Lecture Overview

- Virtual memory
  - Demand paging
  - Page faults
  - Page replacement
  - Frame allocation
  - Thrashing

Virtual Memory

- Up until now we assumed that an entire program/process needed to be in memory to execute
  - This is because we were still directly associating logical memory with physical memory
- Virtual memory further separates 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

Demand Paging

- Demand paging is a common approach for implementing virtual memory
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - Invalid reference ⇒ abort
  - Not-in-memory ⇒ bring to memory

Demand Paging Page Table

- With each page table entry a *valid* bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory or not valid)
- Initially valid bit is set to 0 on all entries
- Example of a page table snapshot

Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

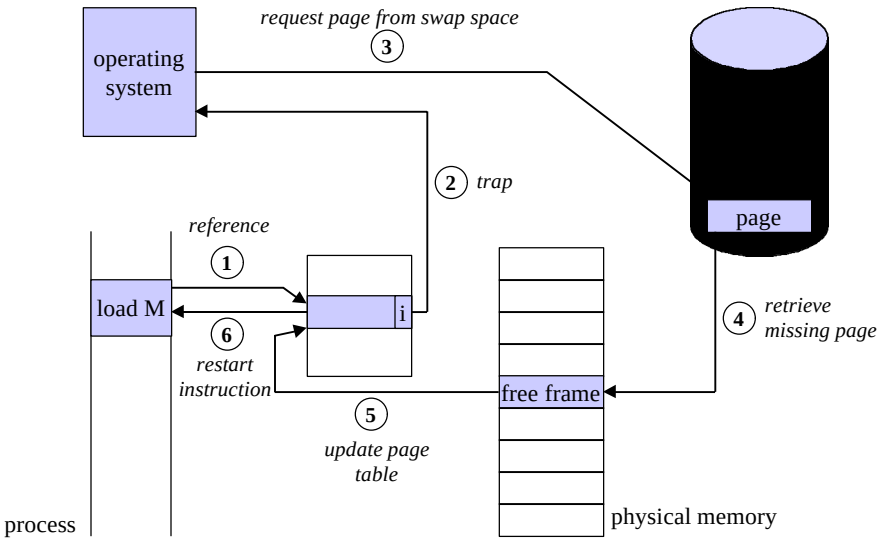
page table

- If valid bit in page table entry is 0 ⇒ *page fault*

Page Fault

- The first reference to a page will trap to the OS as a page fault
- OS looks at process page table to decide
  - Invalid reference ⇒ abort
  - Just not in memory
- Get empty frame
- Swap page into frame
- Reset tables, validation bit = 1
- Restart instruction
  - Not necessarily easy to restart instruction, consider block moves and auto increment/decrement location instructions

Page Fault



Demand Paging Performance

- Page fault rate $0 \le p \le 1.0$
  - If $p = 0$, that means no page faults
  - If $p = 1$, that means every reference is a page fault
- Effective Access Time (EAT)

  EAT = (1 – p) (memory access)

  + p (page fault overhead)

  Page fault overhead =

  + potentially a swap out

  + swap in

  + restart overhead

Demand Paging Performance

- Effective Access Time (EAT)
  - Assume total software overhead of a page faults is about one millisecond
  - Assume hard disk overhead is about 24 milliseconds (8 ms latency, 15 ms seek, and 1 ms transfer)
  - Assume memory access is 100 nanoseconds

  EAT = (1 - p) (100 ns) + p (25 ms)

  = (1 - p) (100 ns) + p (25,000,000 ns)

  = 100 + 24,999,900p ns

  If we page fault once in 1000 references, then EAT is approximately 25100 nanoseconds!!

Page Replacement

- It is possible for the OS to run out of page frames (i.e., physical memory), the page fault service routine must perform 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

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
- In all our examples, the reference string is

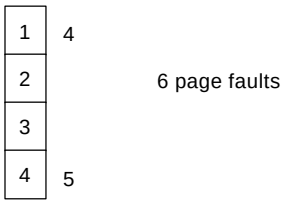
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

Page Replacement Algorithms

Optimal algorithm

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

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

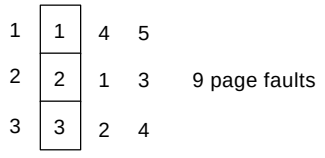


- How do you know this?
- Used for measuring how well your algorithm performs

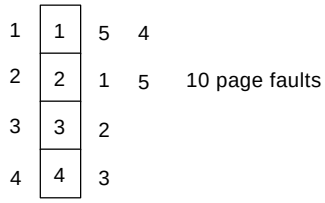
Page Replacement Algorithms

First-in, first-out (FIFO) algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



- 4 frames

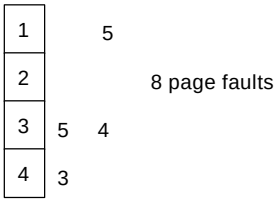


- FIFO Replacement – Belady's Anomaly
  - More frames does not necessarily mean less page faults

Page Replacement Algorithms

Least recently used (LRU) algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- 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 determine which are to change

Page Replacement Algorithms

Least recently used (LRU) algorithm

- Stack implementation – keep a stack of page numbers in a double link form:
  - When a page referenced
    - Removed from stack and put on the top
    - Requires 6 pointers to be changed
  - No search for replacement (use tail pointer)

- LRU is difficult to implement efficiently and usually requires some sort of hardware support

Page Replacement Algorithms

Approximations of LRU algorithm

- Reference bit
  - With each page associate a bit, initially 0
  - When page is referenced bit set to 1
  - Replace a page with a 0 reference bit (if one exists); we do not know the order, however
- Second chance
  - Need reference bit
  - Circular queue
  - If page to be replaced (in clock order) has reference bit of 1, then
    - Set reference bit 0
    - Leave page in memory
    - Replace next page (in clock order), subject to same rules

Page Replacement Algorithms

Counting algorithms

- Keep a counter of the number of references that have been made to each page
  - 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
- These types of algorithms tend to be expensive and do not approximate the optimal algorithm very well

Frame Allocation

- Each process needs minimum number of pages
  - One aspect is because of performance consideration
  - Another aspect is due to the computing hardware itself
- Example: IBM 370 – 6 pages to handle `MVC` instruction
  - Instruction is 6 bytes, might span 2 pages
  - 2 pages to handle **from** memory block
  - 2 pages to handle **to** memory block
- Two major allocation schemes
  - Fixed allocation
  - Priority allocation

Frame Allocation Algorithms

- Equal allocation
  - If there are m frames and n processes, then each process gets m/n frames
  - Does not take into account process memory needs
- Proportional allocation
  - Add up the total size of all processes, S
  - For a given process, p_i , use its size, s_i , to determine is proportional size, s_i/S
  - Multiply the number of frames, m, by the proportional size of a specific process to get its number of frames

Frame Allocation Algorithms

- Priority allocation
  - Use a proportional allocation scheme using priorities rather than size
  - If process P_i generates a page fault
    - Select for replacement one of its frames
    - Select for replacement a frame from a process with lower priority number

Frame Allocation Algorithms

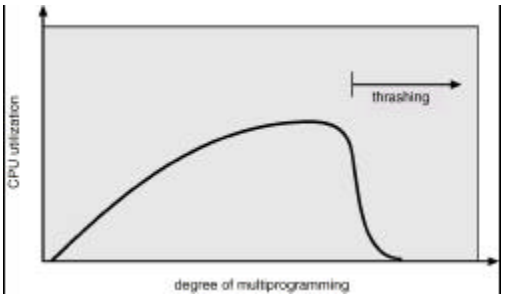
- Local versus global allocation
  - Local replacement
    - Each process selects from only its own set of allocated frames
  - Global replacement
    - Process selects a replacement frame from the set of all frames; one process can take a frame from another

Global replacement is more common since it results in greater system throughput

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high; this leads to
  - Low CPU utilization
  - Operating system thinks that it needs to increase the degree of multiprogramming
  - Another process added to the system
- Thrashing is when a process is swapping pages in and out more than it is actually computing

Thrashing



- Why does paging work?
  - Locality model, i.e., a process migrates from one locality to another
- Why does thrashing occur?
  - Size of locality > maximum number of available page frames

Working-Set Model

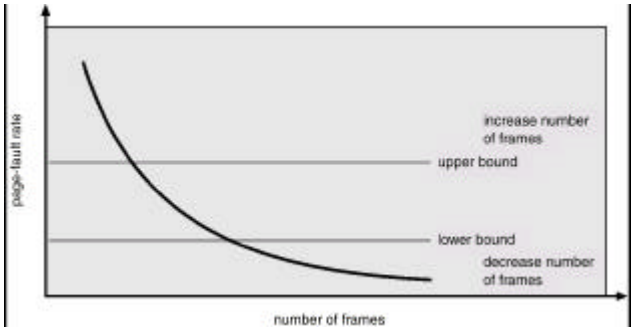
- Used to avoid thrashing
- $\Delta \equiv$ working-set window (i.e., a fixed number of page references) *Example:* 10,000 instruction
- WSS_i (working set size of process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - If Δ too small will not encompass entire locality
  - If Δ too large will encompass several localities
  - If $\Delta = \infty$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
- If $D >$ *total memory frames* then thrashing will occur
- Policy if $D > m$, then suspend one of the processes

Working-Set Model

- Difficult to determine exact working set
- Approximate with interval timer + a reference bit
- *Example:* $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy reference bit into memory bits and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 then page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units, but...

Page-Fault Frequency

Another approach to avoid thrashing



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

Other Considerations

- Pre-paging
  - Bring in pages in advance rather than waiting for faults
- Page size selection
  - Small page reduces internal fragmentation and increases resolution of locality, but increases table size and I/O time
  - Large page reduces table size and I/O time, but increases internal fragmentation and decreases resolution of locality
- Inverted page table
  - Now that we allow processes to be partially in memory, the inverted page table approach does not contain necessary information
  - It must augment with a page table per process to keep track of which pages are in memory for each process

Other Considerations

- Affect of program structure
  - Array A[1024, 1024] of integers
  - Each row is stored in one page, i.e., one frame
  - Program 1: **for** *j* := 1 to 1024
    for *i* := 1 to 1024 **do**
    A[*i*, *j*] := 0;

    1024 x 1024 = 1 million page faults
  - Program 2: **for** *i* := 1 to 1024 **do**
    for *j* := 1 to 1024 **do**
    A[*i*, *j*] := 0;

    1024 page faults
- I/O interlock and addressing