Topic 8 (Textbook - Chapter 9) Virtual Memory

Ahmad AlRjoub



http://fac.ksu.edu.sa/ahmadrj

Operating System Concepts – 9th Edition

Silberschatz, Galvin and Gagne ©2013



Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement





Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames





Background

- Code needs to be in memory to execute, but entire program rarely used
 - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
 - Program no longer constrained by limits of physical memory
 - Each program takes less memory while running -> more programs run at the same time
 - Increased CPU utilization and throughput with no increase in response time or turnaround time
 - Less I/O needed to load or swap programs into memory -> each user program runs faster





Background (Cont.)

- 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
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - More programs running concurrently
 - Less I/O needed to load or swap processes





Background (Cont.)

- Virtual address space logical view of how process is stored in memory
 - Usually start at address 0, contiguous addresses until end of space
 - Meanwhile, physical memory organized in page frames
 - MMU must map logical to physical
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation



Virtual Memory That is Larger Than Physical Memory





Operating System Concepts – 9th Edition



Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
 - Maximizes address space use
 - Unused address space between the two is hole
 - No physical memory needed until heap or stack grows to a given new page
- Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages readwrite into virtual address space
- Pages can be shared during fork(), speeding process creation





Silberschatz, Galvin and Gagne ©2013

Shared Library Using Virtual Memory





Operating System Concepts – 9th Edition



Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed \Rightarrow reference to it
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring to memory
- Lazy swapper never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a pager







Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
 - Need new MMU functionality to implement demand paging
- If pages needed are already memory resident
 - No difference from non demand-paging
- If page needed and not memory resident
 - Need to detect and load the page into memory from storage
 - Without changing program behavior
 - Without programmer needing to change code





Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (v ⇒ in-memory – memory resident, i ⇒ not-in-memory)
- Initially valid—invalid bit is set to i on all entries
- Example of a page table snapshot:



 During MMU address translation, if valid–invalid bit in page table entry is i ⇒ page fault





Page Table When Some Pages Are Not in Main Memory





Operating System Concepts – 9th Edition

Silberschatz, Galvin and Gagne ©2013



If there is a reference to a page, first reference to that page will trap to operating system:

page fault

- 1. Operating system looks at another table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory
- 2. Find free frame
- 3. Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory Set validation bit = v
- 5. Restart the instruction that caused the page fault







Operating System Concepts – 9th Edition

Silberschatz, Galvin and Gagne ©2013



Aspects of Demand Paging

- Extreme case start process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, nonmemory-resident -> page fault
 - And for every other process pages on first access
 - Pure demand paging
- Actually, a given instruction could access multiple pages -> multiple page faults
 - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
 - Pain decreased because of locality of reference
- Hardware support needed for demand paging
 - Page table with valid / invalid bit
 - Secondary memory (swap device with swap space)
 - Instruction restart





Instruction Restart

- Consider an instruction that could access several different locations
 - block move



- auto increment/decrement location
- Restart the whole operation?
 - What if source and destination overlap?





Performance of Demand Paging

- Stages in Demand Paging (worse case)
- 1. Trap to the operating system
- 2. Save the user registers and process state
- 3. Determine that the interrupt was a page fault
- 4. Check that the page reference was legal and determine the location of the page on the disk
- 5. Issue a read from the disk to a free frame:
 - 1. Wait in a queue for this device until the read request is serviced
 - 2. Wait for the device seek and/or latency time
 - 3. Begin the transfer of the page to a free frame
- 6. While waiting, allocate the CPU to some other user
- 7. Receive an interrupt from the disk I/O subsystem (I/O completed)
- 8. Save the registers and process state for the other user
- 9. Determine that the interrupt was from the disk
- 10. Correct the page table and other tables to show page is now in memory
- 11. Wait for the CPU to be allocated to this process again
- 12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction





- 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 $0 \le p \le 1$
 - if p = 0 no page faults
 - if p = 1, every reference is a fault
- Effective Access Time (EAT)

EAT = (1 - p) x memory access

+ p (page fault overhead

+ swap page out

+ swap page in)



Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 p) x 200 + p (8 milliseconds)

 $= (1 - p \times 200 + p \times 8,000,000)$

= 200 + p x 7,999,800

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 x p
 20 > 7,999,800 x p
 - p < .0000025
 - one page fault in every 400,000 memory accesses





Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
 - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
 - Then page in and out of swap space
 - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
 - Used in Solaris and current BSD
 - Still need to write to swap space
 - Pages not associated with a file (like stack and heap) anonymous memory
 - > Pages modified in memory but not yet written back to the file system
- Mobile systems
 - Typically don't support swapping
 - Instead, demand page from file system and reclaim read-only pages (such as code)



Copy-on-Write

- 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 the 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?
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
 - Designed to have child call exec()
 - Very efficient











After Process 1 Modifies Page C







- Used up by process pages
- Also in demand from the 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





- Prevent over-allocation of memory by modifying pagefault 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





Silberschatz, Galvin and Gagne ©2013



- 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



Operating System Concepts – 9th Edition

Silberschatz, Galvin and Gagne ©2013



- Frame-allocation algorithm determines
 - How many frames to give each process
 - Which frames to replace
- Page-replacement algorithm
 - Want lowest page-fault rate on both first access and re-access
- 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, the reference string of referenced page numbers is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1











First-In-First-Out (FIFO) Algorithm

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



- 15 page faults
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!
 - Belady's Anomaly
- How to track ages of pages?
 - Just use a FIFO queue





FIFO Illustrating Belady's Anomaly





Optimal Algorithm

- Replace page that will not be used for longest period of time
 - 9 is optimal for the example
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs







- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



- 12 faults better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?





LRU Algorithm (Cont.)

- 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
 - But each update more expensive
 - No search for replacement
- 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







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)
 - We do not know the order, however
- Second-chance algorithm
 - Generally FIFO, plus hardware-provided reference bit
 - Clock replacement
 - If page to be replaced has
 - Reference bit = 0 -> replace it
 - reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules





Second-Chance (clock) Page-Replacement Algorithm



Silberschatz, Galvin and Gagne ©2013

Operating System Concepts – 9th Edition



- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)
- 1. (0, 0) neither recently used not modified best page to replace
- (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
- Lease 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





Page-Buffering Algorithms

- Keep a 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
- Possibly, keep list of modified pages
 - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame 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





- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge i.e. databases
- Memory intensive applications can cause double buffering
 - OS keeps copy of page in memory as I/O buffer
 - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
 - Raw disk mode
- Bypasses buffering, locking, etc



End of Chapter 9

