Operating Systems

Virtual Memory

Memory Management Outline
- Processes ✓
- Memory Management ✓
  - Basic ✓
  - Paging ✓
  - Virtual memory ←

Motivation
- Logical address space larger than physical memory
  - "Virtual Memory" on special disk
- Abstraction for programmer
- Performance ok?
  - Error handling not used
  - Maximum arrays

Demand Paging
- Less I/O needed
- Less memory needed
- Faster response
- More users
- No pages in memory initially
  - Pure demand paging

Paging Implementation

Validation Bit

Page Fault
- Page not in memory
  - interrupt OS => page fault
- OS looks in table:
  - invalid reference? => abort
  - not in memory? => bring it in
- Get empty frame (from list)
- Swap page into frame
- Reset tables (valid bit = 1)
- Restart instruction
Performance of Demand Paging

Page Fault Rate
\[ 0 \leq p < 1.0 \] (no page faults to every is fault)

Effective Access Time
\[ = (1-p) \text{ (memory access)} + p \text{ (Page Fault Overhead)} \]

Page Fault Overhead
\[ = \text{ swap page out } + \text{ swap page in } + \text{ restart} \]

Page Replacement

- Page fault \( \Rightarrow \) What if no free frames?
  - terminate user process (ugh!)
  - swap out process (reduces degree of multiprog)
  - replace other page with needed page
- Page replacement:
  - if free frame, use it
  - use algorithm to select \textit{victim} frame
  - write page to disk, changing tables
  - read in new page
  - restart process

Page Replacement Algorithms

- Every system has its own
- Want lowest \textit{page fault rate}
- Evaluate by running it on a particular string of memory references (\textit{reference string}) and computing number of page faults
- Example: 1,2,3,4,1,2,5,1,2,3,4,5

Performance Example

- memory access time = 100 nanoseconds
- swap fault overhead = 25 msec
- page fault rate = 1/1000
- \textit{EAT} = \( \text{(1-p)} \times 100 + p \times (25 \text{ msec}) \)
  \[ = (1-p) \times 100 + p \times 25,000,000 \]
  \[ = 100 + 24,999,900 \times p \]
  \[ = 100 + 24,999,900 \times \frac{1}{1000} = 25 \text{ microseconds}! \]
- Want less than 10\% degradation
  \[ 110 > 100 + 24,999,900 \times p \]
  \[ 10 > 24,999,900 \times p \]
  \[ p < .0000004 \text{ or } 1 \text{ fault in } 2,500,000 \text{ accesses} \]

Page Replacement

- \textit{Dirty} Bit - avoid page out

First-In-First-Out (FIFO)

\[ 1,2,3,4,1,2,5,1,2,3,4,5 \]

9 Page Faults

Belady’s Anomaly
**Optimal**

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

![Optimal Diagram](image)

<table>
<thead>
<tr>
<th>4 Frames / Process</th>
<th>1,2,3,4,1,2,5,1,2,3,4,5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Page Faults</td>
<td></td>
</tr>
</tbody>
</table>

**Least Recently Used**

- Replace the page that has not been used for the longest period of time

![Least Recently Used Diagram](image)

<table>
<thead>
<tr>
<th>5 Page Faults</th>
<th>1,2,3,4,1,2,5,1,2,3,4,5</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Page Faults</td>
<td></td>
</tr>
</tbody>
</table>

**LRU Implementation**

- Counter implementation
  - every page has a counter; every time page is referenced, copy clock to counter
  - when a page needs to be changed, compare the counters to determine which to change
- Stack implementation
  - keep a stack of page numbers
  - page referenced: move to top
  - no search needed for replacement

**LRU Approximations**

- LRU good, but hardware support expensive
- Some hardware support by reference bit
  - with each page, initially = 0
  - when page is referenced, set = 1
  - replace the one which is 0 (no order)
  - enhance by having 8 bits and shifting
  - approximate LRU

**Second-Chance**

- FIFO replacement, but …
  - Get first in FIFO
  - Look at reference bit
    - bit = 0 then replace
    - bit = 1 then set bit = 0, get next in FIFO
  - If page referenced enough, never replaced
  - Implement with circular queue

![Second-Chance Diagram](image)

(a) (b)

If all 1, degenerates to FIFO
Enhanced Second-Chance
- 2-bits, reference bit and modify bit
  - (0,0) neither recently used nor modified
    - best page to replace
  - (0,1) not recently used but modified
    - needs write-out
  - (1,0) recently used but clean
    - probably used again soon
  - (1,1) recently used and modified
    - used soon, needs write-out
- Circular queue in each class -- (Macintosh)

Counting Algorithms
- Keep a counter of number of references
  - LFU - replace page with smallest count
    - if does all in beginning, won’t be replaced
    - decay values by shift
  - MFU - smallest count just brought in and will probably be used
- Not too common (expensive) and not too good

Page Buffering
- Pool of frames
  - start new process immediately, before writing old
    - write out when system idle
  - list of modified pages
    - write out when system idle
  - pool of free frames, remember content
    - page fault $\Rightarrow$ check pool

Allocation of Frames
- How many fixed frames per process?
- Two allocation schemes:
  - fixed allocation
  - priority allocation

Fixed Allocation
- Equal allocation
  - ex: 93 frames, 5 procs = 18 per proc (3 in pool)
- Proportional Allocation
  - number of frames proportional to size
  - ex: 64 frames, s1 = 10, s2 = 127
    - $f1 = 10 / 137 \times 64 = 5$
    - $f2 = 127 / 137 \times 64 = 59$
- Treat processes equal

Priority Allocation
- Use a proportional scheme based on priority
- If process generates a page fault
  - select replacement a process with lower priority
- “Global” versus “Local” replacement
  - local consistent (not influenced by others)
  - global more efficient (used more often)
Thrashing

+ If a process does not have “enough” pages, the page-fault rate is very high
  – low CPU utilization
  – OS thinks it needs increased multiprogramming
  – adds another process to system
+ Thrashing is when a process is busy swapping pages in and out

Cause of Thrashing

+ Why does paging work?
  – Locality model
    • process migrates from one locality to another
    • localities may overlap
+ Why does thrashing occur?
  – sum of localities > total memory size
+ How do we fix thrashing?
  – Working Set Model
  – Page Fault Frequency

Working Set Example

+ $T = 5$
+ 1 2 3 2 3 1 2 4 3 4 7 4 3 4 1 1 2 2 2 1
  \[ W = \{1,2,3\} \quad W = \{3,4,7\} \quad W = \{1,2\} \]
  – if $T$ too small, will not encompass locality
  – if $T$ too large, will encompass several localities
  – if $T \to \infty$, will encompass entire program
+ if $D > m$ => thrashing, so suspend a process
+ Modify LRU appx to include Working Set

Working-Set Model

+ Working set window $W =$ a fixed number of page references
  – total number of pages references in time $T$
+ $D =$ sum of size of $W$’s

Page Fault Frequency

+ Establish “acceptable” page-fault rate
  – If rate too low, process loses frame
  – If rate too high, process gains frame
**Prepaging**

- Pure demand paging has many page faults initially
  - use working set
  - does cost of prepaging unused frames outweigh cost of page-faulting?

**Page Size**

- Old - Page size fixed, New -choose page size
- How do we pick the right page size? Tradeoffs:
  - Fragmentation
  - Table size
  - Minimize I/O
    - transfer small (.1ms), latency + seek time large (10ms)
  - Locality
    - small finer resolution, but more faults
      - ex: 200K process (1/2 used), 1 fault / 200K, 100K faults / 1 byte
- Historical trend towards larger page sizes
  - CPU, mem faster proportionally than disks

**Program Structure**

- consider:
  ```c
  int A[1024][1024];
  for (j=0; j<1024; j++)
    for (i=0; i<1024; i++)
      A[i][j] = 0;
  ```
- suppose:
  - process has 1 frame
  - 1 row per page
  - => 1024x1024 page faults!

**Program Structure**

- Old: `int A[1024][1024];`
- New: `for (j=0; j<1024; j++)`
- `A[i][j] = 0;`
- 1024 page faults
- stack vs. hash table
- Compiler
  - separate code from data
  - keep routines that call each other together
- LISP (pointers) vs. Pascal (no-pointers)

**Priority Processes**

- Consider
  - low priority process faults,
    - bring page in
  - low priority process in ready queue for awhile, waiting while high priority process runs
  - high priority process faults
    - low priority page clean, not used in a while
      => perfect!
  - Lock-bit (like for I/O) until used once

**Real-Time Processes**

- Real-time
  - bounds on delay
  - hard-real time: systems crash, lives lost
    - air-traffic control, factor automation
  - soft-real time: application sucks
    - audio, video
- Paging adds unexpected delays
  - don’t do it
  - lock bits for real-time processes
### Virtual Memory and WinNT

- **Page Replacement Algorithm**
  - FIFO
  - Missing page, plus adjacent pages
- **Working set**
  - default is 30
  - take *victim* frame periodically
  - if no fault, reduce set size by 1
- **Reserve pool**
  - hard page faults
  - soft page faults

### Virtual Memory and Linux

- **Regions of virtual memory**
  - paging disk (normal)
  - file (text segment, memory mapped file)
- **New Virtual Memory**
  - `exec()` creates new page table
  - `fork()` copies page table
    - reference to common pages
    - if written, then copied
- **Page Replacement Algorithm**
  - second chance (with more bits)

### Application Performance Studies and Demand Paging in Windows NT

Mikhail Mikhailov  
Ganga Kannan  
Mark Claypool  
David Finkel  
*WPI*

Saqib Syed  
Divya Prakash  
Sujit Kumar  
*BMC Software, Inc.*

### Capacity Planning Then and Now

- **Capacity Planning in the good old days**
  - used to be just mainframes
  - simple CPU-load based queuing theory
  - Unix
- **Capacity Planning today**
  - distributed systems
  - networks of workstations
  - *Windows NT*
  - *MS Exchange, Lotus Notes*

### Experiment Design

- **System**
  - Pentium 133 MHz
  - *NT Server 4.0*
  - 64 MB RAM
  - IDE NTFS
- **clearmem**
  - **Experiments**
    - Page Faults
    - Caching
  - **Analysis**
    - perfmon
Page Fault Method

- "Work hard"
- Run lots of applications, open and close
- All local access, not over network

Soft or Hard Page Faults?

Caching and Prefetching

- Start process
  - wait for "Enter"
- Start perfmon
- Hit "Enter"
- Read 1 4-K page
- Exit
- Repeat

Page Metrics with Caching On