Chapter 6

The point of scheduling
- Maximize CPU utilization by making sure a program is being executed at all times.

6.1.1 CPU-i/o burst cycle
- Process execution consists of a cycle of CPU execution and I/O wait
- Process start with CPU burst and then followed by any range of I/O burst ending with CPU burst and system call to terminate

6.1.2 CPU scheduler
- When the CPU becomes idle OS selects one process out of ready queue to be executed
- Short term scheduler selects process from the ready queue and allocates the CPU to that process
- The records in the queue are PCBs of processes

6.1.3 preemptive scheduling
- CPU scheduling takes place under the following four circumstances
  - Process switches from the running state to the waiting state
    - Like a i/o request of wait()
  - Process switches from the running state to a ready state
    - Like an interrupt occurs
  - Process switches from the waiting state to the ready state
    - Like completion of i/o
  - Process terminates
- Nonpreemptive scheduling (cooperative)
  - Scheduling only takes place when process terminates or switches from running to waiting state
- Preemptive
  - Scheduling can take place at any time

6.1.4 dispatcher
- Dispatcher
  - Module that gives control of the CPU to the process selected by the short term scheduler
    - Involves switching context, switching to user mode, jumping to proper location in the user program to restart that program
- Dispatcher needs to be as fast as possible because it is invoked at every process switch
- Dispatcher latency = the time it takes to stop one process and start another

6.2 scheduling criteria
- CPU utilization
  - Keep the CPU as busy as possible, ranges from 40% light to 90% heavy
- Throughput
  - Number of processes completed per time unit
- Turnaround time
  - Interval from the time of submission to the time of completion
- Waiting time
  - Amount of time spent waiting in the ready queue
- Response time
  - Time from the submission of a request until the first response is produced

6.3 scheduling algorithms

CPU scheduling deals with he problem of deciding which of the processes in the ready queue is to be allocated to the CPU

6.3.1 FCFS first come first served scheduling

Simplest CPU scheduling algorithm
Process that request the CPU first is allocated CPU first
When process enters ready queue it is managed with a FIFO queue, PCB is linked to tail of queue
Average waiting time usually long
Prone to convoy effect
    Large CPU burst time artificially increases the average waiting time because processes are completed as they come without interruption
Nonpreemptive

6.3.2 SJF shortest job first scheduling

Assigns the CPU to the process in the ready queue that will have the shortest next CPU burst
If the CPU burst time is tied FCFS is used to tie break
Used frequently in long term scheduling
Next CPU burst predicted as exponential average of the measured lengths of the previous CPU burst
$Theta_{n+1}$ = alpha t_n + (1-alpha) $theta_n$
As alpha approaches 0 recent history does not count
As alpha approaches 1 only the actual last CPU burst counts
Can be nonpreemptive
    Once assigned the process will run until it terminates
Or preemptive
    If a process arrives and its predicted next CPU burst is shorter than the current process switch

6.3.3 priority scheduling

Each process is associated with a priority
CPU allocated to process with highest priority
Equal priority processes are decided by FCFS
System differ on wether 0 represents the highest or lowest priority
Internally defined priorities
    Use measurable qunatity or quantities to compute priority of process
Externally defined priorities
    Set by criteria of outside the OS people shit
Can possibly leave low priority processes running indefinitely
    Solved by aging, gradually increasing the priority of process that wait in the system for a long time
    Decay the opposite effect

6.3.4 round robin scheduling

Designed for time sharing systems
Preemptive
Ready queue is treated as a circular queue
Time quantum is a small unit of time chosen by the algorithm to switch between process
CPU scheduler goes around the queue allocating the CPU to each process for a time interval of up to 1 time quantum
Ready queue treated as a FIFO
If process has a execution time less than time quantum, terminates gives back CPU
If time quantum is up, context switch
Each process waits no longer than (n - 1) x q time units until its next execution where q is the time quantum and n is the number of processes
Context-switch time should be fraction of q and 80% of CPU burst should be shorter than q

6.3.5 multilevel queue scheduling
- Partitions ready queue into several separate queues
- Processes are permanently assigned to one queue based on memory, size, p-priority, or p-type
- Foreground (interactive) processes
- Background (batch) processes
- Each queue has its own scheduling algorithm
- The queues themselves must be scheduled
    - Fixed-priority preemptive scheduling
- Low priority queues can not allocate processes until high priority queues are empty

6.3.6 multilevel feedback queue scheduling
- Allows for process to move between queues
- Separate processes according to the characteristics of their CPU burst
- If a process uses to much CPU time it is moved to a lower priority queue
- Process that waits to long will be moved up to a higher priority queue
- Defined by
    - Number of queues
    - Scheduling algorithm for each queue
    - Method used to determine when to upgrade process to higher priority queue
    - Method used to determine when to demote process to lower priority queue
    - Method used to determine which queue a process will enter when that process needs service

6.4. thread scheduling
- Scheduling issues involving user-level and kernel-level threads

6.4.1 contention scope
- Many-to-one and many-to-many models
    - (PCS) Process-contention scope
        - Usually done according to priority scheduling, preemptive but not time slicing.
        - Thread library schedules user-level threads to run on an available LWP
        - Competition occurs only between threads belonging to the same process
    - (SCS) system contention scope
        - SCS scheduling decides which kernel thread to connect to CPU
        - Competition occurs between all threads in the system
- One-to-one model
    - Uses only SCS

6.7 operating scheduling
- Descriptions of the scheduling policies of linux, windows, and solaris
    - Process scheduling means in
        - Solaris and windows
            - Scheduling of Kernel-threads
        - Linux
            - Scheduling of Tasks

6.7.1 example: linux scheduling

- Before version 2.5 kernel ran a variation of traditional UNIX scheduling algorithm
- Then moved to O(1) scheduling 2.6
  - Excellent performance SMP systems, poor response time on interactive processes
- Moved to Completely fair scheduler in 2.6.23
  - Scheduling classes
    - Two standard classes
      - Default scheduling class using the CFS scheduling algorithm
        - Assigns proportion of CPU processing time to each task
        - Calculated based on nice value assigned to each task
          - Nice value range from -20 to +19
            - Lower the numerical value the higher the priority
        - Targeted latency
          - Interval of time during which every runnable task should run at least once
          - Proportions of CPU time allocated from value
          - Has default and minimum
          - Can increases if number of active task reaches a threshold
        - Assigns priority via virtual runtime
          - Lower-priority tasks have higher rates of decay than higher-priority tasks
          - Vruntime increments slower for higher priority tasks
        - Scheduler chooses the next task to run based on lowest Vruntime
      - Real time scheduling class
        - POSIX STANDARD
    - Two priority ranges
      - One fore real time tasks and a second for normal tasks
        - Real time tasks are assigned static properties from 0-99 and normal from 100-139
        - Default -20 nice value maps to 100 and +19 maps to 139
        - These map onto a global priority scheme with lower values are higher priority

- 6.7.2 examples windows scheduling
  - Schedules threads using a priority based, preemptive, scheduling algorithm
    - Ensures that highest priority thread always runs
    - Dispatcher
      - Part of kernel that handles scheduling
    - 32-level priority schemes
      - 0, lowest priority
        - Memory management thread
      - 32, highest priority
      - Divided into two classes
        - Variable class
          - IDLE_PRIORITY_CLASS
            - Normal Priority 4, max priority 15, minimum priority 1
          - BELOW_NORMAL_PRIORITY_CLASS
            - Normal priority 6, max priority 15, minimum priority 1
          - NORMAL_PRIORITY_CLASS
            - Normal priority 8, max priority 15, minimum priority 1
          - ABOVE_NORMAL_PRIORITY_CLASS
            - Normal priority 10, max priority 15, minimum priority 1
          - HIGH_PRIORITY_CLASS
            - Normal priority 13, max priority 15, minimum priority 1
        - Real-time class
          - REALTIME_PRIORITY_CLASS
            - Normal priority 24, max priority 31, minimum priority 16
      - Dispatcher uses a queue for each priority and traverses the set of queues form highest to lowest.
      - If no ready thread found executes idle thread
      - Relative priority within each priority class
        - IDLE
        - LOWEST
        - BELOW_NORMAL
        - NORMAL
        - ABOVE_NORMAL
        - HIGHEST
        - TIME_CRITICAL
      - Priority of each thread is based on the priority class and its relative priority
      - When a threads time quantum runs out the thread is interrupted
        - If thread variable
          - Priority lowered but never below the NORMAL level of that class
      - When a thread is released from wait operation
        - If thread variable
          - Priority raised depending on what it was released from
            - i/o; large increase
            - Disk operation; moderate increase
      - Special rule for NORMAL PRIORITY CLASS
        - If a foreground process (process currently selected on the screen)
          - Increases quantum by factor of usually 3
        - If background process
          - Nothing happens
      - (UMS) user-mode scheduling
        - Allows application to create and manage threads independently of kernel
        - More efficient for large number of threads

6.7.3 example: solaris scheduling
- Priority based- thread scheduling
  - Six classes
    - Time sharing (TS)
    - Interactive (IA)
    - Real time (RT)
    - System (SYS)
    - Fair Share(FSS)
    - Fixed priority (FP)
- Threads can only be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is a multilevel feedback queue
- Converts class specific priorities into per thread global priority
  - Thread with highest priority runs next
  - Runs till blocked, uses time slice, or preempted by higher priority thread
  - Multiple threads with same priority are round robined

6.8 algorithm evaluation
- Criteria
  - Maximizing CPU utilization under the constraint that the maximum response time is 1 second
  - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

6.8.1 deterministic modeling
- Analytic evaluation
  - Uses given algorithm and the system work load to produce a formula or number to evaluate the performance of the algorithm for that workload
- Deterministic modeling
  - Takes a particular pre-determined workload and defines the performance of each algorithm for the work that work load

6.8.2 queueing models

6.8.3 simulations
- Runs mode of computer system
- Uses clock as a variable
- Gathers large amount of statistics to examine performance
- Data that simulation uses is derived from
  - Random number generators, according to probabilities
  - Distributions derived either mathematically or empirically
  - Trace tapes, record real sequences of real events in real system

6.8.4 implementation
- Implement the scheduler in a real system
  - High cost, high risk
  - Environments vary
- Most schedulers can be modified per site or per system
- Or use API to modify priorities

Chapter 7 main memory

7.1 background

Memory consists of a large array of bytes, each with its own address

7.1.1 basic hardware

Program must be brought (from disk) into memory and placed within a process for it to be run
Main memory and registers are only storage CPU can access directly
Memory unit only sees a stream of addresses + read requests, or addresses + data and write Requests
Register access in one CPU clock (or less)
Main memory can take many cycles, causing a stall
Cache sits between main memory and CPU registers
Protection of memory required to ensure correct operation
Base and limit registers

A pair of base and limit registers define the logical address space
CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

7.1.2 address binding

Programs on disk, ready to be brought into memory to execute form an input queue

Without support, must be loaded into address 0000

Inconvenient to have first user process physical address always at 0000
Further addresses represented in different ways at different stages of a program's life

Source code addresses usually symbolic
Compiled code addresses bind to relocatable addresses

I.e "14 bytes from beginning of this module"

Linker or loader will bind relocatable addresses to absolute addresses

I.e 74014

Each binding maps one address space to another

Binding of instructions and data to memory

Address binding of instructions and data to memory addresses can happen at three different stages

Compile time:

If memory location known a priori, absolute code can be generated; must recompile code if starting location changed

Load time:

Must generate relocatable code if memory location is not known at compile time

Execution time:

Binding delayed until run time if the process can be moved during its execution from one memory segment to another
Need hardware support for address maps (like, base and limit registers)

7.1.3 logical vs. Physical address space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address
    - Generated by the CPU; (virtual address)
  - Physical address
    - Address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical and physical address differ in execution-time address-binding scheme
- Logical address space
  - Set of all logical addresses generated by a program
- Physical address space
  - Set of all physical addresses generated by a program
- (MMU) memory-management unit
  - Hardware device that at run time maps virtual to physical address
  - Many methods possible, covered in the rest of this chapter
  - To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
    - Base register now called relocation register
    - MS-DOS on intel 80x86 used 4 relocation registers
  - The user program deals with logical addresses; it never sees the real physical addresses
    - Execution-time binding occurs when reference is made to location in memory
    - Logical address bound to physical addresses
- Dynamic relocation using relocation register
  - Routine is not loaded until it is called
  - Better memory-space utilization; unused routine is never loaded
  - All routines kept on disk in relocatable load format
  - Useful when large amounts of code are needed to handle infrequently occurring cases
  - No special support from the operating system is required
    - Implemented through program design
    - OS can help by providing libraries to implement dynamic loading
- Dynamic linking
  - Static linking
    - System libraries and program code combined by the loader into the binary program Image
  - Dynamic linking
    - Linking postponed until execution time
  - Small piece of code, stub, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine
  - Operating system checks if routine is in processes' memory address
    - If not in address space, add to address space
  - Dynamic linking is particularly useful for libraries
  - System also known as shared libraries
  - Consider applicability to patching system libraries
    - Versioning may be needed

7.2 swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- Backing store
  - Fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Roll out, roll in
  - Swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a ready queue of ready-to-run processes which have memory images on disk
- Does the swapped out process need to swap back in to same physical addresses
  - Depends on address binding method
    - Plus consider pending i/o to / from process memory space
- Modified versions of swapping are found on many systems
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold
- Context switch time including swapping
  - If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
  - Context switch time can then be very high
  - 100MB process swapping to hard disk with transfer rate of 50MB/sec
    - Swap out time of 2000ms
    - Plus swap in of same sized process
    - Total context switch swapping component time of 4000ms
  - Can reduce if reduce size of memory swapped - by knowing how much memory is really being used
    - System calls to inform OS of memory use via request_memory() and release_memory()
  - Other constraints as well on swapping
    - Pending i/o
      - Can’t swap out as i/o would occur to wrong process
    - Or always transfer i/o to kernel space, then to i/o device
      - Double buffering, adds overhead
  - Standard swapping not used in modern operating systems
    - But modified version common
      - Swap only when free memory extremely low

7.3 contiguous allocation

- Main memory divided into two partitions
  - Resident operating system
    - In low memory with interrupt vector
  - User process memory
    - For user processes, in higher memory

7.3.1 memory protection

- Prevent process from accessing memory it does not own
  - Limit register
    - Contains range of logical address
  - Relocation register
    - Contains the value of the smallest physical address
- Transient operating-system code; comes and goes as needed (like device drivers)

7.3.2 memory allocation

- Two methods
  - Multiple partitions method
    - Memory divided into fixed sized partitions
    - Each partition can have one process
    - Degree of multi programing is bound by number of partitions
    - When partition is free process is selected from input queue and is loaded into the free partition
    - When process terminates, partition becomes available for another process
    - Used primarily in batch system
  - Variable partition method
    - OS keeps a table indicating which parts of memory are available and which are occupied
    - Hole
      - Available memory
    - At start all of memory is one big hole
    - Memory is allocated to processes according to their size
    - If two holes meet they merge into a larger hole
    - If a process is allocated to too large a hole, the excess is released
    - If there are no more holes large enough to fit the process
      - System waits for a hole large enough to appear
      - Or seeks through input queue for process small enough to fit available holes
    - Three strategies
      - First fit
        - Allocate to first hole in set that is big enough
        - Searching either starts at beginning, or at the location of the end of the last search, stop searching when hole big enough found
      - Best fit
        - Allocate the smallest hole that is big enough
        - Search entire list unless list is ordered by size
        - Smallest sized left over holes
      - Worst fit
        - Allocates the largest hole
        - Search entire list unless sorted by size
        - Largest sized left over holes

7.3.3 fragmentation

- External fragmentation
  - When there are holes separated by process that are two small for any process, but combined could be large enough for a process
- Internal fragmentation
  - Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given N blocks allocated, .5 N blocks may be lost to fragmentation
  - 1/3 may be unusable - 50 percent rule

7.4 segmentation

- Mechanism that maps programmers view to actual memory

7.4.1 basic method

- Segmentation is the division of a program memory into units called segments
- Segments are two tuple, they have a name (number) and a length
  - <segment-number,
    - Label of segment
  - offset>
    - Location within the segment

7.4.2 segmentation hardware

- Maps two tuple segment to one tuple memory array
- Logical address has two parts
  - Segment number
    - Maps to the segment table
  - Offset
    - Where in that segment the reference is
- Segment table
  - The system that maps segments to memory
  - Three components
    - Segment number
      - Used to identify segment
    - Limit
      - Length of segment
    - Base
      - Segment start location in memory
- If the offset is greater than the limit than there is a trap
- If the segment number does not exist in the segment table there is a trap
- After clearing those requirements the base and the offset are added up to find the actual location in memory
- Segment-table base register (STBR)
  - Points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program
- Protection
  - With each entry in segment table associate:
    - Validation bit = 0 -> illegal segment
    - Read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

7.5 paging

- Memory-managment scheme that avoids external fragmentation and compaction

7.5.1 basic method

- Breaks physical memory into fixed-sized blocks called frames
- Breaks logical memory into fixed-size blocks called pages
- At execution pages loaded into any available memory frames
- Logical address divided into two parts
  - Page number
    - Index into a page table
  - Page offset
    - Displacement within the page
- Page table
  - Two tuple table
    - Page number
      - Maps from logical address to page table
    - Frame number
      - Maps from page table to physical frame
- Physical address divided into two parts
  - Frame number
    - Mapped from page table
  - Page Offset
    - Part of original logical address
- Frame table
  - Data structure that manages frames in OS
  - One entry for each physical page frame
  - Each entry has info on whether its free or what page of which process is in it
- OS has to keep a copy of a page table for each process
- The way it works
  - Logical address is given
  - Integer division done to find the page number
    - Entry found in page table
      - Frame number is put in physical address
  - Mod done to find the page offset
    - Page Offset is put in physical address
  - Physical address is used find location in memory
- Processes can only access their own memory

7.5.2 hardware support
- Page table is kept in main memory
- Page pointer scheme
  - Page-table base register (PTBR)
    - Points to the page table
  - Page-table length register (PTLR)
    - Indicates size of the page table
  - Every data/instruction access requires two memory accesses
    - On for page table and one for data/instruction IS NO GOOD
- Translation look-aside buffers (TLBs)
  - Special hast-lookup hardware cache
    - Can be used in the pipeline so it takes no extra time to look up
    - TLB miss
      - Entry not found in TLB
      - Must look in actual page table
      - Entry added to TLB
      - If no space one is removed
        - Either least recently used (LRU) round-robin or random
    - TLB entries can be wired down
      - Cannot be removed from TLB
      - Used for key kernel code
  - Address-space identifiers (ASIDs)
    - With this feature TLB entries can uniquely identify each process to provide address-space protection for that process
    - Without this in place the content of the TLB must be flushed with ever context switch
  - Effective access time EAT
    - The access time for a look up
    - Hit ratio
      - Percentage of times that a page number is found in the associative register; Also alpha
    - EAT equation
      - (1+epsilon) alpha +(2 + epsilon) (1-alpha)
        - 2 + epsilon - alpha

7.5.3 protection
- Protection bit added to each frame to indicate if read-only or read-write access is allowed
  - Add more bits to indicate page execute only
- Valid-invalid bit
  - A bit attached to each entry in the page table
    - Valid indicates that the page Is in the process' logical address space and thus legal
    - Invalid indicates that the page is not in the process' logical address space
- Page-table length register (PTLR) can also be used by checking every logical address to makes sure its all good
- Violations result in a trap in kernel

7.5.4 shared pages
- Shared code
  - One copy of read-only (reentrant) code can be shared among processes
  - Useful for interprocess communication if sharing of read-write pages is allowed
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

7.6 structure of the page table
- Structuring page tables

7.6.1 hierarchical paging
- Break up the logical address space into multiple page tables
- Simple technique is a two-level page table
- We then page the page table
- Forward-mapped page table
  - working from the outer page table inward
- Straight page table
  - Logical address 32-bit machine 1k page size
    - Page number 22 bits
    - Page offset 10 bits
- Hierarchical page table
  - Logical address 32-bit machine 1k page size
    - Page number 12 bits
      - Index into the outer page table
      - Page offset 10bits
        - Displacement within the inner page table page
    - Page offset 10bits
      - Final offset in the page stored in the entry in the inner page table stored in the entry of the outer page table
  - Logical address 64-bit machine 1k page size
    - Need to extend to three level paging scheme
      - Outer outer page l outer page l inner page l page offset
    - Not good
  - Hierarchical not recommended for 64 bit machine

7.6.2 hashed page table
- Common in address spaces greater than 32 bits
- Virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains
  - A virtual page number
  - Value of the mapped paged frame
  - A pointer to the next element
- Linked list is searched for matching virtual page number
  - If match found physical frame is extracted
- Clustered page table
  - Used for 64 bit addresses
    - Similar to hashed but each entry refers to several pages
    - Especially useful for sparse address spaces (memory scattered non-contiguous)

7.6.3 inverted page tables
- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash tables to limit the search to one or at most a few page table entries
  - TLB can accelerate access
- Shared memory - One mapping of a virtual address to the shared physical address

7.7 intel 32 and 64 bit architectures

- Dominant industry chips
- Pentium CPUs are 32 bit and called IA-32 architecture
- Current intel CPUs are 64 bit and called IA-64 architecture
- Many variations in the chips, main ideas here

7.7.1 IA-32 architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4GB
  - Up to 16k segments per process
  - Divided into two partitions
    - First partition of up to 8k segments are private to process
      - Kept in local descriptor table (LDT)
    - Second partition of up to 8k segments are shared among all processes
      - Kept in global descriptor table (GDT)
- CPU generates logical address
  - Logical address <selector, offset>
  - Selector given to segmentation unit
    - First 13 bit identify segment
    - Next 1 bit identify if LDT or GDT
    - Last 2 bits deal with protection
    - Produces linear addresses
      - Linear address given to paging unit
        - Generates physical address in main memory
        - Paging units form equivalent of MMU
        - Pages sizes can be 4 KB or 4 MB
- CPU
  - Creates logical address
  - Passes it to segmentation unit
    - Creates linear address
    - Passes it to paging unit
      - Creates physical address
      - Finds it in memory
- Intel IA-32 page address extensions
  - 32 bit address limits led intel to create page address extension (PAE), allowing 32 bit apps access to more than 4GB of memory space
    - Paging went to a 3-level scheme
    - Top two bits refer to a page directory pointer table
    - Page-directory and page-table entries moved to 64 bits in size
    - Net effect is increasing address space to 36 bits - 64 GB of physical memory
- Bits
  - 31-30
    - Page directory pointer table
  - 29-21
    - Page directory
  - 20-12
    - Page table
  - 11-0
    - Offset

Intel x86-64

- 64 bits capable only uses 48 bit addressing normally
  - Page sixes of 4kb, 2mb, 1gb l four level paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits

Chapter 8 virtual memory

8.1 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 means more programs run at the same time
    - Increased CPU utilization an throughput with no increase in response time or turn around time
  - Less i/o needed to load or swap programs into memory means each user program runs faster
- 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
- 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
  - 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 two is hole
      - No physical memory needed until heap or stack grows to given 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 read-write into virtual address space
  - Pages can be shared during fork(), speeding process creation

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

8.2 demand paging

- Pages only loaded when they are demanded
- Lazy swapper
  - Never swaps a page into memory unless the page will be needed
- Pager is a swapper for pages

8.2.1 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 means in memory
    - i means either invalid or not in memory
  - Initially valid-invalid bit is set to i on all entries
  - During MMU address translation, if valid-invalid bit in page table entry is i you page fault
- Page fault
  - If there is a reference to a page, first reference to that page will trap
  - OS checks internal table
    - Invalid reference OS aborts
    - Not in memory
      - Find free frame
      - Swap page into frame via scheduled disk operation
      - Reset tables to indicate page now in memory and set Valid bit to v
      - Restart the instruction that caused the page fault
- Aspects of demand paging
  - Pure demand paging
    - No pages in memory to start
    - OS sets instruction point to first instruction of process, non-memory-resident
      - Page fault
    - Does same thing for every other process pages on first access
  - A given instruction could access multiple pages and have multiple page faults
    - Consider fetch and decode 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
  - Microcode computes and attempts to access both ends of both blocks, forcing a page fault to happen before anything is modified, if a page fault happens at all
  - Use temporary registers to hold values of overwritten locations, write them back if there is a page fault

8.2.2 performance of demand paging
- Three major activities of page faults
  - Service the interrupt
  - Read the page
  - Restart the process
- Page fault rate 0<p<1
  - If p = 0 no page faults
  - If p = 1 every reference a page fault
- Effective access time
  - (1- p) X memory access + p X page fault time <page fault overhead+swap page out+swap page in>
- 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
  - Demand page in from program binary on disk, but discard rather than paging out when freeing frame
    - Still need to write to swap space
      - Pages not associated with a file
        - Anonymous memory
      - Pages modified in memory but not yet written back to file system

8.3 copy-on-write
- On fork() 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 over
- 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
    - Dont 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()

8.4 page replacement
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Dirty bit
  - A bit that is set if the page has been modified and thus need to be written on disk
- Page replacement completes separation between logical memory and physical memory
  - Large virtual memory can be provided on a smaller physical memory
- Page and frame replacement algorithms
  - 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

8.4.2 FIFO page replacement
- Queue style
  - Head element replaced each time
  - New element added at tail
- Belady's anomaly
  - Sometimes increasing the number of frames increases the amount of page faults

8.4.3 optimal page replacement
- Replace the page that will not be used for a longest period of time
- Used to measure how well your algorithm performs

8.4.4 Least Recently Used page replacement
- Use past knowledge rather than future
- Replace page that has not be been used in the most amount of time
- Associate time of last use with each page
- 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
    - When page is referenced
      - Move it to the top
      - Requires 6 pointers to be changed
    - But each update more expensive
    - No search for replacement
  - No belady's anomaly

8.4.5 LRU approximation page replacement
- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0
    - Dont know order

8.4.5.2 second chance algorithm
- Generally FIFO, plus hardware provided reference bit
- Clock replacement
- If page is selected
  - Check reference bit
    - If 0 replace
    - If 1 leave page in memory, replace next page
  - When given second chance reference bit cleared

8.4.5.3 enhanced second-chance algorithm
- Improve algorithm by using reference bit and modify bit in concert
- Take ordered pair (reference, modify)
  - (0,0) neither recently used or modified first choice to replace
  - (0,1) not recently used but modified - second best choice to replace
  - (1,0) recently used but clean - third best choice to replace
  - (1,1) recently used and modified worst choice

8.4.6 counting based page replacement
- Keep a counter of the number of references that have been made to each page
  - Not common
- Least frequently used algorithm
  - Replace page with smallest count
- Most frequently used algorithm
  - Based on argument that page with the smallest count was probably just brought in and has yet to be used

8.5 Allocation of frames
- Each process needs a minimum number of frames
- Maximum is the total frames in the system
- Two major allocation schemes
  - Fixed allocation
  - Priority allocation
- Many variations

8.5.2 allocation algorithms
- Fixed allocation
  - Equal allocation
    - Each process is given same amount of frames
    - Some are kept as free frame buffer pool
- Proportional allocation
  - Allocate according to size of process
    - Dynamic as degree of multiprogramming, process sizes change
- Allocation for process = size of process / (summation of both process sizes) X number of frames
- Priority allocation
  - Use a proportional allocation scheme using priorities rather than size
  - If process generates a page fault
    - Select for replacement one of its frames
    - Select for replacement a frame from a process with lower priority number

8.5.3 global versus local allocation
- Global replacement
  - Process selects a replacement frame from the set of all frames, one process can take a frame from another
- Local replacement
  - Each process selects from only its own set of allocated frames
    - More consistent per-process performances
    - But possibly underutilized memory

8.5.4 non-uniform memory access
- So far all memory accesses equally
- Many systems are NUMA
  - Non uniform memory access
    - Speed of access to memory varies
    - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performances comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by solaris by creating igroups
    - Structure track CPU / memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the igroup

8.6 thrashing

- If a process does not have enough pages, the page fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - Leads to
    - Low CPU utilization
    - Operating system thinking that it needs to increases the degree of multi programming
    - Another process added to system
- Thrashing
  - Process is busy swapping pages in and out
  - Caused by
    - One or multiple processes spending more time paging in and out than executing, and a scheduler that is set to increase the degree of multiprogramming when CPU utilization is low
- Demand paging and thrashing
  - Why does demand paging work
    - Locality model
      - Process migrates from one locality to another
      - Localities may overlap

8.6.2 working set model

- Locality is a set of pages that are frequently used together
- Delta = working-set window = a fixed number of page references
- WSS_i (working set of process P_i)=
  - Total number of pages referenced in the most recent delta (varies in time)
    - If delta too small will not encompass entire locality
    - If delta too large will encompass several localities
    - If delta infinite will encompass entire program
- D = summation of WSS_i = total demand frames
  - Approximation of locality
- If D (the locality) is greater than m (the number of available frames) thrashing occurs
- Policy if D > m suspend process

8.6.2.2 keeping track of the working set

- Approximate with interval timer + a reference bit
  - Delta = 10,000
    - Timer interrupts after every 5000 time units
    - Keep in memory 2 bits for each page
    - Whenever a timer interrupts copy and sets the values of all reference bits to 0
    - If one of the bits in memory = 1
      - Page in working set
- Why is this not completely accurate
- Improvement = 10 bits and interrupt every 100 units

8.6.3 page fault frequency

- More direct approach than WSS
- Establish “acceptable” page-fault frequency (PFF) rate and use the local replacement policy
  - If actual rate too, low process loses frame
  - If actual rate too high, process gains frame

8.6.4 working sets and page fault rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes overtime
- Peaks and valleys over time

8.8 allocating kernel memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - Device i/o

8.8.1 buddy system

- Allocates memory from fixed size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- For example assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each
        - One used to satisfy request
- Advantage
  - Quickly coalesce unused chunks into larger chunk
- Disadvantage
  - Fragmentation

8.8.2 slab allocation

- Alternate strategy
- Slab is one or more physically contiguous pages
- Cache consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache created, filled with objects
    - Instantiations of data structures
- When cache created, filled with objects marked as free
- When structure stored, objects marked as used
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slab, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

8.8.2.1 slab allocator in linux

- For example process descriptor is of type struct task_struct
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free struct task_struct
- Slab can be in three possible states
  - Full
    - All used
  - Empty
    - All free
  - Partial
    - Mix of free and used
- Upon request, slab allocator
  - Uses free struct in partial slab
  - If none, takes one from empty slab
  - If no empty slab, create new empty
- Slab started in solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 has slab no has both slob and slub allocators
  - Slob for systems with limited memory
    - Simple list of blocks
      - Maintains 3 list objects for small, medium, large objects
  - Slub is performance-optimized slab removes per-CPU queues, metadata stored in page structure

8.9 other considerations

- Cool stuff

8.9.1 prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- If prepaged pages unused i/o and memory was wasted
- Assume s pages are prepaged and a of the pages is used
  - Question becomes is whether the cost of s * a is greater than the or less than the cost of prepaging s * (1-a) unnecessary pages
  - As a approaches 0 prepaging loses if a is close to 1 prepaging wins

8.9.2 page size

- Sometimes OS designers have a choice
  - Especially if running on custom built CPU
- Page size selection must take into consideration
  - Fragmentation
  - Page table size
  - Resolution
  - i/o overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range of 2^{12} (4096 bytes) to 2^{22} (4,194,302 bytes)
- On average growing over time

8.9.3 TLB reach

- Transition look aside buffer reach
  - The amount of memory accessible from the TLB
- TLB reach
  - (TLB size) x (Page size)
- Ideally the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the page size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide multiple page sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increases in fragmentation

8.9.6 I/o interlock

- i/o interlock
  - Pages must sometimes be locked into memory
- Consider i/o
  - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- Pinning of pages to lock into memory

8,10 .1 windows

- Uses demand paging with clustering
  - Clustering brings in pages surrounding the faulting page
- Processes are assigned working set minimum and working set maximum
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

8.10.2 solaris

- Maintains a list of free pages to assign faulting processes
- Lotsfree
  - Threshold parameter (amount of free memory) to begin paging
- Desfree
  - Threshold parameter to increasing paging
- Minfree
  - Threshold parameter to being swapped
- Paging is performed by pageout process
- Pageout
  - Scans using modified clock algorithm
- Scanrate
  - The rate at which pages are scanned
    - Ranges from slowscan to fastscan
- Pageout
  - Called more frewuently depending upon the amount of free memory available
- Priority paging gives priority to process code pages

Chapter 10 file-system interface

10.1 file concept
- Contiguous logic address space
- Types:
  - Data
    - Numeric
    - Character
    - Binary
  - Program
- Contents defined by file's creator
  - Many types
    - Consider text file, source file, executable file

10.1.1 file attributes
- Names
  - Only information kept in human readable form
- Identifier
  - Unique tag (number) identifies file within file system
- Type
  - Needed for systems that support different types
- Location
  - Pointer to file location on device
- Size
  - Current file size
- Protection
  - Controls who can do reading writing executing
- Time, date, and user identification
  - Data for protection, security, and usage monitoring
- Information about files are kept in the directory structure which is maintained on the disk

10.1.2 file operations
- File is an abstract data type
- Create
- Write
  - At write pointer location
- Read
  - At read pointer location
- Reposition within file
  - Seek
- Delete
- Truncate
- Open(F_k)
  - Search the directory structure on disk for entry F_k, and move the content of entry to memory
- Close(Fk)
  - Move the content of entry F_k in memory to directory structure on disk

10.1.2.1 open files
- Several pieces of data are needed to manage open files:
  - Open-file table: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counter of number of times a file is open
    - To allow removal of data from open file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

10.1.2.2 open file locking
- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - Shared lock similar to reader lock
    - Several processes can acquire concurrently
  - Exclusive lock similar to writer lock
- Mediates access to a file
- Mandatory or advisory
  - Mandatory
    - Access is denied depending on locks held requested
  - Advisory
    - Processes can find status of locks and decide what to do

10.1.4 file structure
- None
  - Sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides
  - Operating system
  - Program

10.2 access methods

10.2.1 Sequential access
- Information in file is processed in order on record after another
- Read and writes make up the bulk of operations
- Read_next
  - Reads the next portion of the file and automatically advances a file pointer, which tracks the i/o location
- Write_next
  - Appends to the end of the file and advances to the end of the newly written material

10.2.2 direct access
- File is made up of fixed length logical records
  - Allows programs to read and write records in no particular order
- File is viewed as a numbered sequence of blocks or records
- Read(n)
  - N is the block number
  - Reads block at block number
- Write(n)
  - N is block number
  - Writes at block number
- Position_file(n)
  - Positions file to block number
  - Read_next() reads the block number
  - Write_next() writes the block number
- Relative block number allows OS to decide where file should be placed

10.2.3 other access methods
- Can be built on top of base methods
- General involve creation of an index for the file
- Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about that item)
- If too large, index (in memory) of the index (on disk)
- IBM indexed sequential-access method (ISAM)
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS
- VMS operating system provides index and relative files as another example (see next slide)

10.3.0.1 directory structure
- A collection of nodes containing information about all files
- Both the directory structure and the files reside on disk

10.3.0.2 disk structure
- Disk can be subdivided into partitions
- Disks or partitions can be RAID protected against failure
- Disk or partitions can be used raw
  - Without a file system, or formatted with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a volume
- Each volume containing file system also tracks that file system's infor in device directory or volume table of contents
- As well as general-purpose file systems there are many special purpose file systems, frequently all within the same operating system or computer

10.3.0.3 types of file systems

- We mostly talk of general-purpose file systems
- But systems frequently have many file systems, some general- and some speical purpose
- Consider solaris has
  - Tmpfs
    - Memory based volatile FS for fast, temporary i/o
  - Objfs
    - Interface into kernel memory to get kernel symbols for debugging
  - Ctfs
    - Contract file system for managing daemons
  - Lofs
    - Loopback file system allows one FS to be accessed in place of another
  - Procfs
    - Kernel interface to process structures
  - Ufsm zfs
    - General purpose file systems

10.3.2 directory overview

- Operations performed on directory
  - Search for a file
    - Search directory structure to find entry for particular file
  - Create file
    - New files need to be created and added to directory
  - Delete a file
    - When file no longer needed, be able to remove it from directory
  - List a directory
    - Able to list the files in a directory and the contents of the directory entry for each file in the list
  - Rename a file
    - Be able to change the name of a file, allows position within directory to be changed
  - Traverse the file system
    - Access every directory and every file within a directory structure

10.3.2.1 directory organization

- Directory organized to obtain
  - Efficiency
    - Locating a file quickly
  - Naming
    - Convenient to users
      - Two users can have same name for different files
      - The same file can have several different names
  - Grouping
    - Logical grouping of files by properties

10.3.3 single level directory

- A single directory for all users
  - All files contained in same directory
- Naming problem
  - Each file must be given unique name, multiple users may want to name file the same way
- Grouping problem

10.3.4 two-level directory
- Separate directory for each user
  - Path name
    - The way the system would get to file starting from root
  - Can have the same file name for different user
  - Efficient searching
  - No grouping capability

10.3.5 tree structured directories
- Efficient searching
- Grouping capability
- Current directory (working directory)
  - Cd /spell/mail/prog
  - Type list
- Absolute or relative path name
  - Absolute, from root
  - Relative, from current node
- Creating a new file is done in current directory
- Delete a file
  - Rm <file-name>
- Creating a new subdirectory is done in current directory
  - Mkdir <dir-name>
- Deleting deletes all descendants as well

10.3.6 acyclic-graph directory
- Have shared subdirectories and files
- Two different names (aliasing)
- If dict deletes list
  - Dangling pointer
  - Solutions
    - Backpointers, so we can delete all pointers variable size records a problem
    - Backpointers using a daisy chain organization
    - Entry-hold-count solution
- New directory entry type
  - Link
    - Another name(pointer) to an existing file
  - Resolve the link
    - Follow pointer to locate the file

10.3.7 general graph directory
- How do we guarantee no cycles
  - Allow only links to file not subdirectories
  - Garbage collection
    - Check and remove empty links
  - Every time a new link is added use a cycle detection algorithm to determine whether it is ok

10.4 file system mounting
- A file system must be mounted before it can be accessed
- A unmounted file system is mounted at a mount point
  - Mount point
    - The location within the file structure where the file system is to be attached

10.5 file sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a protection scheme
- On distributed systems, files may be shared across a network
- Network file system (NFS) is a common distributed file-sharing method
- If multi-user system
  - User IDs
    - Identify users, allowing permissions and protections to be per-user
  - Group IDs
    - Allow users to be in groups, permitting group access rights
  - Owner of a file / directory
  - Group of a file / directory

10.5.2 remote file systems

- Uses networking to allow file systems access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using distributed file systems
  - Semi automatically via the world wide web
- Client-server model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Clients and user-on-client identification is insecure or complicated
  - NFS is standard UNIX client server file sharing protocol
  - CIFS is standard windows protocol
  - Standard operating system file calls are translated into remote calls
- Distributed information systems (distributed naming services) such as LDAP, DNS, NIS, Active Directory implement unified access to information for remote computing

10.5.2.3 failure modes

- All file systems have failure modes
  - For example corruption of directory structures or other non-user data, called metadata
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS v3 include all information in each request, allowing easy recovery but less security

10.6 protection

- File owner/creator should be able to control:
  - What can be done
  - By whom
- Types of access
  - Read
    - Can read from file
  - Write
    - Can write or rewrite file
  - Execute
    - Load file into memory and execute
  - Append
    - Write new information at end of the file
  - Delete
    - Delete the file and free its space for possible reuse
  - List
    - List the name and attributes of the file

10.6.2 access control

- Mode of access:
  - Read, write, execute
- Three classes of users on Unix/ linux
  - Owner access
    - Owner who created the file
  - Group access
    - Set of users who are sharing the file and need similar access
  - Public access
    - All users in the system
- Ask manager to create a group (unique name), say g, and add some user to the group
- For a particular file (say game) or subdirectory define an appropriate access
- Attach a group to a file

11.1 file-system structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located and retrieved easily
- Disk provides in-place rewrite and random access
  - i/o transfers performed in blocks of sectors (usually 512 bytes)
- File control block
  - Storage structure consisting of information about a file
- Device driver controls the physical device
- File system organized into layers

11.1.1 layered file system

- Application programs -> logical file system -> file-organization module -> basic file system -> -> i/o control -> devices
- Device drivers
  - manage i/o devices at the i/o control layer
    - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” Outputs low-level hardware specific commands to hardware controller
- Basic file system
  - given command like “retrieve block 123”
    - Translates to device driver
  - Also manages memory buffers and caches (allocation, freeing, replacement)
    - Buffers hold data in transit
    - Caches hold frequently used data
- File organization module
  - Understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation
- Logical file system
  - Manages metadata information
    - Translates file name into file number, file handle, location by maintaing control blocks (inodes in UNIX)
    - Directory management
    - Protection
- Layering useful for reducing complexity and redundancy but adds overhead and can decrease performance.
  - Logical layers can be implements by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
  - Each with its own format
    - CD-ROM is ISO 9660
    - UNIX has UFS,FFS
    - Windows has FAT, FAT32, NTFS
    - Linux has 40 types, with extended file system ext2 and ext3 leading

11.2 file-system implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- Boot control block
  - Contains info needed by system to boot OS from that volume
- Volume control block (super block, master file table)
  - Contains volume details
    - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table
- Per-file file control block (FCB) contains many details about the file
  - Inode number, permissions, size, dates
  - NFTS stored into in master file table using relational DB structures

11.2.1 in-memory file system structures

- Mount table storing file system mounts, mount points, file system types
- The following figure illustrates the necessary file system structures provided by the operating system
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

11.2.2 partitions and mounting

- Partition can be a volume containing a file system (“cooked”) or raw
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or boot management program for multi-os booting
- Root partition
  - Contains the OS, other partitions can hold other OSes, other file systems, or be raw
    - Mounted at boot time
    - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct
    - If not, fix it, try again
    - If yes, add to mount table, allow access

11.2.3 virtual file systems

- Virtual file systems (VFS)
  - On Unix provide an object oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be on of many file systems types, or network file system
    - Implements vnodes which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementations routines
- The API is to the VFS interface, rather than any specific type of file system
- For example, linux has four object types
  - Inode, file, superblock, dentry
- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table
    - Function table has addresses of routines to implement that function on that object

11.3 directory implementation

- Linear list of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- Hash table
  - Linear list with hash data structure
    - Decreases directory search time
    - Collisions
      - Situations where two file names hash to same location
    - Only good if entries are fixed size, or used chained-overflow method

11.4 allocation methods

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
    - Each file occupies set of contiguous blocks
      - Best performance in most cases
      - Simple
        - Only starting location (block#) and length (number of blocks) are required
      - Problems include finding space for file, knowing file size, external fragmentation, need for compaction offline (downtime) or on-line
    - Mapping from logical to physical
      - Block to be accessed = Q + starting address
      - Displacement into block = R

11.4.1 extent-based systems

- Many newer file systems (VFS) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- Extent
  - A contiguous block of disks
    - Extent are allocated for file allocation
    - A file consists of one or more extents

11.4.2 linked allocation

- Linked allocation
  - Each file a liked list of blocks
    - File ends at nil pointer
    - No external fragmentation
    - Each block contains pointer to next block
    - No compaction, external fragmentation
    - Free space management system called when new block needed
    - Improve efficiency by clustering blocks into groups but increases internal fragmentation
    - Reliability can be a problem
    - Locating a block can take many i/os and disk seeks
  - FAT (file allocation table) variation
    - Beginning of volume has table, indexed by block number
    - Much like a linked list, but faster on disk and cacheable
    - New block allocation simple
  - Block to be accessed in the Qth block in the linked chain of blocks representing the file
    - Displacement into block = R + 1

11.4.3 indexed allocation

- Indexed allocation
  - Each file has its own index block(s) of pointers to its data blocks
- Logical view
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes.
  We need only 1 block for index table
  - Q = displacement into index table
  - R = displacement into block
- Mapping from logical to physical in a file of unbounded length(block size of 512 words)
- Linked scheme
  - Link blocks of index table (no limit on size)
    - Q_1 = block of index table
    - R_1 is used as follows
      - R_1 / 512 leads to Q_2 and R_2
    - Q_2 = displacement into block of index table
    - R_2 displacement into block of file
  - Two level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)
  - Q_1 = displacement into outer - index
  - R_1 is used as follows
    - R_1/ 512 leads to Q_2 and R_2
  - Q_2 = displacement into block of index table
  - R_2 displacement into block of file
- Combined scheme: UNIX UFS
  - 4 bytes per block, 32-bit addresses
  - More index blocks than can be addressed with 32-bit pointer

11.4.4 performance

- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead
- Adding instructions to the execution path to save one disk i/o is reasonable
  - Typical disk drive at 250 i/o per second
  - Fast SSD drives provide 60,000 IOPS