Final Review

Monday, April 25, 2016

10:27 PM

SKIP:

* Anything about Solaris, Mach, Mobile OS
* Anything about Windows
* Enhanced clock algorithm
* Disk scheduling
* Flash file systems
* Paravirtualization

**Memory Hierarchy** (fastest to slowest)

* Registers
* L1 Cache (1 clock cycle)
* L2 Cache (4-5)
* L3 Cache (40)
* Main Memory (DRAM)
* Secondary Storage (Disk/Flash)
* Remote Network Disks

**Different types of caches**: write behavior

* **Write-through** on a cache hit
  + Changes to caches written immediately to memory
  + Good for consistency, bad - may take some time to complete
* **Write-back** on a cache hit
  + Lazy writes to memory at some later time
  + Good - fast response in writing, bad - inconsistency between caches and possible loss of cached data
* **Write-allocate** on a miss
  + Load into cache, update line in cache
  + Good if there is temporal/spatial locality
* **No-write-allocate** on a miss
  + Write immediately to memory

**Cache Replacement Policies**

* **LRU**
  + Increment counters for each cache item on each cache hit and zeroing the most recently fetched/hit item
  + Least recently used item has the highest counter
  + Most recently used items will have been zeroed more recently and have lower counter values
* Working Set: PSEUDOCODE, NO ACTUAL C

**Memory Management**

* **MMU**: maps from logical addresses to physical addresses, must do two things:
  1. Address translation: translate logical addresses into physical addresses
  2. Bounds checking: check if requested memory address is within upper and lower limits of the address space
     1. Base register in hardware keeps track of lower limit of physical address space
     2. Limit register keeps track of size of logical address space
     3. Upper limit of physical address space = base register + limit register
* Address Binding at Compile Time: If you know in advance where in physical memory a process will be placed, compile the code with absolute physical addresses
* Address Binding at Load Time: code is compiled in relocatable format. Then replace logical addresses in code with physical addresses during loading
* Address Binding at Run Time
  1. Code compiled in relocatable format as if executing in its own logical address space
  2. As each instruction is executed, MMU relocates logical address to physical address
* Static Linking
  1. Entire executable has all the code it needs at compile time
  2. Nothing can be changed in the code unless its recompiled
* Dynamic Linking
  1. Executable has only a stub that contains info on how to find a dynamically linked library function
  2. When hitting a stub, OS looks for dll file
* Swapping
* Swapping Difficulties
  1. Context-switch time of swapping is very slow
  2. Swapping of processes that are blocked or waiting is complicated
  3. Fragmentation becomes an issue
     1. Small chunks of non-contiguous unallocated memory
* Allocation strategies
  1. Best-fit: find smallest chunk that is big enough - results in more fragmentation
  2. Worst-fit: find largest chunk that is big enough
  3. First-fit: find first chunk that is big enough
  4. Next-fit: allocate based on a circular buffer

* **Paging**: solves issue of fragmentation
  + Each virtual page is mapped to a same sized physical frame, the frames don't have to be contiguous
  + Page table: keep track of where each logical page of a process is located in main memory
  + OS maintains a page table for each process
  + Given a logical address, MMU finds its logical page, then looks up physical frame in page table
  + Typical page size is 4-8kb
* **Paging**
  + No external fragmentation, but there is internal fragmentation
    - Process size of 4001b with page size of 4kb
* 3 ways of implementing page tables:
  + Dedicated bank of hardware registers or memory to store page table
  + Store page table in main memory
  + Cache a subset of page table mappings in a buffer (TLB)
* Paging and the TLB
  + MMU looks in TLB to find a match for a logical address
    - TLB Hit if there is a match
    - TLB miss if there is no match
      1. MMU invoked to convert logical to physical address
      2. Update TLB cache with new entry
* Sharing Pages: can share code/data pages between processes by having entries in different processes page tables point to the same physical page
* Hierarchical Paging: paging page tables
* Inverted Page Table: solution to large, sparse page tables
  + Have only one page table for all of memory, rather than one for each process
* Hashed Page Table: search fewer entries in either an inverted page table or regular page table

* **Segmentation**: another solution for fragmentation
  + Divide a process into variably sized segments that are organized according to some logical criteria
  + Each instruction in a segment has logical address: <segment #, offset>
  + Segments are of variable size
  + Drawback: makes remaining fragmentation even more complex

* **On-Demand Paging**: keep only a subset of a processes pages in memory
  + Not all pages in logical address space need to be kept in memory
  + Based on locality exhibited by most programs
* Virtual Memory Advantages when using On-Demand Paging:
  + Virtual address space can now exceed physical RAM
  + Can fit more processes in memory
  + Decreased swap times - there is less to swap
  + Can have large sparse address spaces, in which most of the address space is unused without taking up lots of physical RAM
* **Performance of On-Demand Paging** determined by Page Replacement Policies
  + Dirty/Modify Bit
    - Dirty bit initially set to 0
    - When a page in memory is written to, bit is set to 1
    - When a victim page is needed, pick a page with bit still = 0 (not yet modified)
  + Page Table Status Bits
    - Valid/invalid bit: accessing invalid page causes page fault
    - Dirty bit
    - R/W or Read-only bits: for memory protection
    - Reference bit: for clock algorithm

* **Page Replacement Policies**: needed for when all pages are dirty or there are a combination of clean and dirty bits (FIFO, LRU, OPT)
  + FIFO - easiest to implement but poorest performance
  + Belady's Anomaly: when the number of frames allocated to a task is increased (larger queue) you EXPECT fewer page faults, but actually get MORE page faults
  + OPT: replace page that will not be referenced for the longest time
  + LRU
    - LRU not subject to Belady's Anomaly
    - Implement with either History (sliding window/circular buffer) or Timer (time stamp for each page), Counters (counter incremented with any page reference, counter is reset to 0 on a reference), Linked List
  + Reference-Bit LRU (3 types)
    - Add an extra bit called a reference bit, set any time a page is referenced
    - Additional Reference-Bits Algorithm: record last 8 reference bits for each page
      1. Timer periodically shifts reference bits one position over
      2. LRU = lowest valued record
    - Second-Chance Clock Algorithm: find LRU using just one reference bit
      1. Pointer rotates around a circular queue
      2. If current frame's reference bit is zero, replace that page
      3. If its 1, set it to zero and keep going (second chance)
      4. Approximates LRU: of all 0 ref. bits, the one closest clockwise to the clock hand will have been in 0 state the longest (LRU)
      5. Simple to implement, but worst case that it has to rotate through entire buffer once before finding first victim frame
    - Enhanced Second-Chance Clock Algorithm: a dirty/modify bit is added to the reference bit and they are seen as a pair

**How to improve Page Replacement Performance**

1. Use a dirty/modify bit to reduce disk writes
2. Chose smart page replacement algorithm
3. Make search for the least important page be fast
4. Page-buffering: while reading in first frame, write out victim frame at a later time/when there is no other activity - faster perceived performance
5. Keep a pool of free frames and remember their content
6. Allocate appropriate number of frames so a process avoids thrashing

**Memory Allocation Policies**

1. Determine number of frames that allow a process to allocate
2. Split all the frames equally among all processes
3. Allocate number of frames relative to the size of each process

**Local vs. Global Allocation/Replacement**

* Problems with local allocation:
  + behavior of processes may change as they execute
  + Local replacement isolates a processes paging behavior from other processes
* Global Allocation/Replacement
  + Pool all pages from all processes together
  + When a page needs to be replaced/evicted, choose it from the global pool of pages

**Thrashing**: repeated page faulting when a processes allocated number of frames is less than the size of its recently accessed set of frames

* Process spends more time page faulting to disk than executing

**Working Set**: solution to thrashing

* Programs tend to exhibit locality of behavior. Find a set of LRU pages that captures this locality. Allocate to a process the size of the working set
* If working set is too small, it won't capture locality of the process. If it is too large, it will capture frames that aren't relevant to the local behavior of the process

**Page Fault Frequency Solution**: another solution to thrashing

* When PFF reaches an upper limit, increase number of frames allocated to the process
* When PFF reaches lower limit, decrease number of frames

**Linux Global Page Replacement Policy**

* Clock-like LRU to entire pool of all processes pages - evicted page is the LRU globally over all processes
* As process needs more pages, the working set expands
* Effectively/adaptively allocates in an implicit way the working set to each process

**Memory-Mapped Files**: map some parts of a file on disk to pages of virtual memory

* Normal read/writes of a file require sys call and read/write to/from disk
* Instead, it's faster and simpler if the file could be loaded into memory so that r/w is to RAM

1. Obtain handle to file by creating/opening it
2. Reserve virtual addresses for file
3. Declare portion of file to be memory mapped (functions of OS)
4. When file is first accessed, its demand paged into physical memory
5. Subsequent r/w accesses to file are served by physical memory

* Advantages: after first access, all subsequent r/w are fast
  + Multiple processes can map the same file concurrently
* Writes to a file in memory can result in momentary inconsistency between the file cached in memory and the file on disk

**File Systems**

* File attributes (name, ID, size, chmod) are stored in a file header or file control block (FCB) or inode.
* System calls to manipulate files (create, read, write, open, close, delete, truncate, append, rename)
* Tree-Structured Directory
  + Hierarchical, unique naming (foo/file vs bar/file), users can share/access files in other directory)
* Sharing Directories/Files: **Symbolic Links**: pointer to a directory entry that points to a file/directory
  + Not a file, just a pointer to a file
  + Deleting a file pointed to by a link can leave the link dangling
* **Mounting File Systems**
  + Virtual File System (VFS) abstracts file representation (if mounted FS is different than OS FS)

**File System Implementation**

* Four main file system components in memory:
  1. Recently accessed parts of the directory structure tree
  2. System-wide open file table (OFT) that tracks process-independent info of open files
  3. Per-process OFT that tracks all files that have been opened by a particular process
  4. Mount table of devices with file systems that have been mounted as volumes
* When a process calls open(file.txt)
  1. Directory structure is searched for the file file.txt
  2. When file is found, directory entry contains a pointer to the FCB on disk
  3. Add an entry to the per-process OFT that points to the file's FCB in the system OFT
  4. Return to process
* On close()
  1. Remove entry from the per-process OFT
  2. Decrement the open file counter for this file in the system OFT
* File Allocation (1): Contiguous Allocation - fast performance but bad fragmentation
* File Allocation (2): Linked Allocation
  1. Each file is a linked list
  2. To add to a file, modify the linked list in the middle or at the tail, depending on where you want
  3. To read file, traverse linked list
  4. Advantages:
     + Solves problem of contiguous allocation - no external fragmentation
     + Minimal overhead in file header, just need a pointer to start of file on disk
     + Good for sequential read/write data access
     + Easy to insert data into middle of list
  5. Disadvantages:
     + Performance is sow for r/w - you must traverse list each time
     + Fragile reliability - if one pointer gets corrupted, rest of data after that pointer is lost
* File Allocation (3): **File Allocation Table (FAT)**
  1. Instead of having pointers of linked list with the file data blocks themselves, separate the pointers out and put them in a special table (FAT), located in section of disk at the beginning
  2. Entries in the FAT point to other entries in the FAT as a linked list
  3. End of file has a special EOF value
  4. Allocating a new block - just find the first 0-valued block
  5. Advantages:
     + R/W faster than the pure linked list
  6. FAT very similar to inverted page tables (except without the pointers)
* File Allocation (4): Indexed Allocation
  1. Collect all pointers into a list or table called an index block
  2. Unlike FAT, index block can be stored in any block on disk, index is just a linear list of pointers
  3. PROBLEM: how big should the index block be, SOLUTION: Multilevel index (like hierarchical page tables)
* File Allocation (5): Multilevel Indexed Allocation
  1. PROBLEM: accessing small files takes just as long as large files
* File Allocation (6): **UNIX Multilevel Indexed Allocation**
  1. If there are 15 entries in the index block:
     + First 12 entries are pointers to direct blocks of file data on disk
     + 13th pointer points to a singly indirect block, which is an index block pointing to disk blocks
     + 14th pointer points to a doubly indirect block
     + 15th pointer points to a triply indirect block
  2. Accessing small files are fast, accessing large files are slow
  3. This ENTIRE hybrid index block is stored with the file inode

**File Allocation vs. Process Allocation**

* In both cases, mapping an entity to storage
* Differences:
  + Address spaces are fixed in size and known in advance, whereas files grow/shrink over time
    - Files need an allocation system that is more flexible than page tables, which cant grow
  + Address spaces can be spare and mostly unused whereas file data is all used
* Similarities:
  + FAT is very similar to IPT except FAT has pointers to the next block
  + Indexed allocation looks very much like a page table, except there's room for growth

**Free Space Management**

* File system needs to keep track of what blocks of disk are free or allocated

1. Bit Vector/Bit Map
   * Each block represented by a bit
   * Concatenate all bits into an array of bits, the j bit indicates the j block is allocated
2. Linked List
   * Link together all free blocks, more efficient and faster than bit map
3. Grouping
   * Linked list except store pointers to free blocks in each list block
   * Last block points to next list block containing more free pointers
4. Counting
   * Grouped link list, with a field to each pointer entry that indicates number of free blocks immediately after the block pointed to

**Fault Recovery**

* Asynchronous writes produce inconsistency between in memory and on disk file system
* Even if writes are synchronous, issues can occur if the disk writes fail halfway through an operation

**Log-Based Recovery**

* Each operation is written to log before operation is actually performed on data on disk
* FS has a sequence of records of operations that were intended if there is a crash
* Operations are grouped into sets called transactions - transactions are atomic (either succeed entirely or not at all)
* Transaction is not considered complete until it has been committed (T1 <commit>)
* Completed transactions: all full transactions prior to a checkpoint (committed to log, AND written to disk)
* Committed transactions: have been committed to log, not yet written to disk, redo() after crash
* On failure:
  + OS looks for latest checkpoint
  + Redo() committed transactions
  + Undo() partial transactions

Flash vs. Spinning Disk

**RAID** (Redundant Arrays of Inexpensive Disks)

* RAID0: data striping with no redundancy
* RAID1: no data striping, simple mirror of disk
* RAID0+1: data is striped, and mirroring of stripes for redundancy

Machine generated alternative text:
Data block 
Striping 
Mirror the stripe 

* RAID1+0: each data stripe is split into separate disks, each of those separate disks has a mirrored disk

Machine generated alternative text:
Data block 
Striping 

* RAID2: adds error correction code bits

Machine generated alternative text:
Data block 
Striping 
ECC 
Computation 

* RAID3: bit-interleaved parity

Machine generated alternative text:
Data block 
Striping 
Parity 
Disk 

* RAID4: block interleaved parity

Machine generated alternative text:
Data block 
Striping 
Parity 
Disk 

* RAID5: block interleaved distributed parity

Machine generated alternative text:
Data block 
Striping 
Universlty Of CO Ora 0 Bou 
Avoids overuse of RAID4' s 
parity disk, and is the most 
common parity RAID system 
= Parity Blocks 
RAID6 replaces paritn 
gee 
Reed-Solomon ECC tur 
multiple disk failures 

**Networking**

* Socket API used to by all network applications to send and receive
* Socket library talks via system call API to OS kernel's network stack
* Network stack consists of
  + Physical Layer 1 | Data Link/Mac Layer 2 | Network Layer 3 | Transport Layer 4 <SOCKET API> Application Layer 5
* Protocols:
  + Physical Layer 1: CDMA OFDM
  + Data Link/Mac Layer 2: Ethernet, Wifi, Bluetooth
  + Network Layer 3: IPv4, IPv6
  + Transport Layer 4: TCP, UDP
  + Application Layer 5: HTTP, SMTP, IMAP
* To send data: go from app to transport to network to data link to physical layer
* To receive data: go from physical layer, to data link, to network, to transport, to application layer
* **TCP**: ensures in-order delivery, pipe abstraction through network, reordering not possible
* **UDP**: not as reliable as TCP but much faster
* **IP**: route the packet through many routers across internet to correct destination
  + Tries to find the shortest path, Dijkstra's Algorithm

**Network File System (NFS)**

* Virtual File systems operate on an NFS client
* Files are abstracted as vnodes
* NFS v4
  + Both client and server can cache data
  + Statefulness at the server - keep track of open() files

**Remote Procedure Calls (RPC)**

* Allows remote execution of a function call. Read() executes on the server and its results are sent back to the client

**Virtual Machines (VM)**

* Extend idea of abstraction of memory (virtual memory) and CPU (time slicing) to hardware
* Hypervisor: provides virtualization layer for guest OS and resides just above the hardware
* Privileged instructions invoked by guest OS are trapped by the hypervisor, which emulates the instructions
* Intel:
  + Ring 0: hypervisor
  + Ring 1: guest OS
  + Ring 3: user mode processes
* **Shadow Page Tables**
  + Guest OS virtual addresses must be translated to actual physical pages in memory
  + Hypervisor creates a shadow page table for each of the guest OS's page tables
* Intel VT-x: hardware support for virtualization
  + Root mode: Hypervisor executes in root mode in ring 0
  + Guest OS executes in non-root mode in ring 0

More

Thursday, April 28, 2016

11:03 AM

* Don't have to implement working set
* At least one theoretical working set question
* Will have to implement LRU

* Encryption
* Flipping individual bits

* Completely understand unix iNode
* Must draw network layer protocol stacks
  + Examples of each layer, draw all layers
  + Theoretically come up with a protocol at a particular layer

* One particular strategy for free space
  + How do we know what blocks are free? Bit vector, bit mask, grouping blocks together
* Indexed allocation
* Multilevel indexed allocation
* File allocation table

* Why you cant contiguously allocate memory - fragmentation
* Sockets api - user mode to kernel to network stack
* RAID levels
* One question about NFS or Samba
* Nothing about deadlocks

* Unix iNode
  + Small files should open up fast
  + Large file should open up more slowly
  + Must have levels of indirection
  + iNode structure (filled top down)
    - [file attributes (size, permissions, etc)]
    - [Direct Blocks]
    - [Single indirect]
    - [Double indirect]
    - [Triple indirect]
  + iNode gets slower the larger the data gets

* FAT
  + Extension of the linked list idea (part of the block is a pointer to the next block)
  + Instead of pointers in the block, pointers go into a special table (FAT) in a different part/partition of the disk
  + VERY SIMILAR to inverted page table