OS Principles

1. Introduction
    a. Size and complexity creates problems:
        i. Sophisticated sw -> more code required
        ii. Increase complexity -> work evolved in creating a sw much worse than linear to the # of lines of code
        iii. Increase complexity -> difficult to learn and fully understand sw -> err in constructions
        iv. Systems assembled from numerously independent developed and delivered pieces -> small change in one result in big failure
        v. # of independent interacting components increase -> begin to see complex emergent behaviors that could not have been predicted from out experiences in smaller system
        vi. Richer functionality & interaction w/ more external systems -> complete testing combinatorically impossible
        vii. Increasing dependency on sw -> failure cause severe problems more people more often
    b. We need tools and techniques to tame this complexity
    c. OS has long been the largest and most complex sw piece:
        i. Involve complex interactions among many sub-systems
        ii. Involve numerous asynchronous interactions and external originated events
        iii. Involve the sharing of statefull resources among cooperating parallel processes
        iv. Involve coordinated actions among heterogenous components in large distributed systems
        v. MUST evolve to meet ever-changing requirements
        vi. MUST be able to run, without error, for years, despite the regular failures of its components
        vii. MUST be portable to virtually all computer architectures and able to function to heterogenous environments, with different versions of implementations and platforms
    d. Few of these problems are unique to operating systems, but operating systems encountered them sooner and more intensely than other software domains. As a result, these problems have come to be best understood and solutions developed in the context of Operating Systems.
    e. When evolving disciplines encounter these problems, they commonly go back to see how they have been characterized and solved within operating systems.
2. Complexity Management Principles
    a. Layered structure and Hierarchical Decomposition
        i. Hierarchical Decomposition:
            1. Principle for designing new system & study existing system
            2. Operations of these systems are not always strictly hierarchical
                a. May be advantages to minimize non-hierarchical actions

        b. Ex: different groups collaborate to set up an event

b. Modularity and Functional Encapsulation
    i. Requirements: differentiate arbitrary vs. hierarchical decomposition
        1. Each group/component at each level have a coherent purpose
        2. Most functions(for which a particular group is responsible) can be performed entirely within the group/component
        3. The union of the groups/components within each super-group/system is able to achieve the system purpose
    ii. Outcomes:
        1. Possible to encapsulate internal operation of sub-group, and view only the external interfaces
        2. We can examine:
            a. Its responsibilities, and role in system it belonged
            b. Internal structure and operating rules by which is fulfills its responsibilities
        3. External managers and clients can ignore internal structures
        4. Possible to change internal implementations when fulfilling respo
        5. We call that component a module and its implementation details encapsulated within that module
        6. When a system is designed so that all of its components have this characteristic, we say the system design is modular
    iii. Characteristic of good modularity:
        1. Smaller components are easier to understand and manage
        2. Combine closely related functionality in a single module, so that change of one won't break the other
        3. Cohesion: we want the smallest possible modules consistent with the co-location with of closely related functionality
        4. If most operations can be accomplished entirely within a single component, they are likely to be accomplished more efficiently. If many operation involve exchange of service between component:
            a. Overhead of communication between components may reduce system efficiency
            b. Increased number of interfaces to service those inter-component requests increase the complexity of the system and opportunity for misunderstanding
            c. Increased dependency between components increase the likelihood of error/misunderstanding
        5. Another type of cohesion: the scope of component responsibilities must be designed with an eye toward how well it will be possible compartmentalize operations within that component

c. Appropriate Abstracted Interface and information hiding
    i. Appropriate abstraction: an interface let clients to specify parameters that are most meaningful to them and easily get the desire result

ii. Opaque (good info hiding): an interface that does not reveal underlying implementation
iii. Exposing implementation details would:
    1. Expose more complexity to the client->interface difficult to learn
    2. Limit provider's flexibility to change the interface in the future, because previously exposed interface included many aspects of the implementation. Different implementation will lead to incompatible interface

d. Powerful abstractions
    i. All tools and resources in OS are abstract concept of from the imaginations of system architects
    ii. Powerful abstractions can be applied to many situations:
        1. Common paradigms (e.g. lock granularity, cache memory, bottlenecks) -> enable easy understanding of many phenom
        2. Common architectures e.g. federations of plug-in modules treating all data sources as files -> used as fundamental models for new solutions
        3. Common mechanisms e.g. registries, remote procedure calls, in terms of which we can visualize and construct solutions

e. Interface contracts
    i. An interface specification is a contract: a promise to deliver specific results, in specific forms, at the response to requests in specific forms
    ii. If the service-providing component operates according to the interface specification, and the clients ensure that all use is within the scope of the interface specifications, then the system should work, no matter what changes are made to the individual component implementations.
    iii. These contracts do not merely specify what we will do today, but they represent a commitment for what will be done in the future.
        1. VIP in complex system like OS
    iv. If new implementation changes interface spec or user use undocumented features, problems arise
    v. Impossible for comprehensive test -> interface contract VIP!

f. Progressive Refinement
    i. Difficulties:
        1. Estimate the work in a large project
        2. Anticipate problems in a large project
        3. Difficult to get the requirement right in a large project
        4. Often take so long that they become obsolete before finished
        5. Often lose support before they are finished
    ii. Modern methodology embrace some form of iterative/incremental development:
        1. Add new functionality in smaller/one feature at a time projects -> easier to debug, estimate required work, delivery date, better communication and efficiency

2. Rather than pursue large speculative (if you build it they will come) projects, identify specific users with specific needs->better requirements, and build something that addresses those needs ->create more value more quickly
3. Deliver new functionality as quickly as possible to get feedback before moving on to the next step. Most software requirements are speculative. The best way to clarify them is to deliver something and get feedback.
4. It is much easier to plan the next step after we have data from the previous step. Real performance data or user feedback will guide us towards the most important improvements.

3. Architectural Paradigms
    a. Mechanism/policy separation
        i. Basic concept: the mechanism for managing resources should not constrain/dictate the policy according to which those resource is used.
        ii. Primary motivation: system is likely to be used for a long time in many places in a way that designers may never anticipate
        iii. Resource managers should be designed in 2 logical parts:
            1. Mechanisms: keep track of resources and give/revoke client access to them
                a. Ex: card-keys, readers, locks, controlling computer
            2. Policy: controls which clients get resources and when
                a. Ex: rules in access-control database
                b. Mechanism impose little constrain on who can enter
            3. Because policy is independent from mechanism, we can use a very different mechanism to support the exact same policy
            4. It should be possible to change policy/mech w/o greatly impacting the other
            5. Such separation allows building system usable in wide range
    b. Indirection, federation, and deferred binding
        i. Need: a new way to add to add support to a new file system independently from OS to which they're coneected
        ii. Sol: accommodate multiple implementations of similar functionality that is with plug-in modules that can be added to the system as needed
        iii. Features of this solution:
            1. Common absn(class interface) that all plug-in module adhere
            2. Implementation not build-in to OS, but accessed indirectly (e.g., ptr to a specific object instance)
            3. Indirection is often achieved through some sort of federation framework that registers available implementations and enables clients to select object/implementation
            4. The binding of a client to a particular implementation does NOT happen when the sw is build but rather deferred until the client actually needs to access a particular resource

5. The deferred binding may go beyond the client's ability to select an implementation at run-time. The implementing module may be dynamically discovered, and dynamically loaded. It need not be known or loaded into OS until a client requests it.

c. Dynamic Equilibrium
   i. Background: system loads change over time
   ii. Implications: no single fit-all configuration for a complex system
   iii. Most complex system have numerous tunable parameters that can be used to optimize behavior, but they're not enough:
      1. Tuning parameters tend to be highly tied to a particular implementation; proper configuration often requires deep understanding of complex processes
      2. Loads in large systems are subject to continuous change -> needed parameters change -> not practical/economical to adjust system configuration in response to changing behavior
      3. Possible to build automated management agents that continuously monitors system behavior and adjust configuration; BUT possible misinterpretation/drive system to uncontrolled oscillation
   iv. Dynamic equilibrium: contribute to stability of complex natural system
   v. any external event that perturbs the equilibrium automatically triggers a reaction in the opposite direction:
      1. The amount of water in a sealed pot is the net result of evaporation and condensation. When the temperature is raised, more evaporation takes place, which leads to an increase in the steam vapor pressure, which leads to an increased rate of re-condensation.
      2. The deer population may be the net result of food supply and predation by wolves. If the food supply increases, an increase in the deer population will lead to an increase in the wolf population, which will reduce the deer population.

d. The criticality of Data structures
   i. Most of our program state is store in data structures
   ii. Most of out instruction cycles are spent searching, copying anf updating data structures
   iii. Data structures determine
      1. Which operations are fast/slow (due to # of ptrs to be followed)
      2. Which operation is simple/complex (# of different data structures to be updated)
      3. Locking requirements for each operation and hence achievable parallelism
      4. Speed and success probability of error recovery (because of # of possible errors and their detectability/repairability)

iv. when confronted with a difficult performance, robustness, or correctness problem, the solution is found in the right data structures. And given those data structures (and their correctness/coherency assertions) the algorithms often become obvious.

4. Life principles
    a. There are no such thing as a free lunch
        i. Common simple solutions often don't handle all cases
        ii. Often have multiple conflicting goals, always tradeoff
        iii. Before we can safely change a system, we should be able to predict likely consequences, optimize their impacts, prioritize their goals, optimize expected utility
    b. Devil is usually details
        i. our first articulation of that vision is seldom right, and it must be refined as we try to understand how we will apply it in all of the required situations:
            1. enumerating/identifying all of the interesting cases is difficult
            2. may not be obvious how to make particular situation fit in out beautiful high level structure
            3. may encounter 1 single case cannot fit in
            4. May be able to extend model to embrace the troublesome case
            5. Can sub-case, preclude/exclode the troublesome cases
            6. Can conclude that while beautiful, idea cannot work
    c. Keep it simple, stupid and if it's not broken don't fix it
        i. problems we face are complex enough, but sometimes we voluntarily add gratuitous complexity to our solutions.
        ii. It is natural to look at a solution and recognize enhancements that would make it smarter, or more complete, or more elegant. These insights are fun, but they often prove to be counter-productive:
            1. Many optimizations require extra work to better handle special case -> reduce benefits of optimization
            2. Simple idea become complex when handling all cases
            3. Complex solution have complex behavior and new problems
            4. More complex solutions are more difficult to understand, more undetected errors and more difficult to maintain
        iii. Simple solution often work better than a smart one
        iv. Better avoid complex solution unless hard data show simple wont work
    d. Be clear about the goals
        i. detailed goals might be to speed up some operation and eliminate a confusing parameter from some interface. But the real goal was to make the system faster and easier to use
    e. Responsibility and sustainability
        i. No software-managed resource can ever be lost. All memory and secondary storage will be recycled over and over again

ii. Errors cannot be dismissed as unlikely events, and no error can be left unhandled. Every reasonably anticipatable problem must be correctly detected and handled, so that the system can continue executing.
iii. main differences between professional and amateur software are completeness and error handling
iv. responsible for anticipating and dealing with all the consequences of our actions

Interface Stability

1. Interface specifications
    a. Manufacturing with interchangeability: if each part is manufactured and measured to be within its specifications, then ANY collection of the parts can be assembled into a working whole.
    b. Open() system call: opening a file in Posix system
        i. Does not matter for different ways of implementations: lead to the same interface specification → standardization
        ii. Benefits of standardization:
            1. Don't have to write file I/O routines -> programmer happy
            2. Provide the same I/O services -> Program likely to be portable to any Posix complaint system
            3. Everyone knows how to use Posix file I/O functions -> reduce training time
2. The criticality of interfaces in architecture
    a. System architecture:
        i. Defines: the major components, the functionality of each, interface between them.
        ii. One important element: component interface specification 黑盒子
            1. Make it possible to independently design and implement THOSE components
            2. ANY implementation that satisfy those functionality and interface specifications should combine to yield a working system
            3. If not nicely **specified**, likely to interfere/break/crash/can't combine
3. Importance of interface stability
    a. Software interface specification is a form of contract
        i. describes an interface and the associated functionality
        ii. implementation providers agree that their systems will conform to the specification
        iii. developers agree to limit their use of the functionality to what is described
        iv. Based on 3 above, they get the benefits
    b. If change the result/required parameters:
        i. When you promise somebody that you will do something (e.g. conform to a specified interface), and they depend on you (e.g. by writing code to

that interface), and you don't follow through (e.g. make an incompatible change) ... problems are likely to ensue (e.g. failures, bug reports, product gets a bad reputation, going out of business, etc.)

4. Interface vs. Implementations
    a. If I write a program depending on some undocumented code, the code would break because: I am NOT working with interface but with some particular implementation
5. Program vs. User Interfaces
    a. To deliver binary software (usual case) to non-developers, you HAVE to trust whatever platform they run it to correctly implement all the interfaces your program depends.
6. Is every interface carved in stone?
    a. NO. You are free to add features that do NOT change the interface specification
        i. Upwards-compatible extensions: all old programs will work the same way, but new interfaces enable new software to access functionality.
    b. Adding incompatible changes to old interface without breaking backward compatibility:
        i. Interface polymorphism
            1. Old-school languages (C and FORTRAN): single interface specification
            2. Contemporary language: support polymorphism
        ii. Versioned interfaces
            1. Backwards compatibility requirement do NOT prevent us from changing interfaces. It JUST require us continue providing old interfaces to old interfaces
            2. Example:
                a. Java: provide run-time environ version label
                b. Network Protocol (e.g., RPC) session often begin with a version negotiation
    c. No ability to deliver multiple interface versions does NOT necessarily mean we can never make incompatible changes
        i. Some interface MAY not be supported for some time
            1. Provide evaluation/alpha access to propose new interfaces
            2. Interface stability guidelines -> users decide which version to use
            3. Specific support commitments
                a. Will support 3.5 for 5 years
7. Interface stability and design
    a. If we want to expose an interface to unbundled software with high quality requirements, we are not allowed to change it in non-upwards-compatible ways
    b. We can:
        i. Rearrange distribution of functionality between components -> create a simpler interface between them
        ii. Design features that we may NEVER implement

iii. might introduce (seemingly) unnatural degrees of abstraction in our services to ensure that they leave us enough slack for future changes

Software Interface Standards

1. Introduction
    a. Change 1:
        i. Initially, hw manufacturers: compatibility is an anti-goal
        ii. Independent Software Vendors (ISV) & killer applications
            1. Cost of different versions for different platforms is high
            2. Computer sale driven by the applications they can support
            3. Software Portability became a matter of live and death for BOTH hw manufacturers and sw vendors.
    b. -> need detailed specifications for ALL services
    c. Change 2:
        i. 1960s: computers used by professionals, industry, and business
            1. Prepared to deal with adverse effect of a new sw release
        ii. Now: ordinary customers:
            1. Upgrade: cannot break existing applications
            2. New applications: work w/ both old and new version.
    d. Impossible to test EVERY OS environ -> kinda test-impossible -> need to find a way to ensure software component combinations work when summed up -> solution: detailed specifications + comprehensive compliance testing
2. Challenges of software interface standardization
    a. Standards are a good thing:
        i. Extensively reviewed by experts -> well-considered
        ii. Platform-neutral
        iii. Clear and complete specifications & well-developed conformance testing procedure
        iv. give technology suppliers considerable freedom to explore alternative implementations (to improve reliability, performance, portability, maintainability), as long as they maintain the specified external behavior. Any implementation that conforms to the standard should work with all existing and future clients.
    b. Standards can be a bad thing:
        i. Constrain range of possible implementations
        ii. application may use any feature in any way that is not explicitly authorized by the standard
        iii. difficult to evolve to meet the requirements (when many stake-holders depends on the standard)
            1. many competing opinions about exactly what the new requirements should be and how to best address them
            2. any change (no matter how beneficial) will adversely affect some stake-holders, and they will object to the improvements

c. confusing interfaces w/ implementation
    i. true implementation neutrality is very difficult
        1. because existing implementations often provide the context against which we frame our problem statements
        2. standard 10 yrs ago may no longer wotk
    ii. interface specifications are written after the fact
        1. Since the implementation was not developed to or tested against the interface specification, it may not actually comply with it
        2. In the absence of specifications, it may be difficult to determine whether particular behavior is a fundamental part of the interface or a peculiarity of the current implementation.
d. Rate of evolution
    i. Computer tech evolves quickly, sw evolves with them to exploit them
    ii. Maintaining stable interfaces in the face of fast and dramatic evolution is extremely difficult: conflict between stable interfaces and embracing change
        1. Maintain strictly compatibility w/ old interfaces
            a. not supporting new applications and/or platforms.
            b. This is (ultimately) the path to obsolescence
        2. Developing new interfaces that embrace new technologies and uses, but are incompatible with older interfaces and applications.
            a. Sacrifices existing customers for the chance to win new ones.
        3. compromise that partially supports new technologies and applications, while remaining mostly compatible with old interfaces.
            a. This is the path most commonly taken
            b. but often prove to be both uncompetitive and incompatible.
        4.
e. Proprietary vs. open standards
    i. Proprietary interface: developed and controlled by a single organization
    ii. Open standard: An *Open Standard* is one that is developed and controlled by a consortium of providers and/or consumers (e.g. the IETF network protocols)
    iii. Decision making:
        1. Should they open their interface definitions to competitors, to achieve a better standard, more likely to be widely adopted? This costs of this decision are:
            a. Reduced freedom to adjust interfaces to respond to conflicting requirements.
            b. Giving up the competitive advantage that might come from being the first/only provider.

        c. Being forced to re-engineer existing implementations to bring them into compliance with committee-adopted interfaces.
    2. Should they keep their interfaces proprietary, perhaps under patent protection, to maximize their competitive advantage? The costs of this decision are:
        a. If a competing, open standard evolves, and ours is not clearly superior, it will eventually lose and our market position will suffer as a result.
        b. Competing standards fragment the market and reduce adoption.
        c. With no partners, we will have to shoulder the full costs of development and evangelization. In an open standards consortium, these costs would be distributed over many more participants.
3. Application Programming Interface (API)
    a. Look-up and exploit some services
    b. Example: open(2)
        i. A list of included routines/methods, and their signatures (parameters, return types)
        ii. A list of associated macros, data types, and data structures
        iii. A discussion of the semantics of each operation, and how it should be used
        iv. A discussion of all of the options, what each does, and how it should be used
        v. A discussion of return values and possible errors
    c. API specifications are written in source programming level (e.g., C, C++, Java, Python)
    d. describe how source code should be written in order to exploit these features
    e. basis for software portability, but applications must be recompiled for each platform
        i. application developers happy: application written to a particular API should easily recompile and correctly execute on any platform that supports that API
        ii. platform suppliers happy: any application written to supported APIs should easily port to their platform
        iii. BUT: only true when API is platform-independent:
            1. Some int might not work on 32-bit machine
            2. Individual byte access might NOT work on big-endian machine
            3. Particular feature implementation might not work on some platform (fork(2) on Windows)
4. Application Binary Interface (ABI)
    a. Def: An Application Binary Interface is the binding of an Application Programming Interface to an Instruction Set Architecture

b. API defines subroutines, what they do, and how to use them. An ABI describes the machine language instructions and conventions that are used (on a particular platform) to call routines
c. Typical ABI contains:
    i. The binary representation of key data types
    ii. The instructions to call to and return from a subroutine
    iii. Stack-frame structure and respective responsibilities of the caller and callee
    iv. How parameters are passed and return values are exchanged
    v. Register conventions (which are used for what, which must be saved and restored)
    vi. The analogous conventions for system calls, and signal deliveries
    vii. The formats of load modules, shared objects, and dynamically loaded libraries
d. Portability benefits of ABI >> API
    i. As long as the CPU and Operating System support that ABI, there should be no need to recompile a program to be able to run it on a different CPU, Operating System, distribution, or release
e. Uses of ABI:
    i. Compiler: generate code for process entry, exit, calls
    ii. Linkage editor: create load module
    iii. Program loader: read load module into memory
    iv. OS: process sys calls
    v. Crazy people programming in assembly

Object Modules, Linkage Editing, Libraries

1. Introduction
    a. Process:
        i. Def: executing instance of a program
        ii. one of the most fundamental abstract resources implemented by OS
    b. Programs:
        i. sources files can be translated into machine language and combined with other machine language modules to create executable programs
        ii. files full of 1's and 0's
2. Software Generation tool chain (for compiled languages)
    a. General Classes:
        i. Source modules
            1. Editable text in some language (C, Java, assembler, etc.)
            2. Can be translated into machine language by compiler OR assembler
        ii. Relocatable object modules
            1. Sets of compiled OR assembled instructions created from individual source modules

2. NOT yet complete programs

iii. Libraries
1. Collections of object module
2. From which we can fetch functions that are required by (and NOT contained in) the original source/object modules

iv. Load modules
1. Complete programs
a. Usually created by combining numerous object modules
2. Ready to be loaded into memory (by OS) and executed (by CPU).

b. Components of Software tool chain

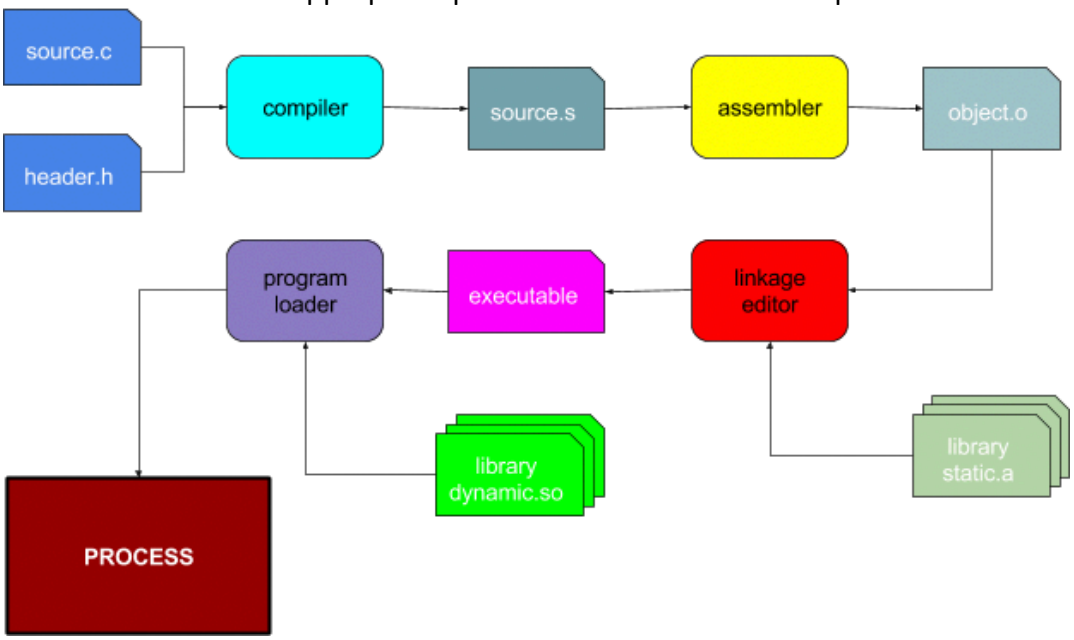
i. Compiler
1. Generate low level code: often in assembly language code rather than machine code (but Java and Python compiles directly into pseudo-machine language) -> flexibility (optimization), make compiler more portable, simplifies the compiler
a. Read source modules & included headers
b. Parse the input language
c. Infers the intended computations

ii. Assembler
1. Allows:
a. Declaration of variables
b. Use of macros
c. referenced to externally defined code and data
2. user mode:
a. performance critical string
b. data structure implementations
c. routines to implement calls into OS
3. OS:
a. CPU initialization
b. First level trap/interrupt handlers
c. Synchronization operations
4. Output: object module containing mostly machine language code
a. BUT because the output correspond ONLY to a single input module for linkage editor:
i. some functions (or data items) may not yet be present, and so their addresses will not yet be filled in
ii. even locally defined symbols may not have yet been assigned hard in-memory addresses, and so may be expressed as offsets relative to some TBD starting point

iii. Linkage editor
1. Reads specified set of object modules

2. Place them consecutively in VM space & note where in VM is placed
3. Notes unresolved references (symbols referenced by but not defined in the loaded program)
4. Searches a specified set of libraries to find object modules that can satisfy those references & place them in VM
5. Resulting bits: represent a program that is ready to be loaded into memory and excutable
    a. Written into a new file: executable load module

iv. Program loader
1. Part of OS
2. Examine the info in a load module
3. Creates an appropriate virtual address space
4. Reads the instructions
5. Initialize values form the load module into the virtual address space
6. If the load module includes references to additional (shared) libraries, the program loader finds them and maps them into appropriate places in the virtual address space as well



3. Object modules
    a. We do NOT put all code that will be executed into one single file:
        i. Too huge.
        ii. Hard to understand
        iii. Cumbersome to update
        iv. Many functions are commonly used (string, output, etc.), REUSE
    b. Most programs are created by COMBINING multiple relocatable object modules
        i. Relocatable object module differs object module:
            1. MAY be incomplete (reference code defined in other modules)

2. Has NOT yet been determined where they'll loaded into VM

c. Code within object module is ISA dependent, but structure shares: ELF

4. Libraries
    a. Def: a collection of usually related object modules
    b. Reusable codes are often, but not necessarily, distributed to public
    c. Different OS may implement libraries differently
    d. Linux command for creating, updating, and examining libraries: ar(1)
    e. Libraries are ALWAYS orthogonal AND independent:
        i. It is common to implement higher level libraries (e.g. image file decoding) using functionality from lower level libraries (e.g. mathematical functions and file I/O)
        ii. It is not uncommon to use alternative implementations for some library functionality (e.g. a diagnostic memory allocator) or to intercept calls to standard functions to collect usage data
        iii. This means that the order in which libraries are searched may be very important. If we call a function from library A, and library A calls functions from library B, we may need to search library A before searching library B. If we want to override the standard *malloc(3)* with valgrind's more powerful diagnostic version, we need to search the valgrind library before we search the standard C library
5. Linkage editing: turn relocatable obj files into runnable program
    a. Resolution
        i. Def: search the specified libraries to find object modules that can satisfy all unresolved external references
        ii.
    b. Loading
        i. Def: lay the text and data segments from all of those object modules down in a single virtual address space, and note where (in that virtual address space) each symbol was placed
        ii.
    c. Relocation
        i. Def: go through all of the relocation entries in all of the loaded object modules, each reference to correctly reflect the chosen addresses
    d. Linux to form linkage editing: ld(1)
6. Load modules
    a. Similar in format to an object module; BUT:
        i. Complete
        ii. Requires NO relocation
    b. Linux to load a new program: exec(2) system call
    c. When OS load a new program:
        i. Consult the load module to determine the data and text section size and locations
        ii. Allocate appropriate segments in VA
        iii. Read the contents of text and data from load module into memory

iv. Create a stack segment, initialize %rsp
v. Program ready to execute!!

7. Static vs. Shared libraries
    a. Downside of static linking:
        i. Many libraries are used my almost every program in system: large space
        ii. Popular libraries change over time (optimizations, etc.)
    b. Steps to implement shared libraries:
        i. Reserve an address for each shared libraries
        ii. Linkage edit each shared library into a read-only code segment, loaded at the address reserved for that library.
        iii. Assign number (0-n) to each routine; put a redirection table at the beginning of that shared segment
        iv. Create a stub library, that defines symbols for every entry point in the shared library, but implements each as a branch through the appropriate entry in the redirection table. The stub library also includes symbol table information that informs the operating system what shared library segment this program requires.
        v. Linkage edit the client program with the stub library
        vi. When the operating system loads the program into memory, it will notice that the program requires shared libraries, and will open the associated (sharable, read-only) code segments and map them into the new program's address space at the appropriate location. This process is described in ld.so
    c. Outcome:
        i. Single copy of shared library implementation can be shared among programs
        ii. The version of the shared segment that gets mapped into the process address space is chosen, not during linkage editing, but rather at program load time -> The choice of which version to use may be controlled by a library path environment variable
        iii. Client programs not affected by the change in library size and modules
        iv. Possible for shared libraries make calls to another
    d. Limitations:
        i. Shared segment's code is READ-only; CANNOT have static
        ii. The shared segment will not be linkage edited against the client program, and so cannot make any static calls or reference global variables to/in the client program
        iii. There may be very large/expensive libraries that are seldom used; Loading/mapping such libraries into the process' address space at program load time unnecessarily slows down
        iv. While loading is delayed until program load time, he name of the library to be loaded must be known at linkage editing time
8. Dynamically Loaded Libraries
    a. General Model:

i. The application chooses a library to be loaded (perhaps based on some run-time information like the MIME type in a message).
ii. The application asks the operating system to load that library into its address space.
iii. The operating system returns addresses of a few standard entry points (e.g. initialization and shut-down). The application calls the supplied initialization entry point, and the application and DLL bind to each other (e.g. by creating session data structures, exchanging vectors of service entry points).
iv. The application requests services from the DLL by making calls through the dynamically established vector of service entry points. When the application has no further need of the DLL, it calls the shut-down method and asks the operating system to un-load this module.

b. Linux: dlopen, dlsys, dlclose, etc.

9. Implicitly Loaded Dynamically Loadable Libraries

Stack Frames and Linkage Conventions

1. Introduction
    a. Fundamental questions:
        i. What constitutes the state of a computation; how can the state be saved and restored?
        ii. What are the mechanisms by which one sw component can request services, and receive results from another?
2. Stack model of program languages
    a. Modern programming languages: procedure-local variables
        i. Auto allocated when procedure is entered
        ii. Only visible to code within THAT procedure
        iii. Each distinct procedure has its distinct set of variables
        iv. Auto deallocated when the procedure exits/returns
    b. Stack: LIFO
        i. New stack frames pushed when a procedure is called
        ii. Old stack frames get popped off when procedure returns
        iii. Stack model for procedure calls is universal: hw instructions for stck mgmt. available
        iv. Depend on stack model to handle subroutines and exceptions
        v. Stack model does NOT work for all memory allocation needs:
            1. Data itens must last longer than the procedure: files read into mem
                a. Handled by: HEAP
3. Subroutine Linkage Conventions
    a. Highly ISA dependent
    b. Basic elements:
        i. Parameter passing
        ii. Subroutine call: save ret address on stack, pass control to entry point

iii. Register saving: save non-volatile registers
iv. Allocate space for local variables

c. Return:
i. Return values: place ret value in the stack
ii. Pop local storage off the stack
iii. Register restoring: restore non-volatile registers
iv. Subroutine return: transfer control to the ret address

d. NOTE:
i. Register saving is the responsibility of the called routine
ii. Cleaning parameters off of the stack is the responsibility of the calling routine
iii. Clear delineation of the responsibilities between caller and callee makes it possible to have C programs called by FORTAN

4. Traps and Interrupts
a. Common points:
i. Transfer control
ii. Need to save state of running computation before passing control
iii. After the event handled, restore the state and continue

b. Differences:
i. Procedure call:
1. Requested by running sw; which expects upon return some functions have been performed and appropriate values returned
2. -> linkage conventions under sw control
3. For running sw NOT expecting a trap/interrupt, will behave as if never happened after the handling

ii. Trap:
1. Initiated by hw
2. -> linkage conventions strictly defined by hw
3.

c. Typical process:
i. Number (0, 1, 2, …) associated w/ each exception
ii. A table (OS initialized) associated w/ PC/PS (program status) pair w/ each possible interrupt/exception
iii. Triggered:
1. CPU uses number to index into the appropriate vector table
2. CPU loads a new PC/PS word from the vector
3. CPU pushes the PC/PS word onto CPU stack
4. Execution continues w/address specified by the new PC
5. The selected code (first level handler):
a. Save all general registers on stack
b. Gathers info from the hw on the cause of interrupt/exception
c. Choose the appropriate second level handler

d. Make normal procedure call to the second level handler, which actually deals with the exception/interrupt

iv. Return to the first-level handler

1. FLH restores all of the saved registers
2. the 1st level handler executes a privileged return from interrupt or return from trap
3. CPU reloads the PC/PS to the point of interrupt in main
4. Execution resumes

d. 100x or 1000x more expensive then procedure call: crash CPU cache

5. Summary

SIGNAL(2)

KILL(2)

Real Time Scheduling

1. Introduction
    a. Priority bases scheduling:
        i. Adjusted based on whether the program is computational or interactive heavy
        ii. "best-effort" approach
2. Real-time Systems
    a. Def: the system where correctness depend on timing AND functionality
    b. Metrics:
        i. Timeliness: how closely does it meet the timing req (ms/day of acu. tardiness)
        ii. Predictability: how much deviation is there in delivered tardiness
    c. Concepts:
        i. Feasibility: whether possible to meet the req for particular task
        ii. Hear real-time: strong req that specified task be run in specified interval
            1. OR: system failure
        iii. Soft real-time: intention to provide good response time
            1. OR: missing deadline or degraded performance or recoverable failures
    d. Characteristics:
        i. Knowing how long each task may take -> intelligent scheduling
        ii. Starvation (of low priority tasks) allowed
        iii. Workload relatively fixed -> BOTH high utilization and response time
3. Real-time scheduling algorithms
    a. Simplest real-time system: all tasks and their exe time known -> might not be a scheduler
    b. Static scheduling:

        i. Based on the list of tasks to run and expected completion time, define (at design OR build-time) a fixed schedule
    c. Dynamic scheduling:
        i. For real-time systems, workload changes from time-to-time, based on external events
        ii. KEY:
            1. How to choose the next task to run
                a. Shortest job first
                b. Static priority: highest priority ready task
                c. Soonest start-time ddl first (ASAP)
                d. Soonest completion time ddl first (slack time)
            2. How to handle overload (infeasible req)
                a. Best effort
                b. Periodicity adjustments: run slower priority tasks less often
                c. Work shedding: stop running low priority task entirely
        iii. Preemption
            1. In ordinary time-sharing: ☺
                a. Break up exe of long programs
                b. Prevents a buggy program from taking over the CPU (inf loop)
            2. In real-time scheduling: ☹
                a. Surely miss the completion ddl
                b. Often know expected time -> no need
                c. Embedded and real-time system run fewer and simpler tasks than general purpose system; code well-tested -> rarely inf loop thing
    d. For the least demanding real time tasks, a sufficiently lightly loaded system might be reasonably successful in meeting its deadlines. However, this is achieved simply because the frequency at which the task is run happens to be high enough to meet its real time requirements, not because the scheduler is aware of such requirements. A lightly loaded machine running a traditional scheduler can often display a video to a user's satisfaction, not because the scheduler "knows" that a frame must be rendered by a certain deadline, but simply because the machine has enough cycles and a low enough work load to render the frame before the deadline has arrived.
4. Real-time and Linux
    a. Linux: NOT designed to be an embedded or real-time OS
        i. But many applications require this probability
    b. Now supports real-time scheduler:
        i. Commands: sched setscheduler(2)
    c. Windows: not suited for real-time
        i. Offers NO native real-time scheduler
        ii. Favor general purpose throughput over meeting the deadline

iii. Low load -> provide fast-enough service for some soft real-time req
1. Examples: play music/video

Garbage Collection and Defragmentation

1. Introduction
   a. Garbage collection: seeking out non-longer using resources and recycle them for reuse.
   b. Defragmentation: reassigning and relocating the previously allocated resources to eliminate external fragmentation to create densely allocated resources with large contiguous pools of free space
   c. Similarity:
      i. Both remedy unusable space
      ii. Both are active resource mgmt. techniques (resource manager engaged & manages)
      iii. Often applied in tandem (1st GC, then D)
2. Garbage Collection
   a. How allocated resources returned:
      i. Explicit/implicit action on client
         1. close(2) a file, free(3), delete(), return from C/C++ subroutine, exit(2) to terminate process
      ii. resource manager (OS) maintains active count for each resource
         1. resources shared by multiple concurrent clients
         2. Example
            a. Increment the reference count when new reference of the obj it obtained
            b. Decrement the reference count when reference is released
            c. When ref cnt ==0, free the obj automatically
      iii. Above two approach not always available:
         1. OS may not know about some resource reference (in languages where can copy and destroy resource reference (ptr))
         2. Want to make programmers life easier
         3. Some explicit resources (DHCP address) may NEVER be explicitly freed -> MUST get recycled after hasn't been used for some time
         4. Some resource's allocation.release is very often -> keep track becomes a significant overhead & performance loss
      iv. SOLUTION: garbage collection
   b. Garbage Collection
      i. Used in systems that:
         1. Resources are allocated BUT never explicitly freed
         2. When the pool of available resources become small, GC start:
            a. Begin w/ list of resources that originally exists
            b. Scan the process to find all resources that are still reachable

c. each time a reachable resource is found, remove it from the original resource list.
d. at the end of the scan, anything that is still in the list of original resources, is no longer referenced by the process, and can be freed
e. after freeing the unused resources, normal program operation can resume

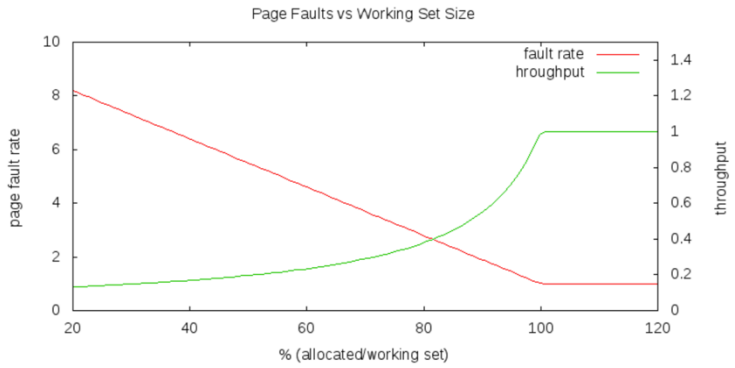
ii. Garbage Collection only works if it is possible to identify all active resource references

3. Defragmentation
   a. coalescing is only effective if adjacent memory chunks happen to be free at the same time. If short-lived allocations are adjacent to a long-lived allocation, or if allocations and deallocations are extremely frequent, there may be very few such opportunities.
   b. How important is contiguous allocation:
      i. if we are allocating blocks of disk, the only cost of non-contiguous allocation may be more and longer seeks, resulting in slower file I/O.
      ii. if we are allocating memory to an application that needs to create a single 1024 byte data structure, the program will not work with four discontiguous 256 byte chunks of memory.
      iii. Solid State Disks (based on NAND-Flash technology) may allocate space one 4K block at a time; but before that block can be rewritten with new data it must be erased ... and erasure operations might be performed 64MB at a time.
   c. Defrag: actually changes which resource is allocated
4. Conclusions

Working sets

1. LRU is not enough
   a. LRU: least recently used
   b. Nobody knows what will happen tomorrow, best guess is similar to today
   c. Unless you have inside information (e.g. understanding non-random patterns), it is generally held that there is little profit gained from trying to out-smart LRU.
   d. Why LRU works well: program exhibits temporal and spatial locality
      i. If they use some code/data, likely to come back access same location
      ii. Access code/data in particular page, likely to reference other code in same vicinity
   e. BUT: above is based on a SINGLE program/process. If we are in the situation of round robin time sharing w/ multiple process we give each process a brief opportunity to run, after which we run all other processes before running it again. With the exception of shared text segments, separate process seldom access the same page
      i. Most recently sued pages in memory belong to the process that will not be run for a long time
      ii. Least recent used page in memory belong to the process that is about to run

        iii. This destroys strict temporal and special locality, as reference behavior becomes periodic, rather than continuous
    f. In the absence of temporal and spatial locality, Global LRU make poor decisions, and does not work well with round-robin.
    g. Pre-process LRU: It might make sense to give each process its own set of page frames. When that process needed a new page, we would let LRU replace the oldest page in set. This would work well w/ round-robin
2. The concept of a working set
    a. Paging observation:
        i. We do not need to give each process as many pages of physical memory as appeared in virtual address space
        ii. We do not even need to give each process as many pages of its virtual address as it will ever access
        iii. It is a good thing if a process experiences occasional page faults and has to replace old with new pages.
        iv. What we want to avoid is page faults due to running a process in too little memory.
        v. We want to keep page fault in manageable rate
        vi. IDEAL: mean-time-between page faults = time slice length
        vii. As we give a process fewer page frames to operate, page fault rate ++, performance – (more time on page fault instead of useful computation)
    b. Thrashing: the dramatic degradation in system performance associated with not having enough memory to efficiently run all of the ready processes
        i. If we think that we can have 25 processes in ready queue, then we had better have enough memory for all 25 of those processes
        ii. If we only a=have enough memory to run 15 processes, then we should take the other 10 out of the ready queue (i.e. by swapping them out)
        iii. The improved performance resulting from reducing the number of page fault will more than make up for the delays processes experienced while being swapped between primary and second storage
    c. For any given computation, at any given time, there is a number of pages that:
        i. If we increase the # of page frames allocated to that process, it makes very little difference to the performance
        ii. If we reduce the # of page frames allocated to that process, the performance suffers noticeably.
    d. Working set size: # of processes at any particular time



3. How large is a working size
    a. Different computation require different amounts of memory
        i. A tight loop may only need two pages
        ii. Complex combinations of functions applied to large amounts of data could require hundreds
    b. Single program may require different amount of memory at different point
    c. Requiring memory is not necessarily a bad thing. Just that reflect cost of comp
    d. Different programs have different level of interactivity -> if we can characterize each program's interactivity -> more efficiently schedule
    e. We can infer process's working set size by observing its behavior
        i. Process experience many page faults: workingSet < mem allocated
        ii. No page fault: workingSet > mem allocated (too much allocated)
    f. If we allocate the right amount of memory to each process, we will minimize the page faults (overhead) and maximize throughput (efficiency)
4. Implementing working set replacement
    a. Regularly scanning all of memory to identify the least recently used page is a very expensive process
    b. With global LRU, we see that clock-like scan has similar effect but much cheaper
    c. The clock hand position is a surrogate for age:
        i. The most recently examined pages are immediately behind the hand
        ii. The pages we examined longest ago are immediately in front of hand
        iii. If the page immediately in front of the hand has not been referenced since the last time it has been scanned, it must be very old even if it's not actually the oldest page in memory
    d. Each page frame is associated with an owning process
    e. Each process has an accumulated CPU time
    f. Each page frame will have a last referenced time, value, taken from the accumulated CPU timer of its owning process
    g. We maintain a target age parameter, which is keep in memory goal for all pages

```
while ...
{
        // see if this page is old enough to replace
        owningProc = page->owner;
        if (page->referenced) {
                // assume it was just referenced
                page->lastRef = owningProc->accumulatedTime;
                page->referenced = 0;
        } else {
                // has it gone unreferenced long enough?
                age = owningProc->accumulatedTime - page->lastRef;
                if (age > targetAge)
                        return(page);
        }
}
```

h. Key elements of the above algorithm;
    i. Age decisions are not made on the basis of clock time, but accumulated CPU time in the owning process. Pages only age when their owner runs without referencing them
    ii. If we find a page that has been referenced since the last scan, we assume it was just referenced
    iii. If a page is younger than the target age, we do not want to replace it -> recycling -> recycling young pages indicates thrashing
    iv. If a page is older than the target age, we take it away from its current owner and give it into a new needy process
i. If there is no pages older than the target age, we apparently have too many processes to fit into the available memory
    i. If we complete a full scan w/o finding anything that is older than the target age, we can replace the oldest page into memory.
        1. Cost: expensive complete scan that the clock algorithm is supposed to aovid
    ii. If we believe that our target age as well chosen, to avoid thrashing, we probably need to reduce the # of processes in memory

5. Dynamic equilibrium to the rescue
    a. ⇔Page stealing algorithm: process which need a new page steals it from a process that does not seem to need it as much
        i. Every process is continuously losing pages that it has not recently referenced
        ii. Every process is continuously stealing pages from other processes
        iii. Processes that reference more pages more often will accumulate larger working sets
        iv. Processes that reference fewer pages less often will find their working sets reduced
        v. When programs change their behavior, their allocated working set adjusts promptly and automatically
        vi. The continuously opposing processes of stealing and being stolen from will automatically allocate the available memory to the running processes in proportion to their working set sizes
        vii. It does not try to manage processes to any pre-configured notion of a reasonable working set size. Rather it manages memory to minimize the number of page faults, and to avoid thrashing.