Virtual memory

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

• How does the virtual memory respond when the stack or heap grows? Or is the hole between them always there? Hole is always there in virtual memory. Frames in memory allocated as needed.

• How is the TLB affected when a page is moved from Memory to Disk, and is replaced by a page brought in from Disk? TLB is cache, has mechanism for removing and adding info to it.

• When does a TLB need to be flushed completely? Context switch

• If there are no free frames and a page needs to be loaded, will process wait indefinitely, or would run using backing store? Answer coming up.

• How does OS decide that a page in memory is not longer being used? Answer coming up.
Steps in Handling a Page Fault

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

Load M
Operating System
Reference
Page Table
Physical Memory
• Three major activities
  – Service the interrupt – careful coding means just several hundred instructions needed
  – Read the page – lots of time
  – Restart the process – again just a small amount of time

• Page Fault Rate $0 \leq p \leq 1$
  – if $p = 0$ no page faults
  – if $p = 1$, every reference is a fault

• Effective Access Time (EAT)
  $EAT = (1 - p) \times \text{memory access time}$
  $+ p \left( \text{page fault overhead } + \text{swap page out} + \text{swap page in} \right)$

$\text{Page swap time} = \text{seek time} + \text{latency time}$
Demand Paging Numerical Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = \((1 - p) \times 200 + p \times 8,000,000\) nanosec.
  \[= (1 - p) \times 200 + p \times 7,999,800\] ns

- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- **If want performance degradation < 10 percent, \(p = ?\)**
  - 220 > 200 + 7,999,800 \(\times p\)
  - 20 > 7,999,800 \(\times p\)
  - \(p < .0000025\)
  - < one page fault in every 400,000 memory accesses
Issues: Allocation of physical memory to I/O and programs

- Memory used for holding **program** pages
- **I/O buffers** also consume a big chunk of memory
- **Solutions:**
  - Fixed percentage set aside for I/O buffers
  - Processes and the I/O subsystem compete
Demand paging and the limits of logical memory

• Without demand paging
  – All pages of process **must be** in physical memory
  – Logical memory **limited** to size of physical memory

• With demand paging
  – All pages of process **need not be** in physical memory
  – Size of logical address space is **no longer constrained** by physical memory

• Example
  – 40 pages of physical memory
  – 6 processes each of which is 10 pages in size
    • Each process only needs 5 pages as of now
  – Run 6 processes with 10 pages to spare

Higher degree of multiprogramming
Coping with over-allocation of memory

Example

- Physical memory = 40 pages
- 6 processes each of which is of size 10 pages
  - But are using 5 pages each as of now
- What happens if each process needs all 10 pages?
  - 60 physical frames needed

- **Option: Terminate** a user process
  - But paging should be transparent to the user

- **Option: Swap out** a process
  - Reduces the degree of multiprogramming

- **Option: Page replacement**: selected pages. Policy?

soon
Copy-on-Write (on Fork)

- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a *pool* of *zero-fill on-demand* pages
  - Pool should always have free frames for fast demand page execution
    - Don’t want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it? *(security)*
Copy-on-write

Before Process 1 Modifies Page C

After Process 1 Modifies Page C
What Happens if there is no Free Frame?

• Could be all used up by process pages or kernel, I/O buffers, etc
  – How much to allocate to each?

• Page replacement – find some page in memory, but not really in use, page it out
  – Algorithm – terminate? swap out? replace the page?
  – Performance – want an algorithm which will result in minimum number of page faults

• Same page may be brought into memory several times

Continued to Page replacement etc...
Page Replacement

• Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

• Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk

• Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
Need For Page Replacement

Logical memory for user 1:
- Frame 0: H
- Frame 1: J
- Frame 2: M

Page table for user 1:
- Frame 3: valid
- Frame 4: valid
- Frame 5: valid
- Frame 6: invalid

Logical memory for user 2:
- Frame 0: A
- Frame 1: B
- Frame 2: D
- Frame 3: E

Page table for user 2:
- Frame 6: valid
- Frame 2: invalid
- Frame 7: valid

Physical memory:
- Frame 0: monitor
- Frame 1: D
- Frame 2: H
- Frame 3: load M
- Frame 5: J
- Frame 6: A
- Frame 7: E

Load M

Colorado State University
Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT
Page Replacement

1. Swap out victim page
2. Change to invalid
3. Swap desired page in
4. Reset page table for new page

-frame
-valid-invalid bit
-page table

physica memory

Colorado State University
FAQ

• Can more than one page loaded into memory when a process starts? prefetching

• Why are disk addresses of non-resident pages not stored in the page table? Generally contains only information used on page hits.
Page Replacement Algorithms

• **Page-replacement algorithm**
  – Which frames to replace
  – Want lowest page-fault rate

• **Evaluate algorithm** by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  – String is just page numbers, not full addresses
  – Repeated access to the same page does not cause a page fault
  – Results depend on number of frames available

• In all our examples, we use 3 frames and the reference string of referenced page numbers is

  7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
Graph of Page Faults Versus The Number of Frames
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
Page replacement algorithms

- **First in first out (FIFO)**
  - Replace the oldest one

- **Optimal**
  - If you could foresee future perfectly

- **Least recently used (LRU)**
  - How will be track usage? Approximate?
FIFO page replacement algorithm:
Out with the old; in with the new

- When a page must be replaced
  - Replace the oldest one
- OS maintains list of all pages currently in memory
  - Page at head of the list: Oldest one
  - Page at the tail: Recent arrival
- During a page fault
  - Page at the head is removed
  - New page added to the tail
First-In-First-Out (FIFO) Algorithm

• Reference string:
  \[7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1\]

• 3 frames (3 pages can be in memory at a time per process)

• 15 page faults (out of 20 accesses)

• Sometimes a page is needed soon after replacement
  \[7,0,1,2,0,3 (0 \text{ out}),0,\ldots\]
Belady’s Anomaly

• Consider Page reference string
  1,2,3,4,1,2,5,1,2,3,4,5
  – 3 frames, 9 faults, 4 frames 10 faults!
  – Adding more frames can cause more page faults!

Belady was here at CSU. Guest in my CS530!

3 frames: 9 page faults
4 frames: 10 page faults
(Try yourself)
FAQ

• Does Belady’s Anomaly affect all page replacement algorithms?
• When Belady’s Anomaly is present in an algorithm, will adding an additional frame always cause more page faults?
• Paging algorithms seem to assume both temporal and spatial locality... how true is this?
“Optimal” Algorithm

- Replace page that will not be used for longest period of time

- 4th access: replace 7 because we will not use if got the longest time...
- 9 page replacements is optimal for the example

- But how do we know this?
  - Can’t read the future in reality.

- Used for *measuring* how well an algorithm performs
Least Recently Used (LRU) Algorithm

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

Number of faults

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

- 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
Buffering and applications

• 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