pointed to by hand
  - If ref bit == 1, clear, advance hand
  - Else return current page as victim
A single step in Clock algorithm
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()
Backup Slides
Dynamic allocation issue: fragmentation

- **Fragment**: small trunk of free memory ("holes"), too small for future allocation requests
  - **External fragment**: visible to system
  - **Internal fragment**: visible to process (e.g. if allocate at some granularity)

- **Goal**
  - Reduce number of holes
  - Keep holes large

- **Stack fragmentation vs heap fragmentation**
Typical heap implementation

- Data structure: free list
  - Chains free blocks together

- Allocation
  - Choose block large enough for request
  - Update free list

- Free
  - Add block back to list
  - Merge adjacent free blocks
Heap allocation strategies

- **Best fit**
  - Search the whole list on each allocation
  - Choose the smallest block that can satisfy request
  - Can stop search if exact match found

- **First fit**
  - Choose first block that can satisfy request

- **Worst fit**
  - Choose largest block (most leftover space)

Which is better?
Example

- **Free space:** 2 blocks, size 20 and 15
- **Workload 1:** allocation requests: 10 then 20
  - **Best fit**
  - **First fit**
  - **Worse fit**
  
  Request of 20: fail!

- **Workload 2:** allocation requests: 8, 12, then 13
  - **Best fit**
  - **First fit**
  - **Worse fit**
  
  Request of 13: fail!
Comparison of allocation strategies

- **Best fit**
  - Tends to leave very large holes and very small holes
  - Disadvantage: very small holes may be useless

- **First fit**:
  - Tends to leave “average” size holes
  - Advantage: faster than best fit

- **Worst fit**:
  - Simulation shows that worst fit is worst in terms of storage utilization
Buddy allocator motivation

- Allocation requests: frequently $2^n$
  - E.g., allocation physical pages in FreeBSD and Linux
  - Generic allocation strategies: overly generic

- Fast search (allocate) and merge (free)
  - Avoid iterating through entire free list

- Avoid external fragmentation for req of $2^n$; keep free pages contiguous

Real: used in FreeBSD and Linux
Buddy allocator implementation

- Allocation restrictions: $2^k$, $0 \leq k \leq N$

- Data structure
  - $N$ free lists of blocks of size $2^0, 2^1, ..., 2^N$

- Allocation of $2^k$:
  - Search free lists $(k, k+1, k+2, ...)$ for appropriate size
    - Recursively divide larger blocks until reach block of correct size
    - Insert “buddy” blocks into free lists

- Free
  - Recursively coalesce block with buddy if buddy free
Buddy allocator illustration

physically contiguous pages

256 KB

128 KB

128 KB

64 KB

64 KB

32 KB

32 KB
Buddy allocation example

freelist[3] = \{0\}
freelist[0] = \{1\}, freelist[1] = \{2\}
freelist[2] = \{4\}

freelist[0] = \{1\}, freelist[1] = \{2\}
freelist[2] = \{0\}
freelist[3] = \{0\}

Legend:
Black: allocated.
Other color: on freelist of that color.
Pros and cons of buddy allocator

- **Advantages**
  - Fast and simple compared to general dynamic memory allocation
  - Avoid external fragmentation by keeping free physical pages contiguous

- **Disadvantages**
  - Internal fragmentation
    - Allocation of block of k pages when k != 2^n
Slab allocator

- **Motivation:**
  - Frequent (de)allocation of certain kernel objects
    - E.g., file struct, inode, ...
  - Other allocators: overly general; assume variable size

- **Slab:** cache of “slots”
  - Slot size = object size
  - Free memory management = bitmap
  - Allocate: set bit and return slot
  - Free: clear bit

- **Real:** used in FreeBSD and Linux, implemented on top of buddy page allocator, for objects smaller than a page
Slab allocator illustration

kernel objects → caches → slabs

3-KB objects

7-KB objects

physically contiguous pages