



Operating Systems

Virtual Memory
(Ch 10)

Memory Management Outline

- Processes (done)
- Memory Management
 - Basic (done)
 - Paging (done)
 - Virtual memory ←

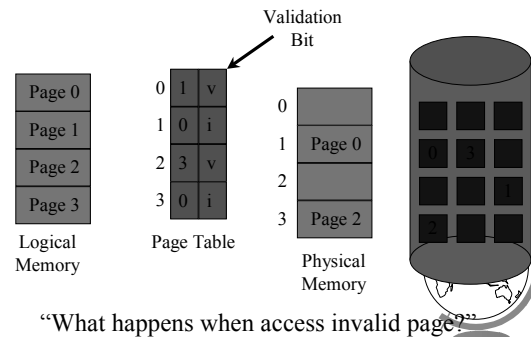


Motivation

- Logical address space larger than physical memory
 - 2^{32} about 4 GB in size
 - “Virtual Memory”
 - on special disk
- Abstraction for programmer
- Performance ok? Examples:
 - Unused libraries
 - Error handling not used
 - Maximum arrays



Paging Implementation



Accessing Invalid Pages

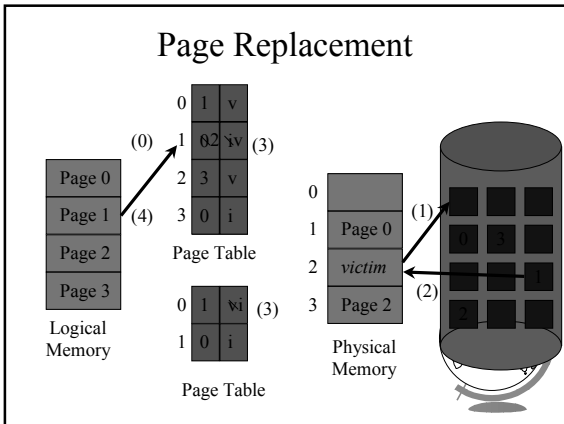
- Page not in memory
 - interrupt OS => *page fault*
- OS looks in table:
 - invalid reference? => *abort*
 - not in memory? => *bring it in*
- Get empty frame (from list)
- Write page from disk into frame
- Reset tables (set valid bit = 1)
- Restart instruction



No Free Frames


- Page fault => What if no free frames?
 - terminate process (out of memory)
 - swap out process (reduces degree of multiprog)
 - replace another page with needed page
 - *Page replacement*
- Page fault with page replacement:
 - if free frame, use it
 - else use algorithm to select *victim* frame
 - write page to disk
 - read in new page
 - change page tables
 - restart process






Performance of Demand Paging

- Page Fault Rate (p)
 $0 \leq p < 1.0$ (no page faults to every ref is a fault)
- Page Fault Overhead
 = write page in + update + restart
 – Dominated by time to write page in
- Effective Access Time
 = $(1-p)$ (memory access) + p (page fault overhead)




Performance Example

- Memory access time = 100 nanoseconds
- Page fault overhead = 25 msec
- Page fault rate = 1/1000
- EAT = $(1-p) * 100 + p * (25 \text{ msec})$
 $= (1-p) * 100 + p * 25,000,000$
 $= 100 + 24,999,900 * p$
 $= 100 + 24,999,900 * 1/1000 = 25 \text{ microseconds!}$
- Want less than 10% degradation
 $110 > 100 + 24,999,900 * p$
 $10 > 24,999,900 * p$
 $p < .0000004$ or 1 fault in 2,500,000 accesses!
- Need good *page replacement* algorithms



Page Replacement Algorithms

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




First-In-First-Out (FIFO)

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

3 Frames / Process

1
2
3



First-In-First-Out (FIFO)


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

3 Frames / Process

1	4	5
2	1	3
3	2	4

9 Page Faults

How could we reduce the number of page faults?



Optimal

vs.

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

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

4 Frames / Process

1
2
3
4

Optimal

vs.

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

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

4 Frames / Process

1	4
2	
3	
4	5

6 Page Faults

How do we know this?
Use as *benchmark*

Least Recently Used

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

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

1
2
3
4

Least Recently Used

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

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

1	5
2	
3	5 4
4	3

8 Page Faults

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
- (Can we do this in software?)

LRU Approximations

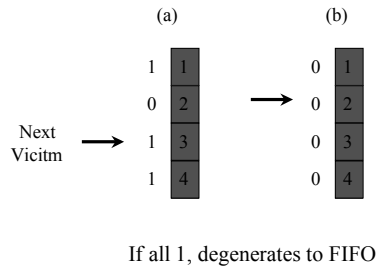
- LRU good, but hardware support expensive
- Some hardware support by *reference bit*
 - with each page, initially = 0
 - when page is referenced, set = 1
 - replace the one which is 0 (no order)
- Enhance by having 8 bits and shifting
 - approximate LRU*

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



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 (“dirty” page)
- (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)



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 => check pool

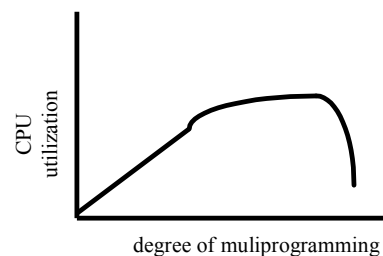


Thrashing

- 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



Thrashing



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 Model

- Working set window W = a fixed number of page references
 - total number of pages references in time T
- $Total$ = sum of size of W 's
- m = number of frames

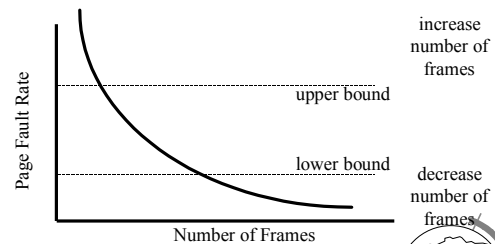


Working Set Example

- $T = 5$
- 1 2 3 2 3 1 2 4 3 4 7 4 3 3 4 1 1 2 2 2 1
 - $W = \{1, 2, 3\}$ $W = \{3, 4, 7\}$ $W = \{1, 2\}$
 - if T too small, will not encompass locality
 - if T too large, will encompass several localities
 - if $T \Rightarrow$ infinity, will encompass entire program
- if $Total > m \Rightarrow$ thrashing, so suspend a process
- Modify LRU appx to include Working Set



Page Fault Frequency



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



Outline

- Demand Paging Intro (done)
- Page Replacement Algorithms (done)
- Thrashing (done)
- Misc Paging
- WinNT/2000
- Linux
- “Application Performance Studies”



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 fault/1 byte
- Historical trend towards larger page sizes
 - CPU, mem faster proportionally than disks



Program Structure

- consider:

```
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

- ```
int A[1024][1024];
for (i=0; i<1024; i++)
 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/2000

- 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 WinNT/2000

- Shared pages
  - level of indirection for easier updates
  - same virtual entry
- Page File
  - stores only modified logical pages
  - code and memory mapped files on disk already



## Virtual Memory and Linux

- Regions of virtual memory
  - paging disk (normal)
  - file (text segment, memory mapped file)
- Re-Examine fork() and exec()
  - exec() creates new page table
  - fork() copies page table
    - + reference to common pages
    - + if written, then copied



## Virtual Memory and Linux

- Page Replacement Algorithm
  - look in reserve pool for free frames
  - reserves for block devices (disk cache)
  - reserves for shared memory
  - user-space blocks
  - enhanced second chance (with more bits)
    - + “dirty” pages not taken first



## 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

Does NT have more hard page faults or soft page faults?

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

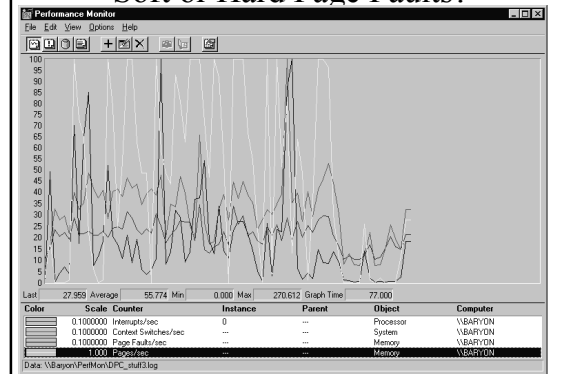


## Page Fault Method

- “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

