RAID(Redundant Array of Inexpensive Disks)

- group of disks/memory all looking like one disk to client
- advantage of performance using parallelism
- consists of a microcontroller that runs firmware, volatile memory such as DRAM to buffer data blocks, and non-volatile memory to buffer writes safely and specialized logic to perform parity calculations

RAID Evaluation Criteria

- capacity => given a set of N disks with B blocks, how much is actually available to a client
- reliability => how many disk faults can it tolerate
- performance => depends on workload presented in the disk array
    - transfer throughput = Amount of Data/(total time to access)
    - steady state throughput = N(number of disks) * S(sequential bandwidth)

RAID Level 0(striping) => performance

- stripe the blocks of the array across the disks in a round robin fashion
- takes advantage of parallelism since contiguous requests to memory can be serviced on multiple disks at once

RAID Level 1(mirroring) => random I/O performance and reliability

- mirrors the data of another block on a consecutive block => example: disk0 and 1 have identical contents
- performance
    - read latency is same as on a single disk => RAID 1 just redirects the read to one of its copies
    - however, sequential read delivers half the bandwidth since each disk receives a request for every other block so when rotating over a skipped block, there isn't useful bandwidth => therefore the steady state throughput/bandwidth is still N/2 *s
    - write latency increases because two physical writes occur => maximum of the worst case seek/rotational delay
- write ahead log
    - prevents a partial write that updates only one disk and causes an inconsistency
    - record what the RAID is about to do in a journal and replay the procedure if a crash occurs
    - this journal is put into non-volatile RAM with a special battery
-

RAID Levels ⅘(parity-based redundancy)

- RAID Level 4
    - adds redundancy using parity
    - XORS all the bits and stores it in a parity block that is used to recover information if columns of data are lost
    - full stripe writes are the most efficient way
    - performance
        - sequential read utilize all the disks except parity disk for (N-1)*S
        - using full-stripe writes sequential writes can also achieve (N-1)*S
        - random 1-block reads are spread across all the disk for (N-1)*R
        - random writes
            - additive parity => compute value of the new parity block by reading in all other data blocks in the stripe and XOR with block 1
            - subtractive parity(better) => read old data at block we are writing and old parity => compare old data with new data and either flip or stay the same
            - bottlenecks at the parity disk => performs two I/Os per logical I/O to achieve R/2 on the parity disk which means that we have to wait for 2 I/Os on that disk to finish
- RAID level 5(capacity and reliability)
    - identical to RAID-4 except that it rotates the parity block across drives
    - This means that the parity blocks are separated so if we have to make a random write, it separates the two I/Os required to different disks resulting in a performance of N/4*R where the ¼ is because of the 4 I/Os we need to do

SSD Flash Storage

- flash chips are organized into banks or planes
- each bank is separated into blocks and pages (128 KB and a few KB in size)
- in order to write to a page within a block, you first have to erase the entire block
- utilize multiple flash chips in parallel
- write amplification => defined as total write traffic issued to the flash chips divided by the total write traffic issued by the client to the SSD
- records all translations of logical block to physical page in in-memory mapping table
- this table is recorded in an out-of-band area => when device loses power, it reconstructs its mapping table by scanning the OOB areas and reconstructing the map in memory => higher end devices use more complex logging and checkpointing techniques to speed up recovery
- uses trimming to inform OS that a block is no longer needed
- block-based mapping
    - to reduce costs of mapping
    - reduce the amount of mapping information
- hybrid mapping
    - FTL keeps a few blocks erased and directs all writes to them called log blocks
    - keeps per-page mappings for these log blocks(log table) and a larger perblock mapping in the data table
    - FTL will consult log table then data table
    - switch merge if an entire block is overwritten, write the new data to a new block and change the block pointer to point to this new block
    - partial merge, if only a part of the block is overwritten => read the data not used and append it to the log of the new data
    - FTL must periodically read all live data out of such blocks and rewrite it elsewhere to making wear leveling

---

DeadLock

- hand over hand locking => one lock for each node of linked list and grab one and release previous as you go down list
- Conditions
    - mutual exclusion
        - resource in question can only be used by one entity at a time
        - Solution:
    - incremental allocation
        - allocation when they truly need it resulting in different allocation times
    - no preemption
        - can't remove a resource from someone who already has it
    - circular waiting(hold and wait)
        - wait in a circular loop
        - Solution: test another lock, if i can't acquire it drop my current lock)
- java has intrinsic lock as part of each object that is required to use that object
    - monitor object is a collection of condition variables and procedures combined together in a special kind of module or package => usually focused on a specific class instance
    - processes running outside the monitor can call procedures of the monitor and only one process at a time can call code inside monitors
    - implemented in semaphore fashion and has process queue
- Solutions
    - reservations => advanced reservations for resources => resource manager tracks outstanding reservation
        - applications still deal with reservation failures
    - resource leases with time outs or revocation under the OS
        - revoke access by invalidating resource handle
        - cannot revoke if object is part of the process's address space

- all resources allocate resources in the same order
    - order by resources(groups before members)
    - order by relationship(parents before children)
- allow deadlocks to occur and detect them then break the deadlock
    - uses health monitoring that monitors the time a response takes/ submit test transactions
- each monitor object has a semaphore that is automatically acquired on any method invocation
- Deadlock avoidance
    - just refuse to grant requests that would put the system in a resource depleted state/deadlock
    - difficult to reject a process gracefully so usually implemented by requiring reservations of resources
    -

- Examples
    - Java has synchronized methods where developer can identify serialized methods to allow for more fine grained granularity
- health monitoring
    - ways to do this
        - internal monitoring agent watches message traffic or a transaction log to make sure work is continuing
        - by asking clients to submit failure reports to a central monitoring service when a server appears to become unresponsive
        - each server sends periodic heart-beat messages to the monitoring service
        - periodic test requests to service being monitored
    - managed recovery
        - any process should have the ability to be killed/restarted at any time with minimal disruption to normal workflow
        - software should be designed to support multiple levels of restart
            - warm-start => restore last saved state
            - cold-start ignore saved state
            - restart single processes vs all processes involved in that service
            - restarts escalate if the current solution isn't working
        - other examples
            - non-disruptive rolling upgrades: if a system is capable of operating without some of its nodes, we can non-disruptively upgrade some software on these nodes making sure we have an auto fall-back option in case the upgrade fails
            - prophylactic reboots => just auto restart a system at regular intervals to solve some problems

Event Based Concurrency

- single event loop that does not do any blocking unless completely necessary(eg page faulting)
- revolves around the AIO control block that makes asynchronous I/Os to the file
    - to find out whether or not the given action has completed, we either poll or get an interrupt

Performance Measurements

- Difficulties
    - components operate on a complex system with many moving parts, ongoing competition for resources, and lack of clear/rigorous requirements
- Time-stamped event logs
    - app instrumentation technique
    - create log buffer and routine for all interesting events
    - extract buffer, archive data
- end-to-end testing
    - client-side throughput latency measurements
    - very accurate and easy to run
    - can't actually debuf if undesirable outcome is found including no diagnostics
- Performance Analysis

- metrics
    - duration/response time
    - processing rate
    - resource consumption
    - reliability => ex: msgs delivered without error
    - indices of dispersion(mean, median, mode)
- metric characteristics
    - completeness
    - redundancy
    - variability
    - feasibility
- non-metrics
    - Job completion
    - Trust

- Design for Performance
    - establish performance requirements early
    - anticipate bottlenecks including limited resources, frequent operations, traffic concentration
    - use levels of performance testing such as categorical(types of file systems) or load
- WorkLoads
    - artificial load generation can be controlled and scaled but can lead to unrealistic loads
    - replayed workloads can represent real usage scenarios but are very limited(traces)
    - live loads are realistic but can't be scaled on demand
    - standard workloads allow comparisons and are well-maintained but are often used in a limited set so aren't applicable
- Things to watch out for
    - cache effects like warming up the cache(start up costs can distort total cost_
    - system performance degrades with age
    - logging may dwarf the costs of instrumentation and distort the resolts
        - use execution profiling which are automated measurement tools
    - Measuring latency without considering utilization of system => not using a characteristic background load
    - Not reporting variability of measurements
- Example
    - conquest file system using persistent RAM to store many files
        - metrics
            - throughput measurements on read, write, creates, and bandwidth
        - factors
            - categories of file system and scaling effects(size of system)
        - workload
            - used standard benchmarks

Devices/Device Drivers and I/O

- each device has its own piece of code called the device driver
- highly specific to the particular device and inherently modular
- OS defines classes of devices and define a specific interface/behavior
- common functionality belongs in the OS including caching, file system code, and network protocols
    - uses Device Driver Interface including basic entry points that correspond directly to system calls
    - Device Kernel Interface specifies services OS provides to the drivers => ABI for device driver writers
    - OS depends on driver implementations of DDI and drivers depend on kernel DKI implementations
    - DDI's are easy to use since most of the high level code is already written, however, DDI's make implementing additional functionality harder since software will not recognize it

- specialized functions belong to the drivers
    - primarily interrupt driven => slower than CPUs and use interrupts to get CPU attention
        - sometimes combination of both, first poll OS but interrupt after certain timeout frame
    - devices and CPU are both connected to a bus and devices communicate with CPU across the bus
    - CPU can enable and disable interrupts from devices
- Device utilization
    - utilize parallelism to maximize throughput
        - DMA allows any two devices attached to the memory bus to move data directly without passing through the CPU first
        - if the bus is servicing DMA requests, the CPU is not making requests to memory => CPU mainly works with the registers
    - queue up devices like process and requesters block to await completion
        - use DMA to do memory/data transfers
            - scatter/gather I/O
                - device controllers support DMA transfers that must be entirely contiguous
                - scatter => read from device to multiple pages
                - gather => write multiple pages to devices
                - pages can be scattered throughout physical memory but are contiguous chunks in the user's virtual memory space in a user I/O buffer and on the device
        - Sometimes DMA is not required b/c it's only designed for large contiguous transfers
            - device register/memory can be treated as part of the regular memory space
            - **DMA vs Memory Mapping**
                - DMA performs large transfers efficiently and better utilizes device and CPU and doesn't have to wait for the CPU; however, has large setup and processing overhead
                - memory mapped I/O has no per operation overhead
        - when request completes, controller generates an interrupt and the CPU calls the appropriate handler
        - use double buffered outputs
            - the deeper the request queues, the less idle time there is/possibility of combining adjacent requests
            - double buffered output has multiple reads queued up and ready to go so that as soon as one buffer is finished, another one is immediately ready
        - use double buffered outputs
            - multiple reads queued up and ready to go => as soon as one read completes, another read starts
            - this way application doesn't need to wait for data to be read
            - we can queue up reads from multiple processes to make sure a single process doesn't block
    - Device classes
        - block devices => block addressable random access storage
            - support queued asynchronous reads and writes
            - uses complex services like buffer allocation, LRU management of cache, data copying services, scheduled I/O and asynchronous completion
        - devices are differentiated using major device number

File Systems

- organize data into coherent units and store each unit in a file
- disk provided cheap persistent storage at the cost of high latency in the form of seeks and rotations
- deal with two kinds of information
    - data => information that the file is supposed to store
    - metadata => information about the file
- file system API
    - file container operations

        - standard file management system calls manipulate files as objects and ignore the contents of the file
        - manages ownership protection, links, etc
    - directory operations
        - provides organization of a file system
    - file I/O operations
- file system cache takes advantage of locality for reading
    - file systems divided into fixed sized blocks that are split into superblock/inode and data bitmaps, actual inodes that store meta-data (block-sized allocation is very important)
    - DOS file system divides space into clusters with a file allocation table that contains one entry per cluster => each entry either contains the next cluster in the file, end of the file, or free cluster
    - File index blocks where there is a file control block that points to all blocks in a file
        - some of these index block points to other blocks
- when file is opened, an in-memory structure is created that points to RAM pages that indicates whether pages are dirty etc.
    - two different structures => one shared by all processes and one shared by cooperating processes
    - two processes can share one descriptor and two descriptors can point to the same inode
- File Systems Layer
        - support multiple different file systems
        - implemented on top of block I/O and device independent
        - same basic functions (e.g. create/destroy files, get/set attributes)
- why multiple file systems
        - multiple storage devices
        - different services
        - different purposes
- relationship with block I/O devices
        - file systems sit on top of general block I/O layer
        - make all disks look same
        - standard operations implemented on block devices
        - map logical block numbers to device addresses
        - encapsulate specifics of device support
- why device independent block I/O
        - better abstraction than generic disks
        - unified LRU buffer cache for data
        - buffers for data re-blocking
        - automatic buffer management

FUSE(File System in User Space)

- software interface for unix like computer operating systems that let non-privileged users create their own file systems without editing kernel code
- to implement a new file system, a handler program linked to the supplied libfuse library needs to be written. the main purpose of this program is to specify how the file system responds to read/write/state requests
- program is also used to mount the file system where the handler is registered with the kernel
- if a user issues read/write/stat requests, kernel forwards these requests to the handler

File System Allocation

- Creating a new file
    - allocate a free file control block
        - UNIX: Search the super block free inode list and take the first free inode
        - DOS: Search the parent directory for an unused directory entry
    - initialize the new file control block and give the file a name
    - best to make large transfer units that allocate space in large chunks
        - unfortunately causes fragmentation which can be fixed by periodic defragmentation routines

        - **large-file allocation:** large files are stored in separate blocks of different tracksso that it doesn't take up all of one track
        - **parametrization:** solves the problem of sequential reads not actually fully taking advantage of locality since reads are longer than rotations across that data segment
- Appending to a file
    - find a free chunk from the free list -> find the file descriptor and update the control block
    - bundles small writes into consecutive larger writes that are stored in write-back cache
        - if the data is subsequently deleted, might prevent disk from actually being written
    - free list ordering by memory address => better coalescing
        - by size => better searches for best fit sized chunks for example using binary search
- Deleting a file
    - release all the space that is allocated to that file
        - UNIX: return block to free block list
        - DOS: garbage collection by doing periodic deallocations
    - deallocate file control block
        - UNIX: zero the inode and return it to the free list
        - DOS: zero the first byte of the name in the parent directory indicating the directory is no longer in use
    - LFS
        - periodically read in old segments and determines the liveness of these blocks
        - determining liveness
            - look in segment summary block => then find corresponding block using imap and check the offset, if the block is located there, gg
            - find its inode summary and offset in segment summary block
            - look in imap to find where inode number is and read it from disk => use offset to look in the inode to see where the inode thinks that block is => if there is a inconsistency, the block is dead
        - clean rarely used segments rather than frequently changing segments
        - keep track of two checkpoint regions at either end of a segment where the segment points to the next segment
        - start at the last known checkpoint region and read in all updates then complete these updates
- Disk Caching
    - general block caching
        - popular files are read frequently
        - files that are written and then promptly reread
        - provides buffers for read ahead and deferred writes
    - special purpose caches
        - same directory
        - inode caches to speed up re-uses of same file
- naming/binding
    - file system uses inode numbers but those aren't readable to people or programs
    - idea of namespace which is the total collection of names known by some naing mechanism
    - file systems typically organized by hierarchical directory name spaces=> file names are interpreted relative to that directory
    - UNIX allows multiple names using hard links where the actual file descriptors are inodes and the directory entries only point to these inodes
        - contents of a file contains its name and a pointer to the inode of the associated file
        - file exists as long as one of these names exist because there is a reference list in the file inode
        - symbolic links on the other hand only contain the address and do not reference the inode => do not prevent deletion, do not guarantee the validity of the path
- crashes/consistency problems
    - NVRAM disk controllers, un-interruptable power supply, super caps and fast flush

- Solutions
    - ordered writes
        - writing data before pointers to it
        - write out deallocations before allocations
        - Problems
            - reduced I/O performance
            - eliminates write buffering and shortest position time first scheduling
            - modern disk drives sometimes reorder queued requests
    - Audit and Repair
        - audit file system and maintain redundant information
        - verifying file system checksums, reference counts etc
        - first check reasonability of superblock
        - then scan inodes, indirect blocks, etc to see which blocks are currently allocated. From there try to build bitmap and see if it makes sense. and vice versa
        - check inodes for corruption (e.g. valid type field)
        - verify inode links. Fix is usually to correct the link count if any problem is found
        - check for corrupted, duplicate pointers
        - check for bad blocks (i.e. if it points to something out of range). Clear the pointer
        - ensure no cycles in directory graph
        -
    - Journaling
        - fast because of sequential writes
        - guaranteed 512 byte atomic writes
        - small compared to file system so fast scans and updates
        - contains
            - transaction metadata, data
            - commit block
            - checkpoint region
        - also writes revoke blocks so that remnant info can be erased before it is written
        - Data+ metadata journaling
            - always sequential and journal completes when real writes happen
            - accumulate batch of journal entries until it is full
            - when system is restarted, review the journal and check which operations haven't been completed
        - meta-data only journaling
            - small and random(I/O inefficient)
            - integrity critical (if meta-data is corrupted, huge potential for data loss)
            - journal data isn't a high priority since it is less order sensitive, basically just write the data first
            - basically just write the data out then journal the meta data to inform that we have finished

OS security

- security is a policy and protection is a mechanism
- authentication determining which party is requesting vs authorization => determining if the party is eligible
- identification on computers
    - primarily rely on user ID and password by storing hash of password
- access control in OS
    - access control lists which maintains a single list for each protected object that is maintained by the OS
        - UNIX and DOS uses rwx read write execute for owner group and everyone else
        - pros/cons

            - easy to figure out who can access the resource but hard to find out what a subject can actually access
        - capabilities
            - addition to every processes PCB block that is a string of bits indicating the objects it can access
            - pros/cons
                - easy to determine which objects a subject can access
                - hard to determine who can access a particular object
                - extra mechanism for revocation
        - UNIX uses both by using ACLS for file open but then adds capabilities to that file descriptor
- cryptography
    - cryptographic algorithms have a key to perform encryption and decryption
    - keys stored in key pairs(public and private key)
    - public key is made known to the world so that when one wants to send message to user, encrypt with his/her public key that will be decrypted by his/her private key
    - private key is only known to owner
    - asymmetric cryptography used to bootstrap symmetric crypto
    - RSA to authenticate/establish a session key
    - password salting => to defend from brute force dictionary attacks (since attackers know the encryption algorithm) they could hash every single combination in a dictionary
        - basically add a random string or salt to the end of a password or other string that requires encrypting and hash the resulting string.
        - this way an attacker would have to generate every dictionary word + every combination that results in a salts

third party signing of a packet with the owner's public key and signature that signs it => hashed and sent to user

Distributed Systems

Goals
- scalability and performance
- reliability and availability
- ease of use
- enable new collaboration

Deutsch's Seven Fallacies
- network is reliable
- no latency
- available bandwidth is infinite
- network is secure
- topology of network doesn't change
- one administrator
- cost of transporting additional data is zero

Systems
- loosely coupled systems: parallel group of independent computers connected by high speed LAN that requires minimal communication
    - scales well, ease of management, reconfigurable capacity(horizontal scalability)
        - service availability is good and inexpensive
        - challenges with automated updating, self monitoring etc.
    - examples: web servers, app servers, cloud computing
- Cloud Computing
    - large number of machines all identically configured
    - supports special kinds of parallel distributed processing

Building One

- RPC(Remote Procedure Calls)
    - procedure calls are a fundamental paradigm
    - implemented with interface specifications on both sides
    - machine independent data type representations
    - client stub => client proxy for methods in the API
    - server stub => server side recipient of API invocations
- Cons of RPC
    - limited failure handling

Distributed System Synchronization

- No shared memory for atomic instruction locks
- can't order spatially separate events b/c they aren't correlated
- Use Lease
    - obtained from a central resource manager
    - gives client exclusive right to update the file
    - valid for a limited time
    - objects that are in the middle of an updating process are restored to last good state prior to lease implementing all-or-nothing transactions
- Distributed Consensus
    - achieving simultaneous unanimous agreement
    - typical algorithm
        - each interested member broadcasts his nomination
        - parties evaluate the received proposals
        - after reasonable time for proposal, each voter votes

Remote Data Access

- complete transparency in the form of making it appear as if just querying from one large system
- client side file system is a proxy that translates file operations into network requests
- server side daemon receives requests and processes them
- many distributed file systems use whole file caching
    - higher network latency justifies pulling entire files
    - AFS caches server files on the client to reduce the need to make remote requests
    - AFS implements callback system where server notifies client if any server files have been updated to have the client invalidate its cache
    - better than the NFS system which required the client to call getAttr to the server to see if something was updated
- Fail Over
    - transferring work/requests from failed server to some other server on a mirrored secondary server
    - data must be mirrored
    - failure of primary server must be detected
    - session state must be renegotiated and established
- Protocols
    - stateful protocols like TCP
        - operations depend on previous operations
        - replacement server must obtain session state to operate properly
    - stateless protocols
        - client supplies necessary context with each request
        - successor servers need no memory of past events
- Security
    - use digital signatures and public key certificates to authenticate the message's sender
    - tamper detection using check sums, cryptographic hashing

    - if you only care about integrity
        - compute a cryptographic hash of your message then encrypt the hash with your private key which is faster than encrypting the whole message(slow)
    - SSL/TLS
        - move encrypted data through socket
        - set up socket and structure to perform cryptography, and hook output of that structure to socket's input
        - reverse process on server side
        - requires particular libraries and sequence of calls into them to allow wide range of generality in supported cryptographic options and how you use them
        - use brand new key to encrypt bulk of data per connection
        - once partner has been authenticated, use symmetric cryptography to encrypt data so best to use a fresh key
        - bootstrap secure connection on asymmetric cryptography when out of usable symmetric keys
        - server and client must negotiate a set of ciphers and techniques that balances security and performance
        - Diffie-Hellman key exchange to create key
            - client obtains certificate containing server's public key and use it in that certificate to verify authenticity of server's messages
            - certificate security based on its embedded cryptography, not the method used to transport it
            - server signs messages with its private key to verify itself to client
            - server is unsure about client's identity at this point but client already authenticated itself with a password prior
            - ok as long as password is checked
            - Question: how does password get back to user
            - the actual package = certificate expiration data, server's public key etc
            - hashed and then encrypted with private key
            - use the same hash provided to hash the info and if it is the same as the one decrypted using the authorities public key we gucci
    - Challenge/response protocol
        - server sends a challenge that must be responded to by user
        - server verifies this response and every challenge is different requiring a different response
    - signed load modules
        - designate certification authorite to verify reliability of software by code review/testing
    -

latent sector errors = disk sectors that have been damaged

Operating Systems Security

- goals
    - confidentiality
    - integrity
    - availability
    - controlled sharing
    - non-repudiation

RAID fail-stop fault model => either disk is failed or its not as opposed to lse latent sector errors
Virtual File System (VFS) Layer

- federation layer to generalize file systems: allow OS to treat all file systems as same and dynamic addition of new ones
- plug-in interface/implementations
- hide implementation from clients

Dynamically Loadable Kernel Modules

- choosing which module to load
  - program needs implementation and a factory to obtain it
  - for browser plugins, there is a MIME-type associated with data to be handled, so a registry to find suitable plug-in can be consulted assuming data is tagged with type and someone is maintaining a tag-to-plugin registry
  - factory loads all known plug-ins and use a probe method to see if any plug-ins could identify the data and then handle it accordingly
  - most I/O busses support self-identifying devices, each of whose info can be used with device driver registry to automatically select the appropriate driver
- loading a new module
  - load into memory
- initialization and registration
  - after module has been loaded into memory, main program calls its initialization method, which allocates memory and I/O resources, initializes driver data structures, assigns I/O resources to devices to be managed, and register all supported device instances
  - modern busses provide mechanisms to discover all available devices and their required resources
  - device driver then allocates those resources from associated bus driver and assigns required resources to each device
- how to use
  - Linux
    - pseudo file system called /dev containing special files associated with registered device instance
    - if process want to open one of those files, OS creates reference from open file instance to registered device instance
    - then process can issue reads, writes, and other system calls on that file descriptor; each call is forwarded to the appropriate device driver entry point
  - higher level frameworks
    - terminal sessions, network protocols, etc
    - each service maintains open references to underlying devices
    - when need to talk to device, OS forwards call to appropriate device driver entry point
  - system maintains table of registered device instances and associated device driver entry points for each standard operation
    - when request is made for device, OS indexes into this table by device identifier to look up address of entry point
    - aka federation frameworks

- unloading
    - un-register itself as device driver
    - shut down managed devices
    - return allocated memory and I/O resources back to OS
    - module can then be unloaded and have its memory freed
- interfaces
    - well-defined set of entry points for any class of device driver; all drivers must implement them
    - well-defined set of functions within OS that modules are allowed to call
- hot-pluggable devices and drivers
    - can be loaded at any time
    - hot-plug manager
        - subscribes to hot-plug events
        - walks configuration space to identify newly inserted device, finds, and loads appropriate driver
        - finds associated driver and call its removal method to remove device
    - hot-pluggable busses have multiple power levels
    - newly inserted device may receive only enough power to be queried and configured
    - when driver is ready to use device, it instructs bus to fully power the device
    - some busses may have mechanical safety interlocks to prevent device from being removed while in use
    - then driver must shut down and release device before it is removed
- backpointer-based consistency (BBC)
    - additional back pointer added to every block for consistency
    - when accessing a file, check for consistency of file by checking if address in inode points to a block that points back to it
- mandatory and discretionary access control
    - some systems have higher authorities besides the administrator that gives and revokes access control(military)
- unreliable communication layers
    - ignore packet loss, assuming some apps know how to deal with it
    - UDP/IP
        - process uses sockets API to establish communication endpoint
        - processes on other machines send UDP datagrams to original process
        - includes checksum to detect packet corruption
- reliable communication layers
    - built on top of unreliable network
    - TCP/IP
    - acknowledgement (ack)
        - sender sends message to receiver, then receiver sends short message confirming receipt if received
        - if not received, we need a timeout
        - sender sets timer to go off after some time after sending message
        - if no acknowledgement has been received after time went off, sender retries sending the message
        - but sometimes it's the acknowledgement, not the message that gets lost
        - so need some way to make sure message is received exactly once
        - sender identifies message uniquely; receiver tracks whether it has already seen the message before. If so, don't pass acknowledgement message back to sender
- sequence counter
    - sender and receiver agree upon start value for counter that they'll both maintain
    - when message is sent, current value of counter is sent too and is ID of the message
    - sender then increments counter
- challenges of distributed synchronization
    - spatial separation: different processes on different systems, no shared memory for locks
    - temporal separation: difficult to "coordinate" spatially separated events

    - independent modes of failure

Information hiding helps the OS encapsulate the complexity and increase ease of use. Also allows users to prevent dependency errors and change the module implementations without harming the overarching functionality.
OS saves PC/PS, the stack pointer, page table
Mechanisms
- Underlying hardware the performs resource management and give/revokes client access to them
- configurable or plug-in policy engine that controls which client gets which resource when

Software Layering
- applications
- OS
- Application Binary Interface(binds the specific API) to the particular ISA)
    - used by the compiler, the OS, the linkage editor, and program loader
    - general libraries
    - drivers/operating system kernel
    - make sure an API is compatible and runs correctly with the particular OS and hardware
- Instruction Set Architecture(ISA)
    - devices/privileged instruction set/general instruction set
- Federation Framework => collection of all available implementations that enables the clients to select the desired implementation and then automatically routes all future requests through that implementation using deferred binding

Compilation/Linkage
- Object Modules => Compiled versions of your program
- Linkage Editing
    - Resolution of symbols by going into libraries and loading the specified data/text segments
    - Relocation: all the symbols at this point in time have relative addresses, but since we have resolved the symbols, we replace the actual addresses here
- Load Module
    - Resolve all external references using linkage editor
    - all modules are combined into a few select segments including(text, data, statically allocated memory without initialization(BSS)
- Shared libraries are initialized and reserved in memory and when programs require shared libraries, it will open the sharable code segments and map them to the correct memory locations
    - Unfortunately, these cannot have any static data, only short lived data can be allocated on the stack
    - also cannot make calls to global variables in the client program
- Shared Libraries vs Dynamic Libraries
    - shared libraries binds to some version chosen at linkage edit while DLLs are chosen at run time
    - shared libraries consume memory throughout run time but DLLs can be unloaded
    - DLLS support complex initialization and communication between application and library
    - support bi-directional calls between client and library
    - more work to load library, register interfaces, and establish work sessions
- Code must be loaded into memory
    - code must be read from load module
    - map segment into process' address space
- Data segments are initialized in address space and copied from load module
    - are read/write and process private
    - program can grow or shrink using sbrk
- Initialize stack frames
    - each procedure call allocates a new stack frame for procedure local variables and parameters.
    - registers are saved and stored

Process Descriptors

- Stores all information relevant to the process
- stores all information relevant to a process
- used for scheduling, security, allocation
- descriptors are stored in a process table containing one entry(PCB) for each process in the system
- PCB(process control block)
        - Linux data structure that keeps track of all processes and their info

Process Creation

- Forking
    - clones existing parent process
    - child shares parent's code but has its own stack and data segment
    - data segment is shared with a copy-on-write feature where if one of them writes to the data segment, then make a copy of that data structure and write to that
- exec
    - remake the process
    - complete replaces the existing program
    - gets rid of child's old code and load brand new set of code, stack and data segment

IPC = Inter-process communication

- Synchronous
    - writes block until message is sent
    - reads block until a new message is available
- Asynchronous
    - writes return when system accepts message
    - Reads return promptly if no message available
- Two major ways of communicating(streams or messages)
- Flow control keeps sender from overwhelming receiver by limiting the number of bytes sent at a time

Types of IPC

- Pipelines
    - simple byte stream w/ simple error conditions and no security
    - named pipes are explicit connections that are persistent, users can open it by name
        - unfortunately writes from multiple writers have no separators
        - no clean fail-overs or failsafes
    - mailboxes
        - data is stored and delivered in distinct messages
        - unprocessed mail remian in mailbox
- Sockets
    - Complex flow control and error handling/trust and security
    - Can get overly complex and overall much slower => deals with security issue, constantly changing addresses, connection/node failures
- Shared Memory
    - mapped into multiple process' address spaces
    - extremely fast, no OS intervention and all data can be seen by all processes that have it mapped in their address space
    - No authentication besides access control on shared file
    - Synchronization to prevent race conditions is the major concern
    - A bug in one process can destroy entire structure

Process Execution

- OS loads initial state into memory, and load the core's registers
- CPU directly executes most application code punctuated by occasional traps and interrupts

- Exceptions
    - Routine
    - Asynchronous
        - supports traps which catch exceptions and transfer control to OS
        - also used for system calls from the program
        - when traps occur they are redirected to a previously set up TRAP vector table that redirects to a trap handler
        - all previous registers PC/PS are pushed onto the stack and address of 1st level trap handler is loaded

Scheduling Performance

- Throughput(processes/second)
- delay
    - turnaround time=>time to finish a job
    - response time
- Time that ready processes spend waiting for CPU
- Overloading
    - When the queue size for processes is maximized, requests are typically dropped and this can cause an infinite response time
    - Solution is graceful degradation which is rejecting performance in favor of maintaining performance

Schedulers

- hard vs soft real time scheduling
    - hard means system fails if deadline is not met
        - scheduler is non-preemptive
        - no run time decisions
    - soft means occasional deadline misses are allowed
        - Earliest Deadline First
- Preemptive Scheduling
    - Scheduler periodically refreshes and finds highest priority ready process
        - if it isn't the current process, force it to yield
        - happens at periodic time intervals due to the processes internal clock that can generate an interrupt at fixed time intervals
        - MLFQ(Multi-Level Feedback Queue)
            - Rule 1: If Priority(A) > Priority(B) A runs
            - Rule 2: If Priorities equal, A & B run in round robin
            - MLFQ varies the priority based on its observed behavior
            - For example, a job relinquishes the CPU while waiting for input will get high priority while a job that has used the CPU for a long time will get its priority reduced.
            - Rules of priority
                - When job enters system, it is placed at highest priority
                - If a job uses up an entire time slice while running, its priority is reduced
                - If a job uses up its time allotment it remains at the same priority level
                - After time period S, move all jobs to the topmost queue
                - Priority Scheduling
                    - linux processes have nice values(soft priority)
                    - priority is constantly adjusted
                        - process that have been run for a long time are lowered
                        - periodically all the lowest priorities are raised to highest
                        - processes that haven't been run are raised

Fragmentation/Memory Allocation

- internal -> caused by fixed sized allocation
- external=> caused by small chunks of memory remaining between large allocated chunks

- Buffer Pools or Segregated Lists
    - List of fixed size chunks of memory that can be requested
    - When buffer pool runs low(and dips below a low point in available space) return space until high space threshold
- Binary Buddy Allocator
    - Memory thought of as size 2^n chunks that it repeatedly splits until it finds the right sized chunks
    - Allocates in fixed size chunks of powers of two, when two side by side chunks(buddy) are free, we just combine them together

  Paging divides the allocated memory into fixed smaller pieces to allow the memory management system to behave more flexibly
- Paging causes about ½ a page on average while segmentation(fixed size allocation) causes about 50% fragmentation

CPU automatically adds base register to every address

When a program is relocated, only the base address registers are reset to the new location

We also has a length or limit register=> any register greater than this is segmentation

Swapping, Paging, Virtual Memory

- Physical memory is divided into physical units of single fixed size called a page frame
- Paging averages only last ½ a page and reduces fragmentation tremendously

Translation

- Virtual address is split into a page number and an offset
- The page number is used as an index into a page table => the offset is then added on to the actual physical page #
- Demand paging puts many of the actual pages into disk and only keeps the most important ones
    - If a page is queried but not present in memory, a page fault occurs and the OS brings in the new page
    - The OS blocks the process and then retries the failed instruction
- LRU(Least Recently Used)
    - All pages are organized in a circular list => reference bits for each page
    - Scan whenever we need a page
    - If the page has been referenced, reset the reference bit and continue
    - If the page hasn't been referenced(reference bit is reset) pull that page out
- Per-Process Frame pools
    - each running process gets an allocation of pages based on working sets
    - working set is the set of pages used by a process in a fixed length sampling window in the immediate past
    - observe the paging behavior(faults per unit time) and adjust the number of page frames
    - Processes that don't use their frames lose them to processes that use their frames more
- Similar to copy-on-write, only pages that are written need to be written back to disk(dirty pages)

TLB

- TLB(Translation Lookaside Buffer)
    - part of the chip's memory management unit
    - TLB is a hardware cache of the most popular virtual-to-physical address translations
        - When a hardware lookup is requested, either there is a TLB hit or TLB miss
        - spatial locality => accessing within the bounds of a page to reduce TLB misses
        - temporal locality => accessing memory elements again in time
    - TLB misses that are handled by software create a trap
        - difference between this trap and other trap handlers is that hardware re-executes the code that handles the trap to result in a TLB hit instead of continuing at the next instruction
    - TLB pages have a valid bit that marks a valid translation and protection bits that tell code how the page can be executed

    - TLB pages differentiate from one another to prevent context switches from using the wrong page table entires using an ASID(address space identifier)
    - Eviction policy => least recently used using the clock as a substitute
- Differs from the invalid bit of a page table entry in that in a page table, the entry is invalid if there is no page at that address
    - TLB entry invalid bit is set if it is not currently valid, in other words it might refer to some other process' address space
    - page table entry is only set if the process has no allocated that page
- To prevent the need to completely flush the TLB buffer when a context switch occurs, we use ASIDs or address space identifiers
- capacity miss => cache runs out of space
- conflict miss => limits on where an item can be placed in hardware cache
- Pages are in virtual memory, page frames are in physical memory => page is mapped to a page frame
- Working set size
    - If we increase the number of page frames, makes very little difference in performance
    - if we reduce the number of page frames, performance suffers dramatically

Turning off Interrupts

- Downsides
    - Must be done in privileged mode
    - Could disable preemptive scheduling and disk IO
    - Delays system response times
    - May lock the program in an infinite loop if the executed code is buggy
    - Concurrent execution can still happen on multicore machines
- When to Use
    - situations involving hardware registers/communications queues

Threads

- When to Use
    - parallel activities in a single program
    - frequent creation and destruction
    - share resources
    - exchange messages
- When to Use processes
    - less creation and destruction
    - running agents have distinct privileges
    - limited sharing of resources
    - no fate sharing
- Each thread has its own registers and stack area
- Thread stacks of a single process share the same address space
    - Since the address space is still one dimensional, multiple thread stacks cannot grow towards a single hole
    - The solution is to limit each threads maximum size, this works b/c most threads are relatively small/simple
- User Threads vs Kernel Threads
    - User threads generally cannot take advantage of multicore system, furthermore, when one thread is blocked, all of the threads are blocked
    - Kernel threads are made aware to the operating system and the OS can choose to schedule them on more than one processor
    - User level threads use voluntary yielding
    - Scheduled system threads use preemption

Condition Variables

- Allow processes to wait for a specific condition without required the process to loop and waste CPU cycles

Still don't quite understand

- named pipes
- DLLs

Synchronization/Concurrency

- why parallelism
    - improved throughput: blocked of one activity does not stop others
    - improved modularity: break complex tasks into smaller, simpler ones
    - improved robustness: failure of one thread does not stop others

Test and Set operates on a bit
Compare and Swap operates on the full 32 bit word

Atomicity

- A enters critical section before B starts
- B enters critical section after A completes
- NO OVERLAP
- An update that starts will complete and an incomplete update will have no effect

Asynchronous completion problem is how to perform asynchronous waits for activities to complete without wasting resources
Conditional Variables

- OS provides condition variables
    - blocks a process when the condition variable is being used, when the desired event has occurred, unblocks process
- Each process should have its own completion event waiting list
    - To prevent further critical section wrongful accesses, we just use the same locking mechanism to lock the critical section in sleep() and wakeup() for conditional variables
- Condition variables allow wait and go without wasting CPU cycles spinning
- Always use while loops to check conditions to wake/signal threads

Semaphores

- A FIFO waiting queue
- wait(P)
    - decrement counter if count >= 0
    - if counter < 0 add process to waiting queue
- Post(V)
    - increment counter
    - If queue non-empty, wake one of the waiting processes

Mutexes

- locks code for a brief amount of time
- typically for multiple threads, low overhead
- PTHREAD_MUTEX_INITIALIZER;

Advisory vs Enforced Locking

- Enforced Locking
    - Done within the implementation of object methods
    - Guaranteed to happen, whether or not user wants it
    - May sometimes be too conservative
- Advisory locking
    - Users expect to lock object before calling methods
    - Gives user flexibility in what to lock
    - Gives users more freedom to do it wrong
    - Mutexes and flocks() are advisory locks

Blocking is 1000x slower than other instructions

int flock(fd, operation) => open instance of file , advisory
int lockf => applies to enitre file, enforced

out of band signals flush buffers in IPC