ICS 143 - Principles of Operating Systems

Lecture Set 6 Part 2 - Virtual Memory

Prof. Nalini Venkatasubramanian
nalini@ics.uci.edu

Virtual Memory

- Background
- Demand paging
  - Performance of demand paging
- Page Replacement
  - Page Replacement Algorithms
- Allocation of Frames
- Thrashing
- Demand Segmentation

Need for Virtual Memory

- Virtual Memory
  - Separation of user logical memory from physical memory.
  - Only *PART* of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Need to allow pages to be swapped in and out.
- Virtual Memory can be implemented via
  - ❑ Paging
  - ❑ Segmentation

Paging/Segmentation Policies

- Fetch Strategies
  - When should a page or segment be brought into primary memory from secondary (disk) storage?
    - Demand Fetch
    - Anticipatory Fetch
- Placement Strategies
  - When a page or segment is brought into memory, where is it to be put?
    - Paging - trivial
    - Segmentation - significant problem
- Replacement Strategies
  - Which page/segment should be replaced if there is not enough room for a required page/segment?

Demand Paging

- Bring a page into memory only when it is needed.
    - Less I/O needed
    - Less Memory needed
    - Faster response
    - More users
- The first reference to a page will trap to OS with a page fault.
- OS looks at another table to decide
    - Invalid reference - abort
    - Just not in memory.

Valid-Invalid Bit

- With each page table entry a valid-invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not in memory).
- Initially, valid-invalid bit is set to 0 on all entries.
  - During address translation, if valid-invalid bit in page table entry is 0 --- ***page fault*** occurs.
  - Example of a page-table snapshot

Frame #	Valid-invalid bit
	1
	1
	1
	1
	0
	0
	0

Page Table

Handling a Page Fault

- Page is needed - reference to page
  - Step 1: Page fault occurs - trap to OS (process suspends).
  - Step 2: Check if the virtual memory address is valid. Kill job if invalid reference.   If valid reference, and page not in memory, continue.
  - Step 3: Bring into memory - Find a free page frame, map address to disk block and fetch disk block into page frame. When disk read has completed, add virtual memory mapping to indicate that page is in memory.
  - Step 4: Restart instruction interrupted by illegal address trap. The process will continue as if page had always been in memory.

What happens if there is no free frame?

- Page replacement - find some page in memory that is not really in use and swap it.
    - Need page replacement algorithm
    - Performance Issue - need an algorithm which will result in minimum number of page faults.
  - Same page may be brought into memory many times.

Performance of Demand Paging

- Page Fault Ratio - $0 \le p \le 1.0$
  - If $p = 0$, no page faults
  - If $p = 1$, every reference is a page fault
- Effective Access Time

```
EAT = (1-p) * memory-access +
            p   * (page fault overhead +
                  swap page out +
                  swap page in   +
                  restart overhead)
```

Demand Paging Example

- Memory Access time = 1 microsecond
- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out.
- Swap Page Time = 10 msec = 10,000 microsec

  EAT = (1-p) *1 + p (15000) $\approx$1 + 15000p microsec
- EAT is directly proportional to the page fault rate.

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
  - large virtual memory can be provided on a smaller physical memory.

Page Replacement Algorithms

- 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.
- Assume reference string in examples to follow is

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

Page Replacement Strategies

- The Principle of Optimality
  - Replace the page that will not be used again the farthest time into the future.
- Random Page Replacement
  - Choose a page randomly
- FIFO - First in First Out
  - Replace the page that has been in memory the longest.
- LRU - Least Recently Used
  - Replace the page that has not been used for the longest time.
- LFU - Least Frequently Used
  - Replace the page that is used least often.
- NUR - Not Used Recently
  - An approximation to LRU
- Working Set
  - Keep in memory those pages that the process is actively using

First-In-First-Out (FIFO) Algorithm

Reference String: 1,2,3,4,1,2,5,1,2,3,4,5

- Assume x frames (x pages can be in memory at a time per process)

3 frames

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

9 Page faults

4 frames

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

10 Page faults

FIFO Replacement - *Belady's Anomaly* -- more frames does not mean less page faults

Optimal Algorithm

- Replace page that will not be used for longest period of time.
  - How do you know this???
  - Generally used to measure how well an algorithm performs.

4 frames

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

6 Page faults

Least Recently Used (LRU) Algorithm

- Use recent past as an approximation of near future.
- Choose the page that has not been used for the longest period of time.
- May require hardware assistance to implement.
- Reference String: 1,2,3,4,1,2,5,1,2,3,4,5

4 frames

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

8 Page faults

Implementation of LRU algorithm

- 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 changes, look at the counters to determine which page to change (page with smallest time value).
- Stack Implementation
    - Keeps a stack of page numbers in a doubly linked form
    - Page referenced
        - move it to the top
        - requires 6 pointers to be changed
    - No search required for replacement

LRU Approximation Algorithms

- Reference Bit
  - With each page, associate a bit, initially = 0.
  - When page is referenced, bit is set to 1.
  - Replace the one which is 0 (if one exists). Do not know order however.
- Additional Reference Bits Algorithm
  - Record reference bits at regular intervals.
  - Keep 8 bits (say) for each page in a table in memory.
  - Periodically, shift reference bit into high-order bit, I.e. shift other bits to the right, dropping the lowest bit.
  - During page replacement, interpret 8bits as unsigned integer.
  - The page with the lowest number is the LRU page.

LRU Approximation Algorithms

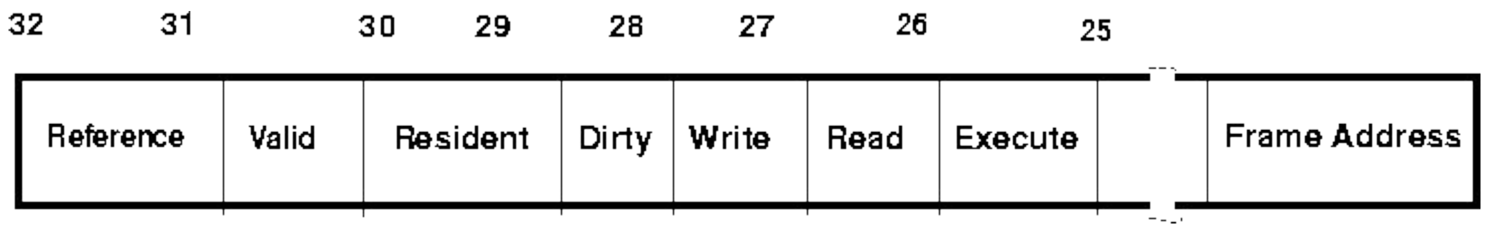
- Second Chance
  - FIFO (clock) replacement algorithm
  - Need a reference bit.
  - When a page is selected, inspect the reference bit.
  - If the reference bit = 0, replace the page.
  - If page to be replaced (in clock order) has reference bit = 1, then
    - set reference bit to 0
    - leave page in memory
    - replace next page (in clock order) subject to same rules.

Counting Algorithms

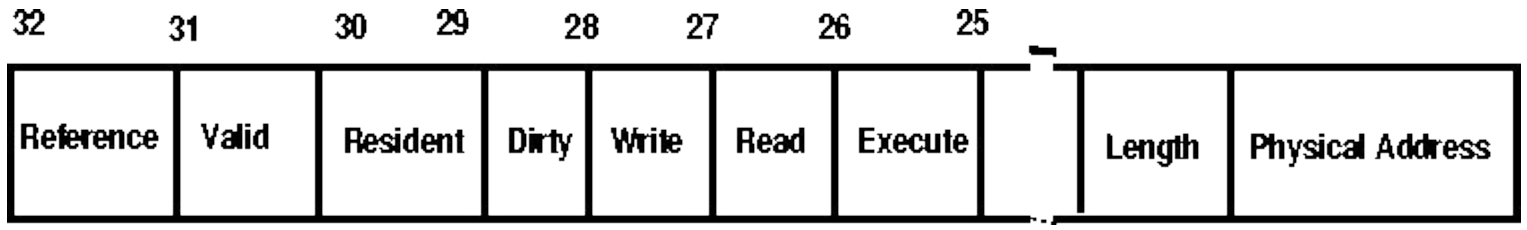
- Keep a counter of the number of references that have been made to each page.
  - LFU (least frequently used) algorithm
    - replaces page with smallest count.
    - Rationale : frequently used page should have a large reference count.
      - Variation - shift bits right, exponentially decaying count.
  - MFU (most frequently used) algorithm
    - replaces page with highest count.
    - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Protection Bits

Page Protection



Segmentation Protection



Reference	- ***Page has been accessed***
Valid	- ***Page exists***
Resident	- ***Page is cached in primary memory***
Dirty	- ***Page has been changed since page in***

Demand Paging + Multiprogramming

- Need to rethink Allocation and replacement

- Allocation of Frames
  - Single user case is simple
    - User is allocated any free frame
  - Problem: Demand paging + multiprogramming
    - Each process needs minimum number of pages based on instruction set architecture.
    - Example IBM 370: 6 pages to handle MVC (storage to storage move) instruction
      - Instruction is 6 bytes, might span 2 pages.
      - 2 pages to handle ***from***.
      - 2 pages to handle ***to***.
    - Two major allocation schemes:
      - Fixed allocation -- Equal, Proportional
      - Priority allocation

Fixed Allocation

- Equal Allocation
  - E.g. If 100 frames and 5 processes, give each 20 pages.
- Proportional Allocation
  - Allocate according to the size of process
    - S_j = size of process P_j
    - $S = \sum S_j$
    - m = total number of frames
    - a_j = allocation for $P_j = S_j/S * m$
    - If $m = 64$, $S_1 = 10$, $S_2 = 127$ then

      $a_1 = 10/137 * 64 \approx 5$

      $a_2 = 127/137 * 64 \approx 59$

Priority Allocation

- May want to give high priority process more memory than low priority process.
- Use a proportional allocation scheme using priorities instead of size
- If process Pi generates a page fault
  - select for replacement one of its frames
  - select for replacement a frame form a process with lower priority number.

Global vs. Local Replacement

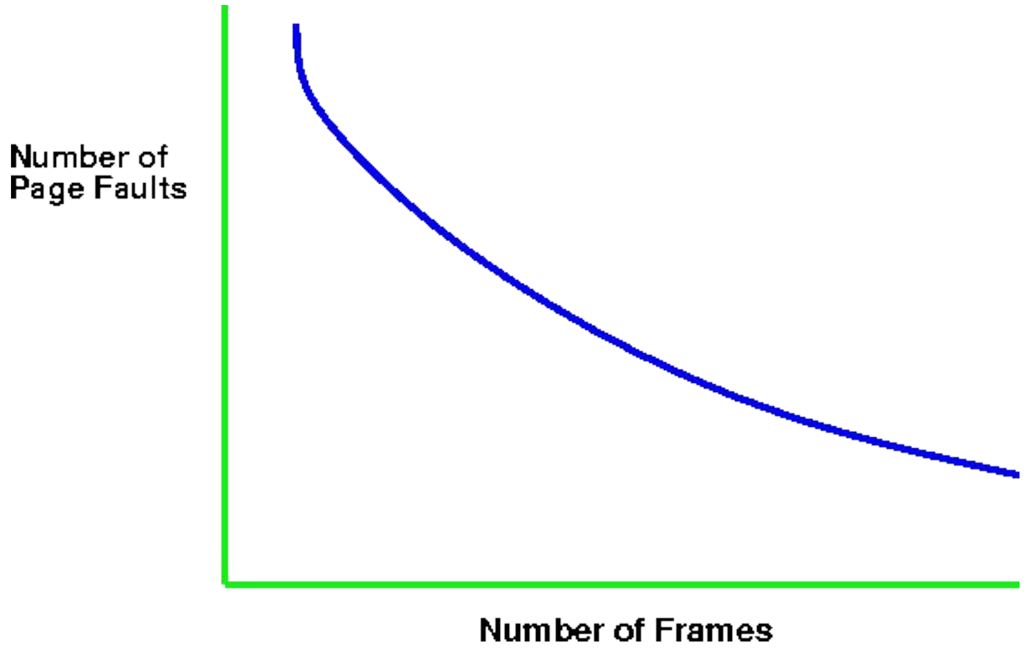
- Global Replacement
  - Process selects a replacement frame from the set of all frames.
  - One process can take a frame from another.
  - Process may not be able to control its page fault rate.
- Local Replacement
  - Each process selects from only its own set of allocated frames.
  - Process slowed down even if other less used pages of memory are available.
- Global replacement has better throughput
  - Hence more commonly used.

Thrashing

- If a process does not have enough pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - OS thinks that it needs to increase the degree of multiprogramming
  - Another process is added to the system.
  - System throughput plunges...
- *Thrashing*
  - A process is busy swapping pages in and out.
  - In other words, a process is spending more time paging than executing.

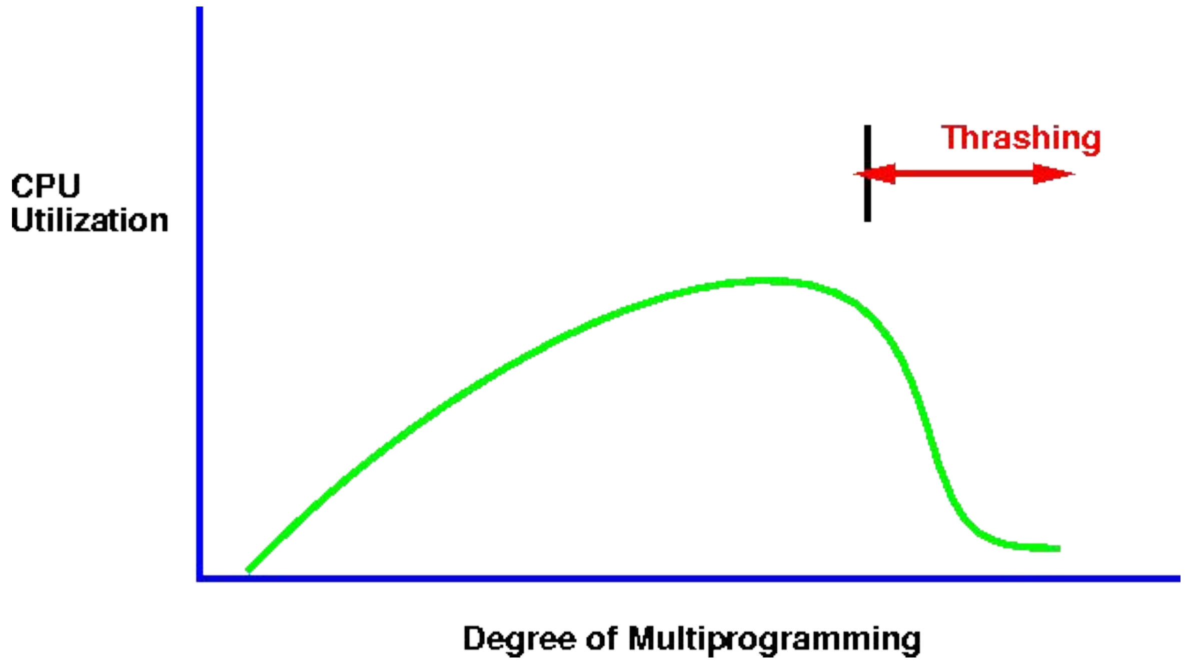
Thrashing (cont.)

- Why does paging work?
  - Locality Model - computations have locality!
  - Locality - set of pages that are actively used together.
  - Process migrates from one locality to another.
  - Localities may overlap.



Thrashing

- Why does thrashing occur?
  - ∑ (size of locality) **>** total memory size



Working Set Model

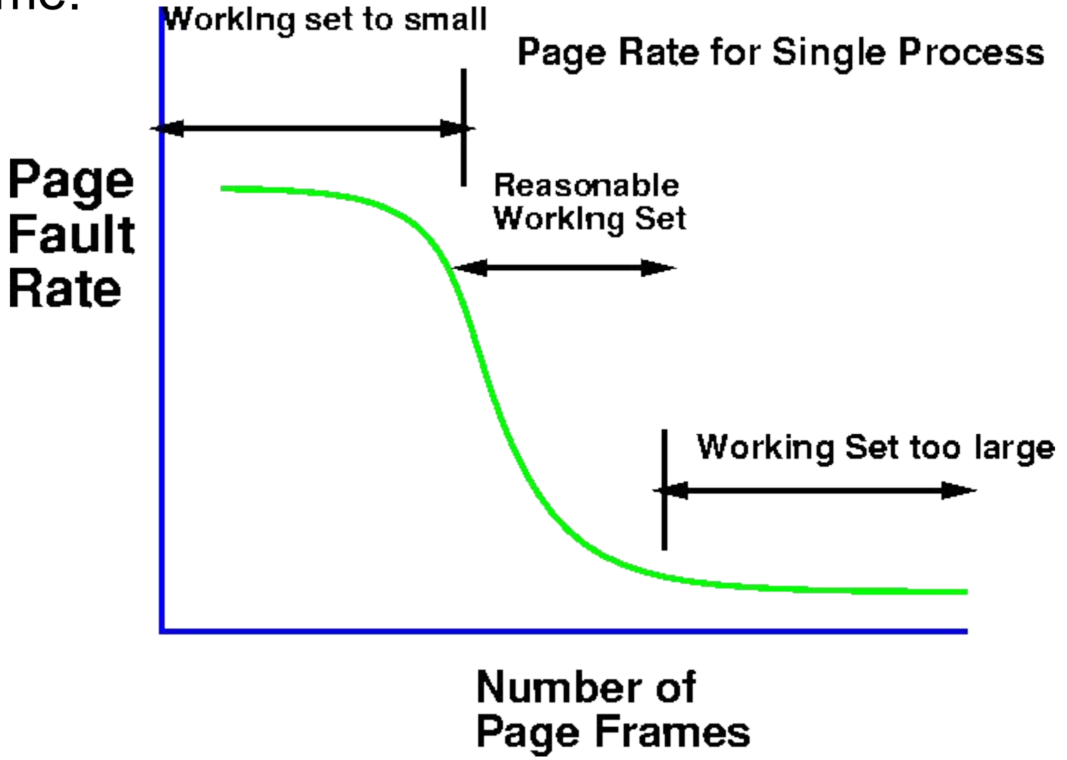
- Δ ≡ working-set window
    - a fixed number of page references, e.g. 10,000 instructions
- WSSj (working set size of process Pj) - total number of pages referenced in the most recent Δ (varies in time)
    - If Δ too small, will not encompass entire locality.
    - If Δ too large, will encompass several localities.
    - If Δ = ∞, will encompass entire program.
- D = ∑ WSSj ≡ total demand frames
    - If D > m (number of available frames) ⇒thrashing
- Policy: If D > m, then suspend one of the processes.

Keeping Track of the Working Set

- Approximate with
  - interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units.
  - Whenever a timer interrupts, copy and set the values of all reference bits to 0.
  - Keep in memory 2 bits for each page (indicated if page was used within last 10,000 to 15,000 references).
  - If one of the bits in memory = 1 ⇒ page in working set.
  - Not completely accurate - cannot tell where reference occurred.
  - Improvement - 10 bits and interrupt every 1000 time units.

Page fault Frequency Scheme

- Control thrashing by establishing *acceptable* page-fault rate.
  - If page fault rate too low, process loses frame.
  - If page fault rate too high, process needs and gains a frame.



EXTRA SLIDES (Not for finals)

Demand Paging Issues

- Prepaging
  - Tries to prevent high level of initial paging.
    - E.g. If a process is suspended, keep list of pages in working set and bring entire working set back before restarting process.
    - Tradeoff - page fault vs. prepaging - depends on how many pages brought back are reused.
- Page Size Selection
  - fragmentation
  - table size
  - I/O overhead
  - locality

Demand Paging Issues

- Program Structure
  - Array A[1024,1024] of integer
  - Assume each row is stored on one page
  - Assume only one frame in memory
  - Program 1

```
for j := 1 to 1024 do
for i := 1 to 1024 do
      A[i,j] := 0;
```

1024 * 1024 page faults

  - Program 2

```
for i := 1 to 1024 do
for j:= 1 to 1024 do
      A[i,j] := 0;
```

1024 page faults

Demand Paging Issues

- I/O Interlock and addressing
  - Say I/O is done to/from virtual memory. I/O is implemented by I/O controller.
    - Process A issues I/O request
    - CPU is given to other processes
    - Page faults occur - process A’s pages are paged out.
    - I/O now tries to occur - but frame is being used for another process.
  - Solution 1: never execute I/O to memory - I/O takes place into system memory. Copying Overhead!!
  - Solution 2: Lock pages in memory - cannot be selected for replacement.

Demand Segmentation

- Used when there is insufficient hardware to implement demand paging.
- OS/2 allocates memory in segments, which it keeps track of through segment descriptors.
  - Segment descriptor contains valid bit to indicate whether the segment is currently in memory.
    - If segment is in main memory, access continues.
    - If not in memory, segment fault.