Other Notes

Saturday, June 29, 2019 12:37 PM

OS Principles (6/29/19)

2.2 Modularity and Functional Encapulation

Hierarchical decomposition: top-down decomposing. e.g. UC system (campuses to colleges to departments etc)

Assigning into groups is not enough. We also need:

- each group at each level to have a coherent purpose
- most functions can be performed entirely within that group/component
- the union of the groups compoents is able to achieve the system's purpose

This makes it easier to "opaquely encapsulate the internal operation of a sub-group/component, and to view it only in terms of its external interfaces."

The external managers and clients can ignore the internal structure and operating rules if a compoent is able to effectively fulfill its responsibilities. Like this, if there are changes made inside the component, the external managers don't have to be concerned as long as the component still does what it is supposed to.
Definition: When it is possible to understand and use the functions of a component without understanding the internal structure, we all that component a *module* and say that its implementation details are *encapsulated* within that module.

Characteristics of good modularity:

- Larger modules are more complex than smaller modules, so it makes more sense to have a larger number of smaller modules. It is much easier to understand the roles/responsibilities of the smaller modules.
- There might be two separate componets who do very closely related operations on the same resources (e.g. deposits and withdrawals in a bank account). There is a danger of a change taking place in one component breaking the other component. It is good practice to place two components who exhibit this characteristic in the same module. "We want the smallest possible module consisting with the co-location of closely related functionaly." This is called **cohesion**, and a module that exhibits this characteristic is called **cohesive.**
- If it is possible for operations to be accomplished entirely within the same module, it increases efficiency. Having many components have to exchange and communicate might risk:
  - overhead of communication reducing efficiency
  - more complex system as the the number of interfaces to services whose inter-component requests increases
  - higher likelihood of misunderstanding or errors as there is more dependency between the components.

2.3 Appropriately Abstracted Interfaces and Information Hiding

We want to hide as much of the information about the component from the user as possible. E.g.
a faucet. A two valve faucet one delivering hot water and one delivering cold water is not preferred because we do not want the user to try and guess the flow rate and temperature by trial and error. We want to *opaquely encapsulate* the water delivery mechanism, so two separate controls one more temperature and one for flow rate makes more sense.

Definition: An interface that enables a client to specify the parameters that are more meaningful to them and easily get the desired results is said to embody **appropriate abstraction**. An interface that does not provide a client with what they want is said to be *poorly abstracted.*

Exposing implementation details, even if the internal implementation was well suited to the intended clients raises other challenges:

a. expose more complexity to the client, making the interface more difficult to describe/learn
b. limit the provider's flexibility to change the interface in the future, because the previously exposed interface included many aspects of the implementation. A different implementation would expose clients to an incompatibile interface.

2.4 Powerful Abstraction

Quitely simply, we want problems to be solved using tools we already have. In OS, a powerful abstraction is one that can be profitably applied to many situations:

- common paradigms that enable us to more easily understand a wide range of phenomena (e.g. lock granularity, cache coherency, bottlenecks.)
- common architectures that can be used as fundamental models for new solutions (e.g. federations of plug-in modues treating all data sources as files.)
- common mechanisms in terms of which we can visualize and construct solutions (e.g. registries, remote procedure calls).

2.5 Interface Contracts

An **interface specification** is a contract: a promise to deliver specific results, in specific forms, in response to requests in specific forms. If all is done as it is within the scope of the interface specification, then the system should work, no matter what changes are made to the individual component implementations.

These contracts do not merely specify what we will do today, but they are a commitment for what will be done in the future. A new client should not use a component in an out-of-spec way because problems are likely to happen in the future (e.g. a system failure.)

3.1 Mechanism/Policy Separation

The mechanisms for managing resources should not dicrate or greatly contrain the policies according to which those resources are used. A single resource management strategy needs to be in two logical parts:

- the **mechanisms** that keep track of resources and give/revoke client access to them
- a configurable or plug-in **policy** engine that controls which clients get which resources when

A good example of mechanism/policy separation can be seen in card-key lock systems:

- each door has a card-key reader and a computer-controlled lock.
- each user is issued a personal card-key.
- to get access to a room, a user swipes a card-key past the reader.
- a controlling computer will lookup the registered owner of the card-key in an access control database in order to decide whether or not that user should be granted access to that door at that time, and (if appropriate) send an unlock signal to the associated lock.

The physical *mechanisms* are the card-keys, readers, locks, and controlling computer. The resource allocation *policy* is represented by rules in the access control data-base, which can be configured to support a wide range of access policies. The mechanisms impose very few constraints (beyond those of the rule language) on who should be allowed to enter which rooms when.

3.2 Indirection, Federation, and Deferred Binding

It is not smart to write an OS that understands all possible file formats as it will lead to problems as the file formats change and evolve. To add support for new file systems independently from the OS, we use **plug-ins**.

Many services (e.g. device drivers, video codec, browser plug-ins) have similar needs, so we accommodate multiple implementatuins of similar functionality with plug-in modules that can be added to the system as needed. Such implementations typically share a few features:

- A common abstraction (class interface) to which all plug-in modules adhere.
- the implementations are not built-in to the OS, but accessed **indirectly** (e.g. thru a pointer to a specific object instance.)
- the indirection is often achieved through some sort of **federation framework** that registers available implementations and enables clients to select the desired implementation, and automatically routes all future requests to the selected object/implementation.
- the **binding** of a client to a particular implementation does not happen when the software is built but rather is **deferred** until the client actually needs to access a particular resource.
- 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 to or loaded into the OS until a client requests it.

3.3 Dynamic Equilibrium

The loads on most systems change over time. There is no such thing as a single, one-size-fits all configuration for a complex system. Most systems have tunable parameters that can be used to optimize behavior for a given load. These tunable parameters are not enough because:

- Tuning parameters tend to be highly tied to a particular implementation, and proper configuration often requires deep understanding of complex processes.
- Loads in large systems are subject to continuous change, and parameters that were right five seconds ago may be wrong five seconds from now. It is neither practical nor economical to expect human beings to continuously adjust system configuration in response to changing behavior.
- It is possible to build automated management agents that continuously monitor system behavior, and promptly adjust configuration accordingly. But such automatons can misinterpret symptoms or drive the system into uncontrolled oscillation.

Stability of a complex natural system is often the result of a **dynamic equilibrium**, where the state of the system is the net result of multiple opposing forces, and any external event that perturbs the equilibrium automatically triggers a reaction in the opposite direction (e.g. deers/food/wolves system)

4.5 Responsibility and Sustainability

"It is commonly observed that the main differences between professional and amateur software are completeness and error handling."

Software Interface Standards (6/29/2019)

Standards are a good thing:

- Experts debate the details over long periods of time and extensively review them to so it encompasses a wide range of applications.
- Platform-neutral: not greatly favoring or disadvantaging any particular provider
- They tend to have very clear and complete specifications and well developed conformance testing procedure (b/c they must serve as a basis for compatible implementations.)
- As long as they maintain the specified external behavior, tech suppliers now have more freedom to explore alternative implementations (e.g. to improve reliability, performace, portability, maintainability, etc.) The implementations that conform to the standard should work with both existing and future clients.

Cons of standardization:

- Constrains the range of possible implementations. There is now less room for diversity and evolution.
- If an application uses any feature that is not explicitly authorized by the standard, it stops working. This means that some apps consumers want will not become available sometimes on new releases.
- The people doing the decision making might have many competing opinions about what the new requirements should be and how to best address them.

Technology and applications evolve very quickly. Maintaining stable interfaces in the face of fast and dramatic evolution is extremely difficult. When faced with such change we find ourselves forced to choose between:

- Maintaining strict compatibility with old interfaces, and not supporting new applications and/or platforms. This is (ultimately) the path to obsolescence.
- Developing new interfaces that embrace new technologies and uses, but are incompatible with older interfaces and applications. This choice sacrifices existing customers for the chance to win new ones.
- A compromise that partially supports new technologies and applications, while remaining mostly compatible with old interfaces. This is the path most commonly taken, but it often proves to be both uncompetitive and incompatible.

There is a fundamental conflict between stable interfaces and embracing change.

A **Proprietary** interface is one that is developed and controlled by a single organization (e.g. the Microsoft Windows APIs). An **Open Standard** is one that is developed and controlled by a consortium of providers and/or consumers (e.g. the IETF network protocols).

Application Programming Interfaces (APIs)
A typical API specification is open(2), which includes:

- A list of included routines/methods, and their signatures (parameters, return types)
- A list of associated macros, data types, and data structures
- A discussion of the semantics of each operation, and how it should be used
- A discussion of all of the options, what each does, and how it should be used
- A discussion of return values and possible errors

They describe how the source code should be written in order to exploit these features.

- The benefit for app developers is that an application written to a particular API should easily recompile and correctly execute on any platform that supports that API.
- The benefit for platform suppliers is that any app written to supported APIs should easily port to their platform.

This can raise challenges:

- An API that defined int on a 64-bit machine might not work on a 32-bit machine
- An API that accesses individual bytes within an int might not work on a big-endian machine.
- Some specific feature implementations of an API might not work on some platforms

Application Binary Interfaces (ABIs)
How is it possible to build a binary application that will (with high confidence) run on any Android phone, or any x86 Linux? The problem is not addressed by (source level) APIs. It is addressed by ABIs.
An Application Binary Interface (ABI) is the binding of an Application Programming Interface (API) to an Instruction Set Architecture (ISA).

An 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.
A typical ABI contains things like:

- the binary representation of key data types
- the instruction to call to and return from a subroutine
- stack-frame structure and respective responsibilities of the caller and callee
- how parameters are passed and return values are exchanged
- register conventions (which are used for what, which must be saved and restored)
- the analogous conventions for system calls, and signal deliveries
- the formats of load modules, shared objects, and dynamically loaded libraries

"The portability benefits of an ABI are much greater than those for an API. If an application is written to a supported API, and compiled and linkage edited by ABI-compliant tools, the resulting binary load module should correctly run, unmodified, on any platform that supports that ABI. A program compiled for the x86 architecture on an Intel P6 running Ubuntu can reasonably be expected to execute correctly on an AMD processor running FreeBSD or (on a good day) even Windows. 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. This much stronger portability to a much wider range of platforms makes it practical to build and distribute binary versions of applications to large and diverse markets (like PC or Android users)."

Interface Stability (6/29/2019)

If each part is manufactured and measured to be within its specifications, then any collection of these parts can be assembled into a working whole. This greatly reduces the cost and difficulty of building a working system out of component parts.

The software interface specification is a contract:

- the contract describes an interface and the associated functionality
- implementation providers agree that their systems will conform to the specification
- developers agree to limit their use of the functionality to what is described in the interface specification

If someone comes up with new Posix file semantics and e.g. adds a new required parameter to the Posix open call. This results in:

- software written to the new semantics would have access to richer behavior, and would function better in cases of node and network failures
- software written to the new semantics would not work on the other Posix compliant systems
- software written to the old semantics would not work on the new system
- customers will face inexplicable failures and complain to their software and system providers, even though its not the softwware and system providers' faults
- programmers are confused about which version of open to use and probably end up writing their own I/O packages => time wasted writing the packages and also to have to train people to learn to use it.

Implementations change. Interfaces shouldn't. People providing poor documentation for their code makes it tempting to reverse engineer the interface specifications from a provided implementation. However, this can lead to issues later if they release a new version of the code that was undocumented (and was reverse engineered).

Interfaces also change
You are free to add upward-compatible extensions (old programs will still work the same way, but new interfaces enable new software to access new functionality).

Interface Polymorphism: different method signatures with different parameter and return types. If different versions of a method are readily distinguishable (e.g. by the number and types of their parameters), it may be possible to provide new interfaces to meet new requirements, while continuing to support the older interfaces for backwards compatibility with older applications.

Versioned Interfaces: backward compatibility requirements do not prevent us from changing interfaces; they merely require us to continue providing old interfaces to old clients.

Not all interfaces need to be supported all the time. It just needs to be understood and to achieve the product (or project) goals:

- common to provide alpha/evaluation access to proposed services/interfaces. This helps get quick feedback and evaluate how users react to changes. It is prototyping.
- There may be different types of releases:
  - a micro-release (e.g. 3.4.1) including only bug fixes
  - a minor-release (e.g. 3.5) including new functions, but us upwards compatible with the underlying major release version
  - a major-release (4.0) is allowed to make incompatible changes to previously commited interfaces
- Suppliers might give developers a time-frame for how long they will support the commitments (e.g. we will support release 3.5 for 5 years)

When designing, we need to have high confidence that the external component interfaces will be accomodating all the envisioned evolution while preserving those interfaces.

Object Modules, Linkage Editing, Libraries (7/5/2019)

If we limit our discussion to compiled (vs interpreted) languages, we can typically divide the files that represent programs into a few general classes:

- **source modules:** editable text in some language (e.g. C, assembly, Java) that can be translated into machine language by a compiler or assembler.
- **relocatable object modules:** sets of compiled or assembled instructions created from individual source modules, but which are not yet complete programs.
- **libraries:** collections of object modules, from which we can fetch functions that are required by (and not contained in) the original source/object modules.
- **load modules**: complete programs (usually created by combining numerous object modules) that are ready to be loaded into memory (by the OS) and executed (by the CPU)

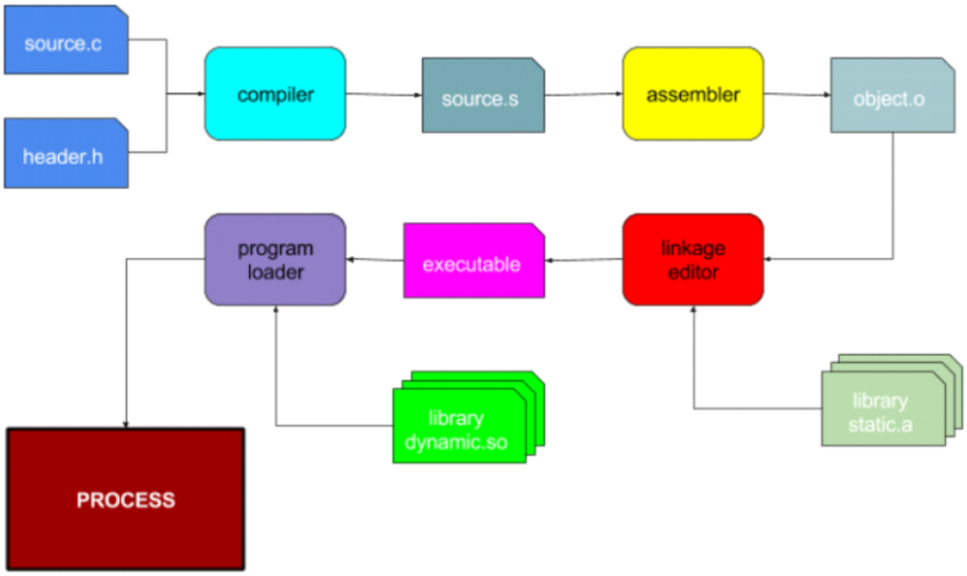


Fig 1. Components of the Software Generation Tool Chain

- **compiler**
  - The job of the compiler is to read source modules and included header files, parses the input language, and figures out the intended computations, for which it will generate lower level code.
  - The lower level code is more often than not assembly, not machine language
    - this provides higher flexibility for further processing (e.g. optimization)
    - makes the compiler more portable and simplifies it by pushing the work to another phase
    - exceptions: Java, Python
- **assembler**
  - assembly language still allows the declaration of variables, the use of macros, and references to externally defined code and data
  - in user-mode code, modules written in assembler are:
    - performances critical string and data structure manipulations
    - routines to implement calls into the OS
  - in the OS:
    - CPU initialization
    - first level trap/interrupt handlers
    - synchronization operations
  - output of assembler is an object module containing mostly machine language code. However,
    - some functions (or data items may not yet be present, and so their addresses will not yet be filled in
    - 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.
- **linkage editor**
  - the job of the linkage editor is to read a specified set of object modules and places them consecutively into a virtual address space, and noting where (in that virtual address space) each was placed.
  - It also notes unresolved external references and searches a specified set of libraries to find object modules that can satisfy those references, and places them in the evolving virtual address space as well.
  - after locating and placing all the required object modules, it finds every reference to a relocatable or external symbol and update it to reflect the address where the desired code/data was actually placed
  - the resulting bits represent a program that is ready to be loaded to memory and executed.
- **program loader**
  - usually part of the OS, it examines in a load module,
  - creates an appropriate virtual address space,

    - and reads the instructions and initialized data values from the load module into that virtual address space
    - 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.

Object Modules

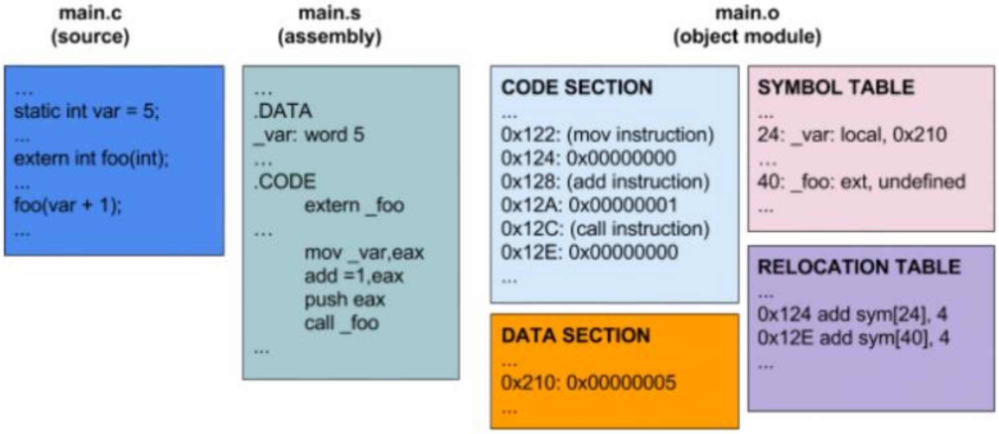
Most programs are created by combining multiple modules together in program fragments called **relocatable object modules** which differ from **executable (load) modules** in at least two interesting respects:

- they may be incomplete, as in they make references to code or data items that must be supplied from other modules
- since they have not yet been combined together into a program, it has not yet been determined (at which addresses) they will be loaded into memory, and so references to code or data items within the same module can have only relative (to the start of the module) addresses.

ELF (Executable and Linkable Format)

divided into multiple consecutive sections:

- a header section, that describes the types, sizes, and locations of the other sections
- Code and Data sections, each containing bytes (of code or data) that are to be loaded (contiguously) into memory.
- A symbol table that lists external symbols defined or needed by this module
- a collection of relocation entries, each of which describes:
    - the location of a field (in a code or data section) that requires relocation
    - the width/type of the field to be to relocated (e.g. 32 or 64 bit address)
    - the sybol table entry, whose address should be used to perform that relocation



```
...
static int var = 5;
...
extern int foo(int);
...
foo(var + 1);
...
```

```
...
.DATA
_var: word 5
...
.CODE
        extern _foo
...
        mov _var,eax
        add =1,eax
        push eax
        call _foo
...
```

```
CODE SECTION
...
0x122: (mov instruction)
0x124: 0x00000000
0x128: (add instruction)
0x12A: 0x00000001
0x12C: (call instruction)
0x12E: 0x00000000
...
```

```
DATA SECTION
...
0x210: 0x00000005
...
```

```
SYMBOL TABLE
...
24: _var: local, 0x210
...
40: _foo: ext, undefined
...
```

```
RELOCATION TABLE
...
0x124 add sym[24], 4
0x12E add sym[40], 4
...
```

Fig 2. A Program in Various Stages of Preparation for Execution

Libraries

Simply a collection of (usually related) object modules.

Building a program usually starts by combining a group of enumerated object modules. The result will almost surely contain unresolved external references (e.g. calls to routines that are to be supplied from libraries). The nex step is to search a list of enumerated libraries to find modules that contain the required functions (can satisfy the unresolved external references).

- Sometimes you can have libraries that are higher level and use functionality from lower level libraries
- You can use alternative implementations for some library functionality or to intercept calls to standard functions to collect usage data.
- Also sometimes the order in which libraries are searched may be very important

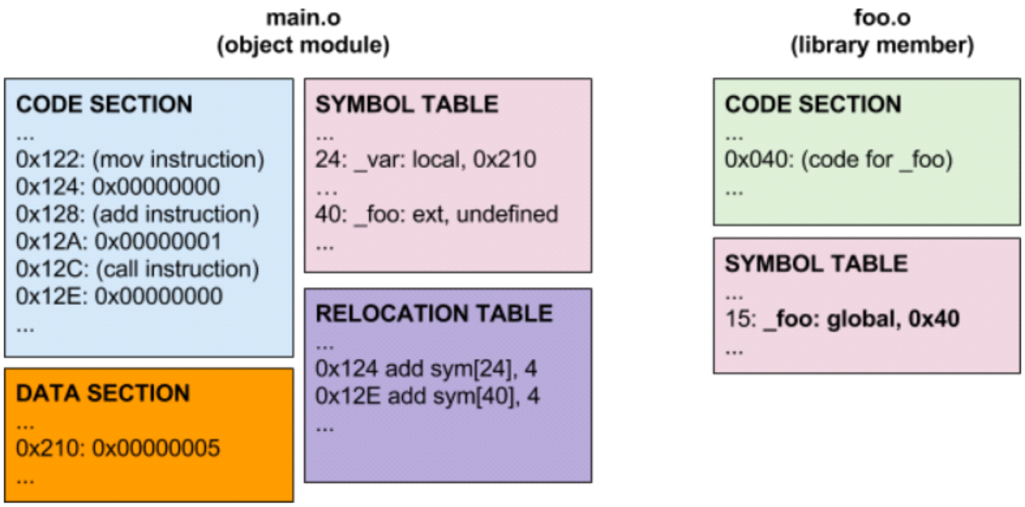
Linkage Editing

Three things need to be done to turn a collection of relocatable object modules into a runnable program:

1. Resolution: search the specified libraries to find object modules that can satisfy all unresolved external references.

2. Loading: 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.
3. Relocation: go through all of the relocation entries in all of the loaded object modules, each reference to correctly reflect the chosen addresses.

   ○ The linkage editor would search the specified libraries for a module that could satisfy the external reference:



```
CODE SECTION
...
0x122: (mov instruction)
0x124: 0x00000000
0x128: (add instruction)
0x12A: 0x00000001
0x12C: (call instruction)
0x12E: 0x00000000
...
```

```
DATA SECTION
...
0x210: 0x00000005
...
```

```
SYMBOL TABLE
...
24: _var: local, 0x210
...
40: _foo: ext, undefined
...
```

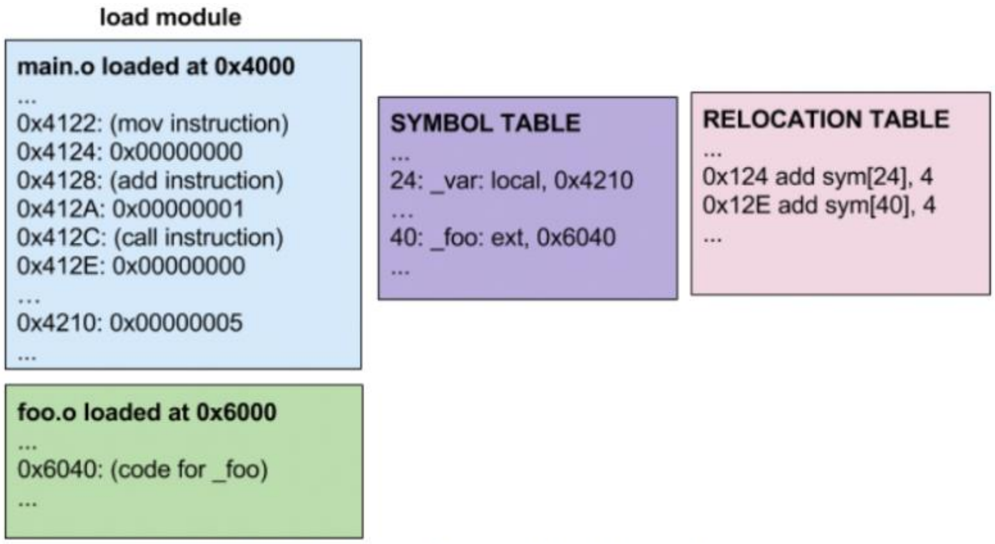
```
RELOCATION TABLE
...
0x124 add sym[24], 4
0x12E add sym[40], 4
...
```

```
CODE SECTION
...
0x040: (code for _foo)
...
```

```
SYMBOL TABLE
...
15: _foo: global, 0x40
...
```

Fig 3. Finding External References in a Library

- Find such a module, the linkage editor would add it to the virtual address space it was accumulating:



```
main.o loaded at 0x4000
...
0x4122: (mov instruction)
0x4124: 0x00000000
0x4128: (add instruction)
0x412A: 0x00000001
0x412C: (call instruction)
0x412E: 0x00000000
...
0x4210: 0x00000005
...
```

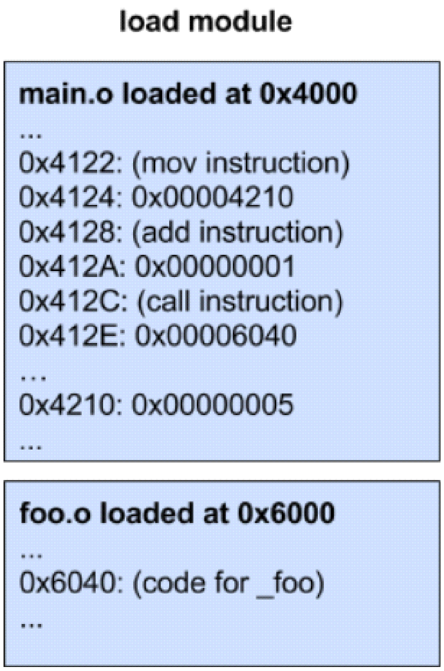
```
foo.o loaded at 0x6000
...
0x6040: (code for _foo)
...
```

```
SYMBOL TABLE
...
24: _var: local, 0x4210
...
40: _foo: ext, 0x6040
...
```

```
RELOCATION TABLE
...
0x124 add sym[24], 4
0x12E add sym[40], 4
...
```

Fig 4. Updating a Process' Virtual Address Space

Then, with all unresolved external references satisfied, and all relocatable addresses fixed, the linkage editor would go back and perform all of the relocations called out in the object modules:



```
main.o loaded at 0x4000
...
0x4122: (mov instruction)
0x4124: 0x00004210
0x4128: (add instruction)
0x412A: 0x00000001
0x412C: (call instruction)
0x412E: 0x00006040
...
0x4210: 0x00000005
...
```

```
foo.o loaded at 0x6000
...
0x6040: (code for _foo)
...
```

Fig 5. Performing a Relocation in an Load Module

At this point, all relocatable addresses have been adjusted to reflect the locations at which they were loaded, and all unresolved external references have been filled in. The program (load module) is now complete and ready to be executed.

Load Modules

Contains multiple sections (code, data, symbol table) and is (a) complete and (b) requires no relocation.
When the OS wants to load a new program, it:

- consults the load module to determine the required text and data segment sizes and locations
- allocates the appropriate segments within the virtual address space
- reads the contents of the text and data segments from the load module into memory
- creates a stack segment, and initializes the stack pointer to point to it

The program is now ready to execute.

Static vs. Shared Libraries

Disadvantages of static libraries:

- Popular libraries are used by most programs so it is not smart to make thousands of identical copies of the same code.
- It increases the required down-load time, disk space, start-up time, and memory
- Popular libraries change over time with newer versions. With static linking, each program is built with a frozen version of each library.

Shared libraries:

- reserve an address for reach shared library
- linkage edit each shared library into a read-only code segment that is loaded at the address reserved for that library
- assign a number (0-n) to each routine, and put a redirection table at the beginning of that shared segment, containing the addresses (to be filled in by the linkaged editor) of each routine in the shared library
- 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.
- linkage edit the client program with the stub library
- when the OS 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.

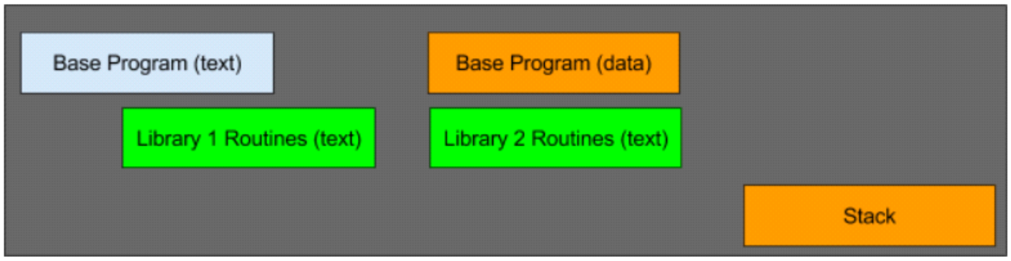


Fig 7. Virtual Address Space with Shared Libraries

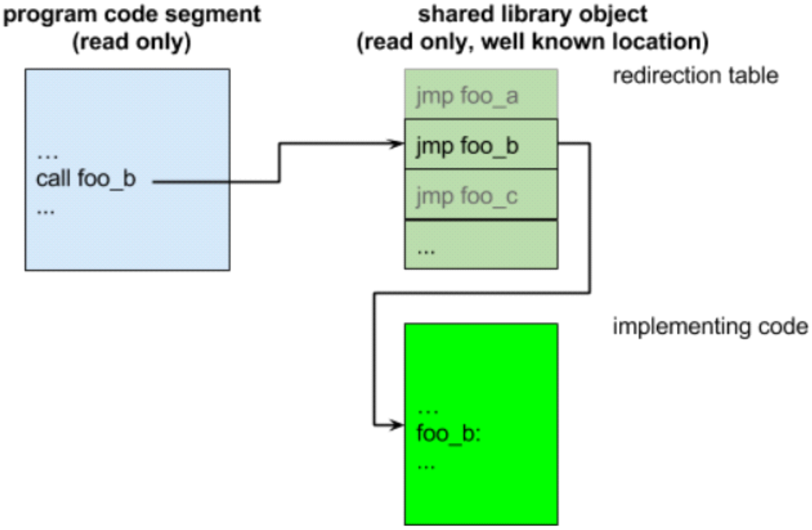


Fig 8. Linking Shared Libraries

In this way:

- a single copy of a shared library implementation can be shared among all the programs that need it
- 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.
- Because all calls between the client program and the shared library are vectored through a table, client programs are not affected by changes in the sizes of library routines or the addition of new modules to the library.
- With correct coding of the stub modules, it is possible for one shared library to make calls into another.

But there are a few important limitations to such an implementation:

- The shared segment contains only read-only code. Routines to be used in this fashion cannot have any static data. Short lived data can be allocated on the stack, but persistent data must be maintained by the client.
- 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.

Dynamically Loaded Libraries

Distadvantages of shared libraries:

- they may be very large/expensive so it might unnecessarily slow down the program start-up and increase memory footprint.
- While loading is delayed until program load time, the name of the library to be loaded must be known at linkage editing time. There are situations (e.g. MIME data types or browser plug-ins) where extensions will be designed and delivered independently from the client that exploits them.

This leads us to dynamically loaded libraries: libraries that are not loaded until they are actually needed.

- ○ the application chooses a library to be loaded
- ○ the application asks the OS to load that library into its address space
- ○ the OS returns addresses of a few standard entry points (e.g. initialization and shut-down)
- ○ the app calls the supplied initialization entry point, and the app and DLL bind to each other
- ○ the app requests services from the DLL by making calls through the dynamically established vector of service entry points
- ○ when the app has no further need of the DLL, it calls the shut-down method and asks the OS to un-load this module.

Stack Frames and Linkage Convention (7/6/2019)

Most programming languages support precedure-local variables:

- they are automatically allocated whenever the procedure (or block is entered).
- they are only visible to code within that procedure (or block). They CANNOT be seen or accessed by code in calling or called procedures.
- Each distinct (e.g. recursive or parallel) invocation of the procedure (or block) has its own distinct set of local variables.
- They are automatically deallocated when the procedure (or block) exits/returns

They are stored in a LIFO (**L**ast **I**n **F**irst **O**ut) **stack.**

The basic elements of subrouting linkage are:

- **parameter passing -** mashaling the information that will be passed as parameters to the called routine.
- **subroutine call -** save the return address (in the calling routine) on a stack, and transfer control to the entry point (of the called routine)
- **register saving -** saving the contents of registers that the linkage conventions declare to be *non-volatile*, so that they can be restored when the called routine returns.
- **allocating space** for the local variables in the called routine

When the subroutine completes:

- **return value -** placing the return value in the place where the calling routine expects to find it.
- **popping the local storage** (for the called routine) off the stack.
- **register restoring -** restore the non-volatile registers to the values they had when the called routine was entered.
- **subroutine return –** transfer control to the return address that the calling routine saved at the beginning of the call.

x86 Register Conventions

- **%esp** is the hardware-defined stack pointer
- the stack grows downwards, so a **push** operation causes the top of the stack to be the next lower address. A **pop** operation causes the top of the stack to be the next higher address.
- **%ebp** is typically used as a **frame pointer -** it points to the start of the current stack frame (where the top of the stack was at the time of entry)
- **%eax** is a volatile register that contains the return value when the called routine returns
- parameters are pushed onto the stack immediately before the caller's return address
- the **call** instruction pushes the address of the next instruction onto the stack, and then transfers control to the next location
- the **ret** instruction pops the retur address off of the top of the stack and transfer back to that location

We observe:

- register saving is the responsibility of the called routine
- cleaning parameters off of the stack is the responsibility of the calling routines
- The clear deliniation of responsbilities between the caller and the callee makes it possible for different language codes to call each other

Traps and Interrupts

interrupts (to inform the software that an external event has happened)
traps (to inform the software of an execution fault)

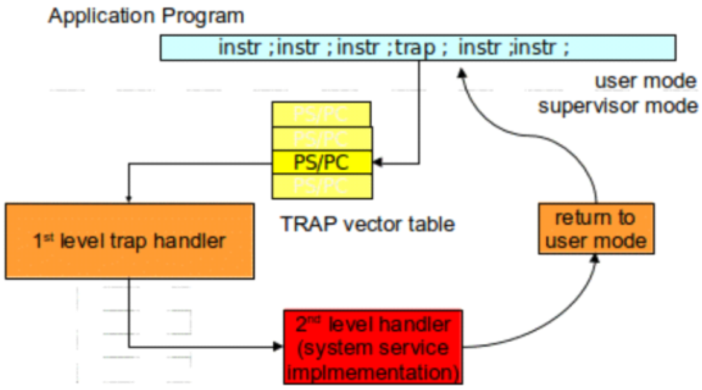
- we want to transfer control to an interrupt or trap handler
- we need to save the state of the running computation before doing so
- after the event has been handled, we want to restore the saved state and resume the interrupted computation

Key differences between a procedure call and an interrupt/trap:

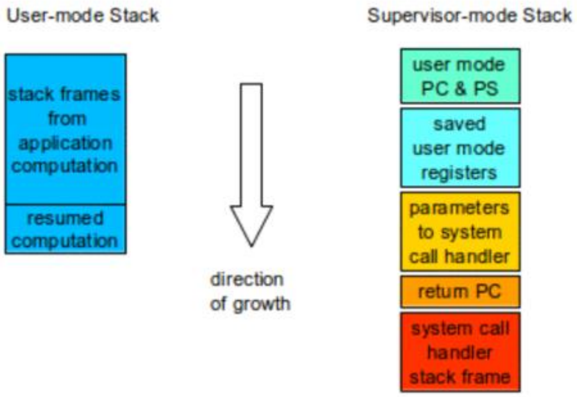
- a procedure call is requested by the running software, and the calling software expects that the procedure called will perform some function and return an appropriate value
- since procedure call is initiated by software, all the linkage conventions are under software control
- since traps/interrupts are initiated by hardware, the linkage conventions are strictly defined by the hardware
- the running software was not expecting a trap or interrupt, and after the event is handled, the computer state should be restored as if the trap/interrupt never happened

A typical interrupt or trap mechanism works as follows:

- here is a number (0, 1, 2 ...) associated with every possible external interrupt or execution exception.
- there is a table, initialized by the OS software, that associates a Program Counter and Processor Status word (PC/PS pair) with each possible external interrupt or execution exception.
- when an event happens that would trigger an interrupt or trap:
    a. the CPU uses the associated interrupt/trap number to index into the appropriate interrupt/trap vector table.
    b. the CPU loads a new program counter and processor status word from the appropriate interrupt/trap vector.
    c. the CPU pushes the program counter and processor status word associated with the interrupted computation onto the CPU stack (associated with the new processor status word).
    d. execution continues at the address specified by the new program counter.
    e. the selected code (usually written in assembler, and called a *first level handler*:
        - saves all of the general registers on the stack
        - gathers information from the hardware on the cause of the interrupt/trap
        - chooses the appropriate second level handler
        - makes a normal procedure call to the second level handler, which actually deals with the interrupt or exception.
- after the second level handler has dealt with the event, it returns to the first level handler, after which ...
    a. the 1st level handler restores all of the saved registers (from the interrupted computation).
    b. the 1st level handler executes a privileged return from interrupt or return from trap instruction.
    c. the CPU re-loads the program counter and processor status word from the values saved at the time of interrupt/trap.
    d. execution resumes at the point of interruption.



Flow-of-control associated with a (system call) trap


User-mode Stack
stack frames from application computation
resumed computation
direction of growth
Supervisor-mode Stack
user mode PC & PS
saved user mode registers
parameters to system call handler
return PC
system call handler stack frame

Stacking and un-stacking of a (system call) trap

This interrupt/trap mechanism:

- does a better job of saving and restoring the state of the interrupted computation
- translates the (much simpler) hardware-driven call to the 1st level handler into a normal higher-level-language procedure call to the chosen 2nd level handler.

The interrupt/trap is somewhere between **100x to 1000x** more expensive than a procedure call.

Summary

Procedure and trap/interrupt linkage instructions provide us with:

- a preliminary low level definition of **the state of a computation** and what it means to **save** and **restore** that state.
- an intro to the basic CPU-supported mechanisms for **synchronous** and **asynchronous** transfers of control.
- an initial example of how it might be possible to **interrupt** an on-going computation, do other things, and then **return** to the interrupted computation as if it had never been interrupted.

Real Time Scheduling

A real-time system is one whose correctness depends on timing as well as functionality.

Real-time metrics:

- *timeliness …* how closely does it meet its timing requirements (e.g. ms/day of accumulated tardiness)
- *predictability …* how much deviation is there in delivered timeliness

New real-time concepts:

- *feasability* - whether or not it is possible to meet the requirements for a particular task set
- *hard real-time* - there are strong requirements that specified tasks be run a specified intervals (or within a specified response time). Failure to meet this requirement may result in system failure (even by a micro-second)
- *soft real-time* - we may want to provide very good (e.g. microseconds) response time, the only consequences of missing a deadline are degraded performance or recoverable failures.

Ways real-time scheduling is easier:

- We may actually know how long each task will take to run. This enables much more intelligent scheduling.
- *Starvation* (of low priority tasks) may be acceptable. Understanding the relative criticality of each task gives us the freedom to intelligently shed less critical work in times of high demand.
- The work-load may be relatively fixed. Normally high utilization implies long queuing delays, as bursty traffic creates long lines. But if the incoming traffic rate is relatively constant, it is possible to simultaneously achieve high utilization and good response time.

Static scheduling

Based on the list of tasks to be run, and the expected completion time for each, we can define (at design or build time) a fixed schedule that will ensure timely execution of all tasks.

Dynamic scheduling

required for real-time scheduling. Two key questions:

1. how they choose the next (ready) task to run
    - shortest job first
    - static priority ... highest priority ready task
    - soonest start-time deadline first (ASAP)
    - soonest completion-time deadline first (slack time)
2. how they handle overload (infeasible requirements)
    - best effort
    - periodicity adjustments ... run lower priority tasks less often.
    - work shedding ... stop running lower priority tasks entirely.

Preemption in ordinary time-sharing vs in real-time

Preemption in ordinary time-sharing systems means improved mean response time and preventing buggy (infinite loop) programs from taking over the CPU.

In real-time:

- preempting a running task will almost surely cause it to miss its

completion deadline
- since we so often know what the expected execution time for a task will be, we can schedule accordingly and should have little need for preemption
- embedded and real-time systems run fewer and simpler tasks than general purpose systems,
- and the code is often much better tested - infinite loop bugs are extremely rare.

Garbage Collection and Defragmentation

http://htmlpreview.github.io/?https://github.com/markkampe/Operating-Systems-Reading/blob/master/gc_defrag.html#GC

1. Garbage collection: seeking out no-longer-in-use resources and recycling them for reuse.
2. Defragmentation: reassigning and relocating previously allocated resources to eliminate *external defragmentation* to create densely allocated resources with large contiguous pools of free space.

Garbage Collection

With many resources, and in many programming languages, the allocated resources are free by an exlicit or implicit action on the client's part. (e.g. close, free, delete, etc)

If a resource is shareable by multiple concurrent clients, we cannot just free it cause one process chose to. It increments the reference count when a new reference to the object is obtained and decrements for vice versa. When ref count is zero, automatically free the object.

Since that may be hard on the developer's end, *Garbage Collection* is a popular alternative to explicit or reference count based freeing.

- Resources are allocated, and never explicitly freed
- when the pool of available resources becomes dangerously small, the system initiates garbage collection:
    1. begin with a list of all of the resources that originally existed.
    2. scan the process to find all resources that are still reachable.
    3. each time a reachable resource is found, remove it from the original resource list.
    4. at the end of the scan, anything that is still in the list of original resouces, is no longer referenced by the process, and can be freed.
    5. after freeing the unused resources, normal program operation can resume.

Defragmentation

How important is contiguous allocation?

- 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.
- 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.
- 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.

Garbage collection analyzes the allocated resources to determine which ones are still in use. Defragmentation goes farther. It actually changes which resource are allocated.

Working Sets

http://htmlpreview.github.io/?https://github.com/markkampe/Operating-Systems-Reading/blob/master/workingsets.html

The LRU works because most programs and processes make use of locality.
If we are doing Round-Robin time-sharing with multiple processes:

- the most recently used pages in memory belong to the process that will not be run for a long time
- the least recently used pages in memory belong to the process that is just about to run again
- this destroys strict temporal and spatial locality, as reference behavior becomes periodic (rather than continuous).

Traditionally, we use Global LRU, but it works poorly as a team with Round-Robin. It might make sense to give each process its own dedicated set of page frames. When that process needed a new page, we would let LRU replace the oldest page in that set. This would be *per-process* LRU, and it works very well with Round-Robin.

Anyway,

- We do not need to give each process as many pages of physical memory as appear in the virtual address space.
- We do not even need to give each process as many pages of its virtual address as it will ever access.

A process experiencing occasional page faults means it will replace old pages with new ones but we'd wanna limit that. As we give a process fewer and fewer page frames in which to operate, the page fault rate rises and our performance gets worse.

Thrashing
The dramatic degradation in system performance associated with not having enough memory to run all of the ready processes.

There is at any given time a number of pages such that:

- if we increase the number of page frames allocated to that process, it makes very little difference in the performance.
- if we reduce the number of page frames allocated to that process, the performance suffers noticeably.
- Called **working-set size**

If a process is experiencing many page faults, its working set is larger than the memory currently allocated to it. If a process is not experiencing page faults, it may have too much memory allocated to it. If we can allocate the right amount of memory to each process, we will minimize our page faults (overhead) and maximize our throughput (efficiency).

Implementing Working Set Replacement
With Global LRU, we saw that a clock-like scan was a very inexpensive way to determine the least recently used pages. The clock hand position was a surrogate for age:

- the most recently examined pages are immediately behind the hand.
- the pages we examined longest ago are immediately in front of the hand.
- 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 not actually the oldest page in memory.

We can use a very similar approach to implement working sets. But we will need to maintain a little bit more information:

- each page frame is associated with an ***owning process***.
- each proces has an ***accumulated CPU time***
- each page frame will have a ***last referenced*** time, value, taken from the ***accumulated CPU*** timer of its ***owning process***.
- we maintain a *target age* parameter … which is ***keep in memory*** goal for all pages

- 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.
- If we find a page that has been referenced since the last scan, we assume it was just referenced.
- If a page is younger than the target age, we do not want to replace it ... since recycling of young

pages indicates we may be thrashing.

- If a page is older than the target age, we take it away from its current owner, and give it to a new (needy) process.

If there are no pages older than the target age, we apparently have too many processes to fit in the available memory:

- If we complete a full scan without finding anything that is older than the target age, we can replace the oldest page in memory. This works, but at the cost of the very expensive complete scan that the clock algorithm was supposed to avoid.
- If we believe that our target age was well chosen (to avoid thrashing) we probably need to reduce the number of processes in memory.

Dynamic Equilibrium algorithm

this is often referred to as a **page stealing** algorithm, because a process that need another page **steals** it from a process that does not seem to need it as much.

- Every process is continuously losing pages that it has not recently refrenced.
- Every process is continuously stealing pages from other processes.
- Processes that reference more pages more often, will accumulate larger working sets.
- Processes that reference fewer pages less often will find their working sets reduced.
- When programs change their behavior, their allocated working sets adjust promptly and automatically.

This results in a dynamic equalibrium mechanism.

User-Mode Thread Implementation

http://htmlpreview.github.io/?https://github.com/markkampe/Operating-Systems-Reading/blob/master/usermodethreads.html

A thread

- is an independently schedulable unit of execution
- runs within the address space of a process
- has access to all of the system resources owned by that process
- has its own general registers
- has its own stack (within the owning process' address space)

- a process is a container for an address space and resources
- a thread is the **unit** of scheduled execution

Basic model:

- Each time a new thread was created:
  - we allocated memory from the heap for a (fixed size) thread-private stack.
  - we create a new thread descriptor that contains identification information, scheduling information, and a pointer to the stack.
  - we add the new thread to the ready queue.
- When a thread calls *yield()* or *sleep()* we save its general registers (on its own stack), and then select the next thread on the ready queue.
- To dispatch a new thread, we simply restore its saved registers (including the stack pointer), and return from the call that caused it to *yield*.
- If a thread called *sleep()* we would remove it from the ready queue. When it was re-awakened, we would put it back onto the ready queue.
- When a thread exited, we would free its stack and thread descriptor.

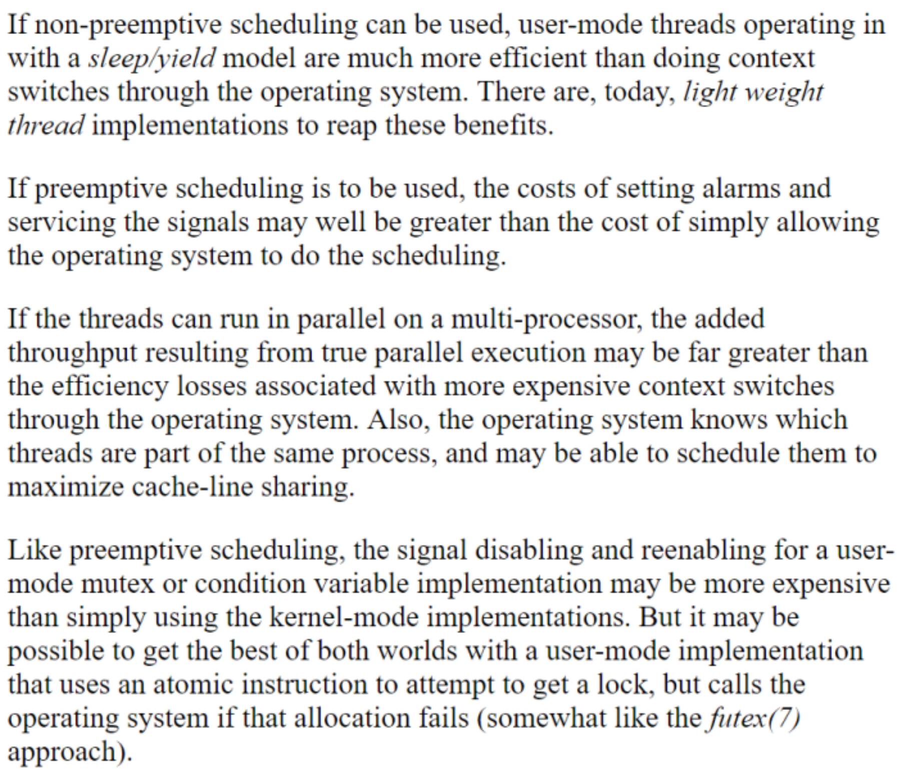
To have preemptive schedulign to prevent buggy threads:

- Linux processes can schedule the delivery of (SIGALARM) timer signals and registers a handler for them.
- Before dispatching a thread, we can schedule a SIGALARM that will interrupt the thread if it runs too long
- If a thread runs too long, the SIGALARM handler can *yield* on behalf of that thread, saving its state, moving on to the next thread in the ready queue.

Problems with implementing threads in a user mode library:

- What happens when a system call blocks
  - If a user-mode thread issues a system call that blocks (e.g. *open* or *read*), the process is blocked until that operation completes. This means that when a thread blocks, all threads (within that process) stop executing. Since the threads were implemented in user-mode, the operating system has no knowledge of them, and cannot know that other threads (in that process) might still be runnable.
- Exploiting multi-processors:
  - If the CPU has multiple execution cores, the operating system can schedule processes on each to run in parallel. But if the operating system is not aware that a process is comprised of multiple threads, those threads cannot execute in parallel on the available cores.

Performance Implications



If non-preemptive scheduling can be used, user-mode threads operating in with a *sleep/yield* model are much more efficient than doing context switches through the operating system. There are, today, *light weight thread* implementations to reap these benefits.

If preemptive scheduling is to be used, the costs of setting alarms and servicing the signals may well be greater than the cost of simply allowing the operating system to do the scheduling.

If the threads can run in parallel on a multi-processor, the added throughput resulting from true parallel execution may be far greater than the efficiency losses associated with more expensive context switches through the operating system. Also, the operating system knows which threads are part of the same process, and may be able to schedule them to maximize cache-line sharing.

Like preemptive scheduling, the signal disabling and reenabling for a user-mode mutex or condition variable implementation may be more expensive than simply using the kernel-mode implementations. But it may be possible to get the best of both worlds with a user-mode implementation that uses an atomic instruction to attempt to get a lock, but calls the operating system if that allocation fails (somewhat like the *futex(7)* approach).

Inter-Process Communication (IPC)

Process Interaction

1. **coordination** of operations with other processes
    a. synchronization (e.g. mutexes and condition variables
    b. the exchange of signals (e.g. kill)
    c. control operations (e.g. fork, wait, ptrace)
2. the **exchange of data** between processes
    a. uni-directional data processing **pipelines**
    b. bi-directional interactions

Uni-directional byte streams: Pipes

A pipe can be opened by a parent and inherited by a child who simply read standard input and write standard output.

Common characteristic of Pipes

- each program accepts a byte-stream input, and produces a byte-stream output
- each program in the pipeline operates independently; programs are not aware of one another
- byte streams are unstructured; if there is any structure to the data they carry, they are implemented by parsers in the affected applications
- ensuring that the output of one program is a suitable input to another is the responsibility of the programmer
- you can also write the output of one program into a temporary file and then using that file as input to the next program. That temporary file is called a **Pipe**.

Pipes recognize the difference between a file and an inter-process data stream

- if there is no more data to be read but there is still a write file descriptor, the reader is blocked until the write size is closed. It just waits for more data instead of getting an EOF

- **flow control**: if the write part of the pipe gets too ahead of the read, the OS blocks the writer until the reader catches up
- if you try to write to a pipe whose write file descriptor is closed, you'll get a **SIGPIPE**
- when the read and write file descriptors are both closed, the file is automatically deleted

Protection
Since the pipeline is a closed system, the only data privacy mechanism are the protection on the initial input files and final output files. There is generally no authentication or encryption.

Named Pipes & Mailboxes
a **named-pipe** is one whose reader and writer can open it by name, rather than using the pipe system call. It can thought of as a "meet-point" for unrelated processes. Some limitations:

- Readers and writers have no way of authenticating one another
- writes from multiple writers can get mixed up without being able to tell who sent what
- they do not enable clean fail-overs from a failed reader to its successor
- all the readers and writers must be on the same node

Mailboxes are designed to combat most of these limitations

- Instead of data being byte-stream, each write is stored and delivered as a message
- there is no ID authentication about its sender
- unprocessed messages remain in the mailbox after a reader's death and can be picked up by another reader
- they still need to be on the same node

General Network API's

- **socket**: creates an inter-process communication end-point with an associated protocol and data model
- **bind**: associate a **socket** with a local network address
- **connect**: establish a connection to a remote network address
- **listen**: wait for an incoming connection request
- **accept:** accept an incoming connection request
- **send**: send a message over a socket
- **recv**: receive a message over a socket

More general networking models allow processes to interact with services all over the world. However, there are limitations:

- making sure the software running on different OS computers with different ISA's can communicate
- dealing with the security issues raised with unknown systems over public networks
- discovering address of ever-changing servers
- detecting and recovering from connection and node failures

High performance for IPC generally means:

- efficiency - low cost per byte transferred
- throughput - maximum number of bytes transferred per second
- latency - minimum delay between sender writer and receiver read

To get ultra high performance IPC between two local processes, buffering the data thru the OS is not the way. Instead, use **shared memory**

- create a file for communication
- each process maps that file into its virtual address space
- the shared segment might be locked down so it never gets paged out
- anything written into the shared memory will immediately become visible to all processes who have it mapped in to their address spaces.

The OS plays no role here.

Downsides:

- a bug in one of the processes can potentially crash the whole communication data structures
- this can only be used between processes on the same memory bus
- there is no authentication of which data came from which process

Named Pipes (FIFOs)

A **named pipe** Works much like a regular pipe. However, there are some differences:

- a **named pipe** exists as a device special file in the system
- processes of different ancestry can share data through a named pipe
- when all I/O is done by sharing processes, the named pipe remains in the file system for later use (i.e. when one reader is out of the game, the named pipe can be picked up by another reader)

FIFO Operations

I/O operations on a FIFO are mostly the same as for normal pipes.

Blocking Actions on a FIFO

Blocking occurs on a FIFO. If a FIFO is opened for reading, the process will "block" until some other process opens it for writing.

The SIGPIPE Signal

If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline.

Deadlock Avoidance:

There are many cases where

- mutual exclusion is fundamental
- hold and block are inevitable
- preemption is unacceptable
- the resource dependency networks are imponderable

Reservation

Declining to grant requests that would put the system into an unsafely resource-depleted state is enough to prevent deadlock. But the failure of a random allocation request (in mid-operation) might be difficult to gracefully handle. For this reason, it is common to ask processes to **reserve** their resources before they actually need them.

- we could refuse to create new files when the file system space gets too low
- we could refuse to create new processes if we found ourselves thrashing due to pressure on main memory
- we could refuse to create or bind sockets when network traffic saturates our service level agreement

Over-Booking

In most situations, it is unlikely that all clients will simultaneously request their maximum resource reservations. For this reason, it is considered *relatively safe* to grant somewhat more reservations that we actually have the resources to fulfill. The reward for over-booking is that we can get more work done with the same resources. The danger is that there is a request we cannot gracefully handle.

Dealing with Rejection

What should a process do when some resource allocation request (e.g. a call to *malloc(3)*) fails?

- A simple program might log an error message and exit.
- A stubborn program might continue retrying the request (in hope tht the problem is transient).
- A more robust program might return errors for requsts that cannot be processed (for want of resources), but continue trying to serve new requests (in the hope that the problem is transient).
- A more civic-minded program might attempt to reduce its resource use (and therefore the number of requests it can serve).

Health Monitoring and Recovery

Suppose that a system seems to have locked up… it has not recently made any progress. How would we determine if the system was deadlocked?

- identify all of the blocked processes
- identify the resource on which each process is blocked
- identify the owner of each blocking resource
- determine whether or not the implied dependency graph contains any loops

How could we know whether or not the systems are making progress?

- by having an internal monitoring agent watch message traffic or a transaction log to determine whether or not work is continuing
- by asking clients to submit failure reports to a central monitoring service when a server appears to have become unresponsive
- by having each server send periodic heart-beat messages to a central health monitoring service
- by having an external health monitoring service send periodic test requests to the service that is being monitored, and ascertain that they are being responded to correctly and in a timely fashion

Short-comings of these approaches:

- heart beat messages can only tell us that the node and application are still up and running. They cannot tell us if the application is actually serving requests.
- clients or an external health monitoring service can determine whether or not the monitored application is responding to requests. But this does not mean that some other requests have not been deadlocked or otherwise wedged.
- an internal monitoring agent might be able to monitor logs or statistics to determine that the service is processing requests at a reasonable rate (and perhaps even that no requests have been waiting too long). But if the internal monitoring agent fails, it may not be able to detect and report errors.

Managed Recovery

Suppose that some or all these monitoring mechanisms determine that a service that hung or failed. What can we do about it? Highly available serivces must be designed for restart, recovery, and fail-over.

- Any process in the system can be killed and restarted at any time. When a process restarts, it should be able to reestablish communication with the other processes and resume working with minimal disruption
- Should be able to support multiple-levels of restart:
    - warm-start -- restore the last saved state and resume service where we left off.
    - cold-start -- ignore any saved state and restart new operations from scratch
    - reset and reboot -- reboot the entire system and then cold-start all of the apps
- Escalating scopes of restarts:
    - restart only a single process, and expect it to resync with the other processes when it comes back up.

    - maintain a list of all of the processes involved in the delivery of a service, and restart all processes in that group.
    - restart all of the software on a single node.
    - restart a group of nodes, or the entire system.

False Reports

Ideally a problem will be found by the internal monitoring agent on the affected node, which automatically triggers a restart of the affected software on that node.
Declaring a process to have failed can potentailly be a very expensive operation. It might cause the cancellation and retransmission of all requests that had been sent to the failed process or node among other problems. We want to be **certain** that a process actually failed.

- the best option would be for a failing system to detect its own problem, inform its partners, and shut-down cleanly.
- if the failure is detected by a missing heart-beat, it may be wise to wait until multiple heart-beat messages have been missed before declaring the process to have failed.
- to distinguish a problem with a monitored system from a problem in the monitoring infrastructure, we might want to wait for multiple other processes/nodes to notice and report the problem.

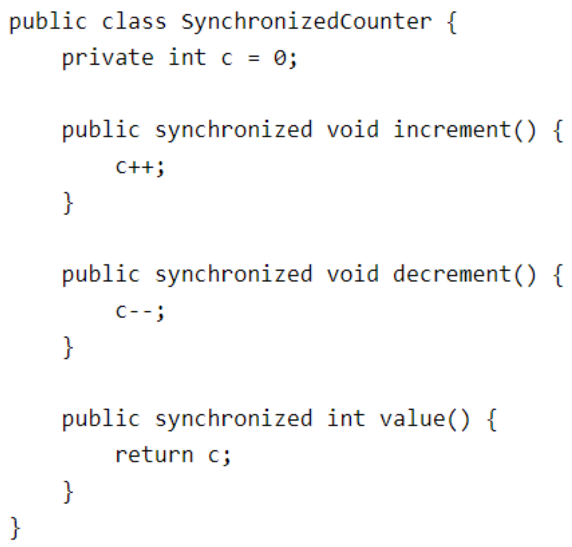
Other Managed Restarts

TL

Java Synchronization

Synchronized methods

To make a method synchronized, simply add the *synchronized* keyword to its declaration



```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the frist thread is done with the object.
- When the synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a syncrhonzied method for the same object.

Constructors CANNOT be synchronized

Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the **intrinsic lock** or **monitor lock**

- every object has an intrinsic lock associated with it
- a thread has to **acquire** the object's intrinsic lock before accessing it and then **release** the intrinsic lock when it's done with it
- a thread **owns** the intrinsic lock between the time it has acquired and released the lock
- other threads trying to acquire that lock are blocked
- when a thread releases a lock, a **happens-before** relationship is established between that action and any subsequent acquisition of the same lock

For a **static method**, the thread acquires the lock for the **Class** object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Synchronized Statements
Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock
tltltltlltltltltltlltl

Monitors (from wikipedia)
a Monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true.
A monitor consists of a mutex object and a condition variable.

a **condition variable** is basically a counter of threads that are waiting for a certain condition.

Another definition of a monitor is a **thread-safe class, object, or module** that wraps around a mutex in order to safely allow access to a method or variable by more than one thread

Consumer/Producer Problem
There is a queue of tasks with a maximum size, with one or more threads being "producer" threads that add tasks to the queue, and one or more threads being "consumer" that take tasks out of the queue.

Whenever the queue is full of tasksm then we need the producer threads to block unitl there is room from consumer threads dequeueing tasks. On the other hand, whenever the queue is empty, then we need the consumer threads to block until more tasks are available due to producer threads adding them.

Condition Variables
A queue of threads, associated with a monitor, on which a thread may wait for some condition to become true. When a thread is waiting on a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state.

Three main conditional variable operations:

1. **wait** c, m: called by a thread that needs to wait until the assertion P_c is true before proceeding
    - c is a condition variable and m is a mutex associated with the monitor.
2. **signal** c: called by a thread to indicate that the assertion P_c is true.
3. **broadcast** c: wakes up all threads in c's wait-queue. It empties the wait-queue.

Performance Measurement

metrics - the numbers we choose to characterize the performance of our systems.
factors - things you intentionally alter in performance experiments to determine which of several alternatives to use.

workloads

1. **traces** - take or obtain a detailed trace of the workload of the system in its ordinary activities.
   - traces are realistic - they represent realistic activities
   - traces are reproduceable - the same trace can be replayed over and over
   - if the system being tested is the one generating the acknowledgements, that can cause the replayed trace to produce unrealistic results
   - not easily reconfigurable
   - not very available
2. **live workloads** - sometimes you can perform measurements on a working system as it goes about its normal activities
   - realistic
   - lack of control
   - not being able to scale loads up and down as desired
   - experimental framework needs to have minimal impact, both in performance and functionality
3. **standard benchmarks** - these are either sets of programs or sets of data that are intended to drive performance experiments, typically on some particular thingm such as a file system, a database, a web server, or an intrusion detection system
4. **simulated workloads** - you build models of the loads you are interested in, typically models instantiated in executable code.
   - can be scaled up or down
   - very customizable to different scenarios and possibilities
   - difficult to produce good models

Common Mistakes in Performance Measurements

1. **measuring latency without considering utilization** - measuring the latency of an operation when absolutely nothing else is going on is poor testing of latency. One should measure the latency when the system has a characteristic background load.
2. **not reporting the variability of measurements** - even taking multiple measurements and taking their average isn't enough. One needs to understand the distribution of those values.
3. **ignoring important special cases** - comes in two varieties
   a. ignoring the fact that a few special cases distort the measurement
   b. while carefully measuring the ordinary case, you fail to consider that there will be some special circumstances that are very important and that are likely to display different performance.
4. **ignoring the costs of your measurement program** - the measurement program itself might deteriorate performance. Find ways to minimize its affect on system you're measuring
5. **losing your data** - never throw away experimental data, even if you think that you are finished with your experiment, nor even if you think the data in question was gathered in an erroneous way.
6. **valuing numbers over wisdom** - the numbers gathered is useless if it's not analyzed to make sense of them.

Load and Stress Testing

A typical suite is a collection of test cases, each of which has the general form:

- establish connections
- perform operations

- ascertain that assertions are satisfied

The art of creating such a test suite is being able to define a set of test cases that simultaneously:

a. adequately execises the program's capabilities
b. adequately captures and verifies the program's behavior
c. is small enough to be practically implementable

The initial and primary **purpose** of load testing is to measure the ability of a system to provide a service at a specified load level. A test harness issues requests to the component under test at a specified rate (the offered load) and collects performance data. The collected performance data includes:

- response time for each request
- aggregate throughput
- CPU time and utilization
- disk I/O operations and utilization
- network packets and utilization

load generators: tools to generate less traffic, corresponding to a specified profile, at a calibrated rate.

In all cases, the test is broadly characterized by:

- **request rate** - the number of operations per second
- **request mix** - diff types of clients use a system in diff ways. E.g. a database might do a large number of disk reads and writes.
- **sequence fidelity** - in many situations, it is sufficient to merely generate the right mix of read and write operations. In other cases, it may be critical to simulate particular access patterns (timed sequences operations on related objects). In some situations it may be necessary to simulate realistic scenarios against predetermined or random objects.

There are a few typical ways to use a load generator for performance assessment:

1. Deliver requests at a specified rate and measure the response time.
2. Deliver requests at increasing rates until a maximum throughput is reached.
3. Deliver requests at a rate, and use this as a calibrated background for measuring the performance other system services.
4. Deliver requests at a rate, and use this as a test load for detailed studies performance bottlenecks.

Load generators are (fundamentally) intended to generate request sequences that are representative of what the measured system will experience in the field. **Accelerate Aging** uses cranked up load generators to simulate longer periods of use. If we go further, we enter the realm of stress testing.

random stress testing

- use randomly generated complex usage scenarios, to increase the likelihood of encountering unlikely combinations of operations
- deliberately generate large numbers of conflicting requests (e.g. multiple clients trying to update the same file)
- introduce a wide range of randomly generated errors, and simulated resource exhaustions, so that the system is continuously experiencing and recovering from errors
- introduce wide swings in load, and regular overload situations

THIS is what gives us serious confidence and the robustness and stability of our systems.

Device Drivers: Classes and Services

https://lasr.cs.ucla.edu/classes/111_fall16/readings/device_drivers.html

Device drivers represent both:

- **generalizing abstractions**: taking all the very different devices and generalizing them into a few general class (e.g. disks, network interfaces, graphics adaptors) and standard models, behaviors and interfaces to be implemented by all drivers for a given class.
- **simplifying abstractions**: providing an implementation of standard class interfaces while **encapsulating** the details of how to effectively/efficiently use a particular device.

Because of the ever-growing number of available devices, there is tremendous demands for object oriented code reuse:

- we need the system to behave similarly regardless of what the underlying devices were being used to provide storage, networking, etc. To ensure this, we want most of the higher level functionality to be implemented in **common, higher level modules**.
- we want to minimize the cost of developing drivers for new devices. The majority of the functionality needs to be implemented in common code inherrited by individual drivers.

we need to derive device-driver sub-classes for each of the major classes of devices
we need to define sub-class specific interfaces to be implemented by the drivers in a sub-class
And create per-device implementations of those sub-class interfaces

Major Driver Classes
Two fundamental classes
Block devices

- Random-access devices, addressable in fixed size (e.g. 4 KB) blocks
- their drivers implement a **request method** to enqueue asynchronous DMA requests.
- the request descriptor includes info about the desired operation, completion info (how much data was transferred, error indications), and a condition variable the requested could use to await the eventual completion of the request.

Character devices

- may be sequential or byte-addressable
- support standard synchronous *read(2), write(2),* and *seem(2)* operations
- for devices that support DMA, read and write operations were expected to be done as a single (potentially very large) DMA transfer between the device and the buffers in user address space.

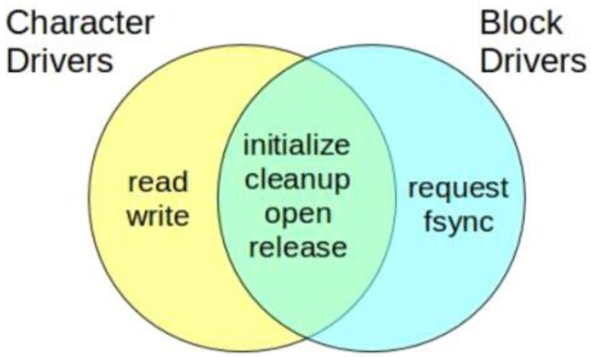
Even in the oldest UNIX systems, device drivers were divided into distinct class based on the needs of the clients:

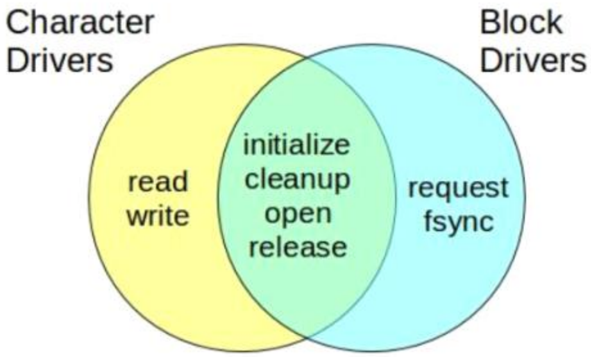
- block devices were designed to be used within the OS by **file system** to **access disks**.
- character devices were designed to be used directly by applications. The large DMA transfer directly between the device and user-space buffers meant that character I/O operations might be much more efficient than requests to a block device.

No mutual exclusion. A single driver could use both block and character interfaces.

A file system would be mounted on top of the block device, while back-up and integrity checking software might access the disk through its character device.

All device drivers supporte *initialize* and *cleanup* methods, *open* and *release* methods, and optional *ioctl* method.

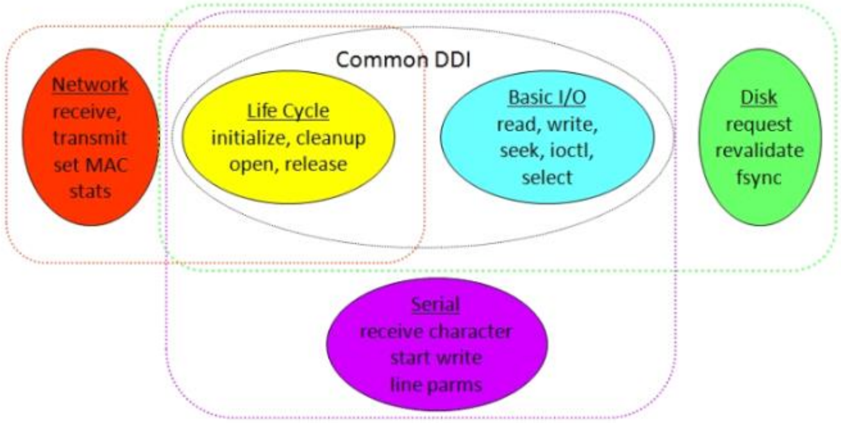




Driver sub-classes

Some examples:

- input editing and outut translation for terminals and virtual terminals
- address binding and packet sending/receipt for network interfaces
- display mapping and window management for graphics adaptors
- character-set mapping for keyboards
- cursor positioning for pointing devices
- sample mixing, volume and equalization for sound devices



Rewards pf this structure are:

- Sub-class drivers become easier to implement because so much of the important functionality is implemented in higher level software.
- The system behaves identically over a wide range of different devices.
- Most functionality enhancements to will be in the higher level code, and so should automatically work on all devices within that sub-class.

Price for these rewards for the device drivers:

- if a driver does not correctly implement the standard interfaces for its device sub-class, it will not work with the higher level software
- if a driver implements additional functionality (not defined in the standard interfaces for its device sub-class), those features will not be exploited by the standard higher level software.

Services for Device Drivers

- dynamic memory allocation
- I/O and bus resource allocation and management
- condition variable operations (wait and signal)
- mutual exclusion
- control of, and being called to service interrupts
- DMA target pin/release, scatter/gather map management
- configuration/registry services

The collection of services, exposed by the OS for use by device drivers is sometimes referred to as the **Driver-Kernel Interface (DKI).** If an OS eliminated ir incompatibly changes a DKI function, device drivers that depend on that function may cease working.

Dynamically Loadable Kernel Modules

Introduction

key elements:

- all implementations provide similar functionality in accordance with a common **interface**.
- the selection of a particular implementation can be deferred until **run-time**.
- decoupling the overarching service from the plug-in implementations make it possible for a system to work with a potentially open-minded set of algorithms or adaptors.

Dynamically Loadable Modules are selected and loaded at run time.

Why use DLM?

- the number of possible I/O devices is far too large to build all of them into the OS
- if we want an OS to automatically work on any computer, it must have the ability to automatically identify and load the required device drivers.
- many devices (e.g. USB) are hot-pluggable, and we cannot know what drivers are required until the associated device is plugged into the computer
- new devices become available long after the OS has been shipped, so must be **after market** addable.
- most device drivers are developed by the hardware manufacturers, and delivered to customers independently from the OS.

Loading a New Module

the module to be loaded may be entirely self-contained (it makes no calls outside of itself) or uses only standard shared libraries (which are expected to be mapped in to the address space at well known locations).
In these cases loading a new module is as simple as allocating some memory (or address space) and reading the new module into it.

In other cases, the DLM may need to make sure of other functions (e.g. memory allocation, synchronization, I/O) in the program into which it is being added. It will result in *unresolved external references* and requires a **run-time loader** that can look up and adjust all of those references as the new module is being loaded.

Initialization and Registration

When the run-time loader is invoked to load a new dynamically loadable module (DLM), it is common for it to return a vector that contains a pointer to an **initialization function**.

The intialization method might:

- allocate memory and initialize driver data structures
- allocate I/O resources and assign them to the devices to be managed
- register all of the device instances it supports

Using a Dynamically Loaded Module

The system often maintains a table of all registered device instances and the associated device driver entry points for each standard operation. Whenever a request is to be made for any device, the OS can simply index into this table by a device identifier to look up the address of the entry point to be called.

Such mechanisms for registering heterogenous implementaitons and forwarding requests to the correct entry point are often referred to as **federation frameworks** because they combine a set of independent implementations into a single unified framework.

Unloading

When all open file descriptors into those devices have been closed and the driver is no-longer needed, the operating system can call an shut-down method that will cause the driver to:

- un-register itself as a device driver
- shut down the devices it had been managing
- return all allocated memory and I/O resources back to the operating system.

After which, the module can be safely unloaded and that memory freed as well.

The Criticalilty of Stable Interfaces

All this is dependent on dependent and well-specified interfaces:

- well defined entry points for any class of device drivers, with all drivers compatibly implementing all of the relevenat interfaces
- the set of functions within the main program (OS) that the DLM calls must be well defined and stable

Hot-Pluggable Devices and Drivers

Hot-plug buses (e.g. USB) can generate events whenever a device is added to or from from the bus. The hot-plug manager:

- subscribes to hot-plug events
- when a new device is inserted, walks the configuration space to identify the new device, finds, and loads the appropriate driver.
- when a device is removed, finds the associated driver and calls its removal method to inform it that the device is no longer on the bus.

Hot-pluggable buses also have multiple power levels, and a newly inserted device may receive only enough power to enable it to be queries and configured.

File Types and Attributes

(week 8 lecture 1)

Ordinary Files

- a **text file** is a byte stream, but when we process it we generally break it into lines (**\n**), and render it as characters.
- an **archive** (e.g. zip or tar) is a single file that contains many others. It is an alternating sequence of headers (that describe the next file in the archive) and data blobs (that are the contents of the described file).
- a **load module** is similar to an archive but the different sections represent different parts of a program (codem initialized data, symbol table, …)
- an **MPEG** stream is a sequence of audio, video, and frames containing compressed program information.

An ordinary file is just a blob of ones and zeros. They only have meaning when rendered by a program that understands the underlying data format.

data types and associated applications

a file can only be understood by a program that understands the meaning that has been encoded in the byte stream. A few general approaches:

- require the user to specifically invoke the correct command to process the data:
  - to edit a file you type vi *filename*
  - to compile a program you type gcc *filename*

- consult a registry that associated a program with a file type:
    - there may be a system-wide registry that associates programs with file types (e.g. Windows).
    - there may be a program-specific registry that associates programs with file types (e.g. configuring browser plug-ins).
    - the owning program may be an attribute of the file (e.g. classic Mac OS).

A registry presupposes that we know what the type of the is. There are a few approaches to **classing** files:

- the simplest approach is based on file name suffix (e.g. .c, .png, .txt).
- **magic number** at the start of the file. Each type of file begins with a reserved and registered magic number that identifies the file's type.
- in systems that support extended attributes the file type can be an attribute of the file

File Structures and Operations
Even ASCII text bytes have structure, and some streams (e.g. MPEG-4 video) have very rich structure In these cases, the file can be viewed as a serialized representation of data that is intended to be viewed by a particular program.

Other types of files
Ordinary files are still just blobs of data, but there are other types of files that are not merely blobs of data to be written and re-read.

Directories
Directories do not contain client data, rather they represent name-spaces (i.e. the association of names with blobs of data).

One very important difference is that a key-value store, as a single file, typically contains data owned by a single user. The namespace implemented by directories includes file owned by numerous users, each of whom wants to impose different sharing/privacy constraints on the access to each referenced file.

They are implemented within the OS can only have a few supported update operations (e.g. **mkdir(2), rmdir(2), link(2), unlink(2), create(2),** and **open(2)**).

Despite these differences, directories exist in same namespace as files, and have the same notions of user/group ownernship and file protection.

Inter-Process Communications Ports
An IPC port is not a container in which data is stored; it is a **channel through which data is passed**. That data is exchanges via write(2) and read(2) system calls on file descriptors that can be manipulated with the dup(2) and close(2) operations.

I/O Devices
I/O devices connect a computer to the outside world. Many OS's put devices in the file namespace. In Unix/Linux systems the special file associated with a device can be anywhere and have any name.

File Attributes
In addition to **data**, files also have a **metadata** - data that describes data.

System Attributes
Unix/Linux files all have a standard set of attributes:

- **type**: regular, directory, pipe, device, symbolic link, etc.
- **ownership**: identity of the owning user and owning group.
- **protection**: permitted access (read/write/execute), by the owner, by owning group, and others.
- **when** the file was created, last updated, and last accessed.
- **size** (for regular files): the number of bytes in the file (which may be a **sparse**).

aside: **sparse file**
a type of file that attempts to use file system space more efficiently when the file itself is partially empty.
adv: storage is only allocated when it is needed: disk space is saved, and large files can be created even if there is insufficiently free space on the file system.
disadv: may become fragmented

Extended Attributes
There may be other information that is important to file processing:

- if the file has been encrypted or compressed, by what algorithm(s)?
- if a file has been signed, what is the associated certificate?
- if a file has been check-summed, what is the correct check-sum?
- if a program has been internationalized, where are its localizations?

^these are all metadata.

Object Storage

(week 8 lecture 1)
object storage is a data storage architecture that manages data as objects.
tltltltltltltltltl

Key-Value Database

a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known as a **dictionary** or **hash table**.

Types
Can use consistency models ranging from **eventual consistency** to **serialization**.

Another example of key-value database is Oracle NoSQL database.
This provides a key-value paradigm to the application developer. Every entity (record) is a set of key-value pairs. A key has multiple components, specified as an ordered list. The **major key** identifies the entity and consists of the leading components of the key. The subsequent components are called **minor keys**.

Filesystem in Userspace (FUSE)

(week 8 lecture 1)

FUSE is a software interface that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while using the FUSE module provides only a "bridge" to actual kernel interfaces.

Uses
to implement a new file system, a handler program linked to the supplied library needs to be written. The program is also used to **mount*** the new file system. At the time the file system is mounted, the handler is registered with the kernel. If a user now issues read/write/stat requests, the kernel forwards these IO-requests to the handler and then sends the handlers' response back to the user.

*mounting is a process by which the OS makes file and directories on a storage device available for users to access via a computer's file system.

The FAT File System

(week 8 lecture 1)

Structural Overview

All file systems include a few basic types of data structures:

- **bootstrap**: code to be loaded into memory and executed when the computer is powered on.
- **volume descriptors**: information describing the size, type, and layout of the file system… and in particular how to find the other key meta-data descriptors.
- **file descriptors**: information that describes a file (ownership, protection, time of last update, etc.) and points where the actual data is stored on the disk.
- **free space descriptors**: lists of blocks of (currently) unused space that can be allocated to files.
- **file name descriptors**: data structures that user-chosen names with each file.

The DOS FAT file systems divide the volume into fixed-sized (physical) blocks, which are grouped into larger fixed-sized (logical) block clusters.

the DOS FAT file system divides the system into fixed-sized blocks, which are grouped into larger fixed-sized block clusters.

The first block of DOS FAT volume constains the bootstrap and some volume info. After comes the much longer **File Allocation Table (FAT)** which is used as

1. a free list
2. to keep track of which blocks have been allocated to which files

The remainder of the volume is data clusters.

Boot block BIOS Parameter Block and FDISK Table

conceptually, the boot record:

- begins with a branch instruction (to the start of the real bootstrap code)
- followed by a volume description (BIOS Parameter Block)
- followed by the real bootstrap code
- followed by an optional partitioning table
- followed by a signature (for error checking)

BIOS Parameter Block

Contains a brief summary of the device and file system. It describes the device geometry:

- number of bytes per (physical) sector
- number of sectors per track
- number of tracks per cylinder
- total number of sectors on the volume

It also describes the way the file system is layed out on the volume:

- number of sectors per (logical) cluster
- the number of reserved sectors (not part of file system)
- the number of Alternative FATs
- the number of entries in the root directory

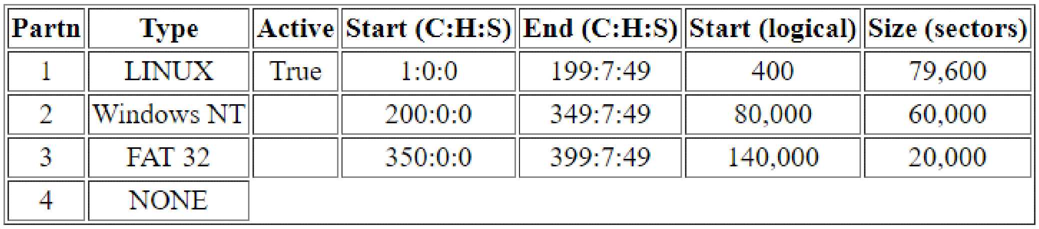
With these, the OS can interpret the remainder of the file system.

FDISK Table

Added because customers wanted to put multiple file systems on each disk. They added a partition table to the end of the bootstrap block.

The FDISK table has four entries:

- a partition type
- an ACTIVE indication
- the disk address where that partition starts and ends
- the number of sectors contained within the partition

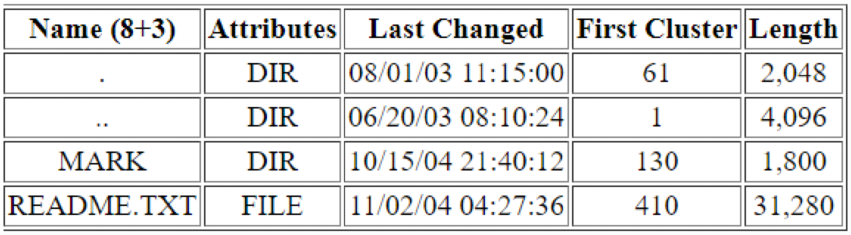


Partn	Type	Active	Start (C:H:S)	End (C:H:S)	Start (logical)	Size (sectors)
1	LINUX	True	1:0:0	199:7:49	400	79,600
2	Windows NT		200:0:0	349:7:49	80,000	60,000
3	FAT 32		350:0:0	399:7:49	140,000	20,000
4	NONE					

File Descriptors (directories)

DOS file systems combine both file description and file naming into a single file descriptor (directory entries). A DOS directory is a file (of a special type) that contains a series of fixed sized (32 byte) directory entries. Each entry describes a single file:

- an 11-byte name
- a byte of attribute bits for the file, which includes
    - is this a file, or sub-directory
    - has this file changed since the last backup
    - is this file hidden
    - is this file read-only
    - is this a system file
    - does this entry describe a volume label
- times and dates of creation and last modification, and date of last access
- a pointer to the first logical block of the file
- the length (# of valid data bytes) in the file



Name (8+3)	Attributes	Last Changed	First Cluster	Length
.	DIR	08/01/03 11:15:00	61	2,048
..	DIR	06/20/03 08:10:24	1	4,096
MARK	DIR	10/15/04 21:40:12	130	1,800
README.TXT	FILE	11/02/04 04:27:36	410	31,280

Links and Free Space (FAT)

Many file systems have very compact (e.g. bitmap) free lists, but most of them use some per-file data structure to keep track of which blocks are allocated to which file. The DOS FAT contains one entry for each logical block in the volume. If a block is free, this is indicated by the FAT entry. If a block is allocated to a file, the FAT entry gives the logical block number of the **next** logical block in the file.

Cluster Size and Performance

Space is allocated to files, not in physical blocks, but in logical multi-block clusters. The number of clusters per block is determined when the file system is created.

Allocating space to files in larger chunks improves I/O performance since you're reducing the number of operations required to read or write a file. This comes at the cost of higher internal fragmentation (since on average half of the last cluster of each file is left unused).

Next Block Pointers

A file's directory entry contains a pointer to the first cluster of that file. The FAT entry for that cluster tells us the cluster number **next** cluster in the file. On the last cluster the FAT entry is −1.

If we had to go to disk to re-read the FAT each time we needed to figure out the next block number, the file system would perform very poorly. Fortunately, the FAT is so small that the entire FAT can be kept in memory as long as a file system is in use.

Free Space
We reserve 0 to mean **this cluster is free**.

Garbage Collection
Starting from the root directory, you find every "valid" entry. You would then follow the chain of next block pointers to determine which clusters were associated with each file, and recursively enumerate the contents of all sub-directories**.** After completing the enumeration of all allocated clusters, they inferred that any cluster not found in some file was free, and marked them as such in the FAT.

Security for Operating Systems

https://ccle.ucla.edu/pluginfile.php/2953729/mod_resource/content/1/Security%20for%20Operating%20Systems.pdf

Three big security-related goals:

- **Confidentiality** - if some piece of information is supposed to be hidden from others, don't allow them to find it out.
- **Integrity** - if some piece of information or component of a system is supposed to be in a particular state, don't allow an adversary to change it. This also requires **authenticity**.
- **Availability** - If some information or service is supposed to be available for your own or others' use, make sure an attacker cannot prevent its use.

Designing Secure Systems

1. **Economy of mechanism** - this basically means keep your system as small as possible. Simple systems have fewer bugs and it's easier to understand their behavior. If you don't understand your system's behavior, you're not likely to know if it's achieving its security goals.
2. **Fail safe defaults** - default to security, not insecurity. If policies can be set to determine the behavior of a system, have the default for those policies be more secure, not less.
3. **Complete mediation** - this is a security term meaning that you should check if an action to be performednmeets security policies every single time the action is taken.
4. **Open design** - assume your adversary knows every detail of your design. If the system can achieve its security goals anyway, you're in good shape. This principle does not necessarily mean that you actually tell everyone all the details, but base your security on the assumption that the attacker has learned everything. He often has, in practice.
5. **Separation of privilege** - require separate parties or credentials to perform critical actions. For example, two-factor authentication, where you use both a password and possession of a piece of hardware to determine identity, is more secure than using either one of those methods alone.
6. **Least privilege** - give a user or a piece the minimum privileges required to perform the actions you wish to allow. The more privileges you give to a party, the greater the danger that they will abuse those privileges. Even if you are confident that the party is not malicious, if they make a mistake, an adversary can leverage their error to use their superfluous privileges in harmful ways.
7. **Least common mechanism** - For different users or processes, use separate data structures or mechanisms to handle them. For example, each process gets its own page table in a virtual memory system, ensuring that one process cannot access another's pages.
8. **Acceptability** - A critical property not dear to the hearts of many programmers. If your users won't use it, your system is worthless. Far too many promising secure systems have been abandoned because they asked too much of their users.

Authentication for Operating Systems

https://ccle.ucla.edu/pluginfile.php/2953730/mod_resource/content/1/Authentication.pdf

We refer to the party asking for something as the **principal**. The process or other active computing entity performing the request on behalf of a principal is often called its **agent**.

Associated with the calling process is the OS-controlled data structure that describes the process, so the OS can check that data structure to determine the identity of the process. Based on that identity, the operating system now has the opportunity to make a policy-based decision on whether to perform the requested operation.

The request is for access to some particular resource, called the **object** of the access request.

Any form of data created and managed by the operating system that keeps track of such access decisions for future reference is often called a **credential**.

Once a principal has been authenticated, systems almost always rely on that authentication decision for at least the lifetime of the process.

Attaching Identities to Processes
We need to use IDs to differentiate between who can access what.

Q. How does a shell or window manager get your identity attached to itself?
A. When a user first starts interacting with a system, the OS can start a process up for him. Since the OS can fiddle with its own data structures, like the process control block, it can set the new process' ownership to the user who just joined the system.

The user logged in, implying that the user provided identity information to the OS proving who he was. Therefore, the OS must be able to query identity from human users and verify that they are who they claim to be, so we can attach reliable identities to processes, so we can use those identities to implement our security policies.

Authentication by What You Know
most commonly done with **passwords**.
No one should know it or they can steal your identity. **Not even the OS.**

Instead of knowing it, the OS stores a has of the password, not the password itself. When the user provides you with wha the claims to be the password, hash his claim and compare it to the stored hashed value.

It's not enough just to store something different from the password; we also want to ensure that whatever we store offers an attacker no help in guessing what the password is.

If an attacker steals the hashed password, he should not be able to analyze the hash to get any clues about the password itself. This is part of a special class of hashing algorithms called **cryptographic hashes** that make it infeasible to use the hash to figure out what the password is.

To prevent guessing, the types of characters and length are important.

Authentication by What You Have
Two-step authentication.
Read pages 9-11 (if there's time)

Authenticaiton by What You Are

Face-recognition: take a picture of the user's face and convert it into 1s and 0s.

When trying to log in, the system cant do an bit-for-bit comparison (as even two pictures taken a second apart are unlikely to have all the same bits).

Instead, it is higher level analysis of the two photos. E.g. length of nose, color of eye, shape of mouth.

False negative - incorrectly deciding not to authenticate the real user
False positive - incorrectly deciding to authenticate a malicious user

Access Control

https://ccle.ucla.edu/pluginfile.php/2953731/mod_resource/content/1/Access%20control.pdf

Two important steps:

1. Figure out if the request fits within our security policy (**access control**)
2. If it does, perform the operation. If not, make sure it isn't done

We will determine which system resources or services can be accessed by which parties in which ways under which circumstances.

Important Aspects of the Access Control Problem
At the high level, access control is usually spoken of in terms of **subjects**, **objects**, and **access**.

- a **subject** is the entity that wants to perform the access, perhaps a user or process
- an **object** is the thing the subject wants to acess, perhaps a file or a device
- **access** is some particular mode of dealing with the object, such as reading it or writing it.

So an access control decision is about whether a particular subject is allowed to perform a particular mode of access on a particular object (sometimes called **authorization**)

We address the issue of the time access control decisions are made, done by an algorithm called a **reference monitor**. Its efficiency is important because it will add overhead whenever it is used.

We can't check after security conditions every time someone asked for something.

One solution is to give subjects objects that belong only to them, objects that is inherently theirs. E.g.

- **virtual memory**: a process is allowed to access its virtual memory freely
- **peripheral devices**: if a process is given access to some virtual device, which is actually backed up by a real physical device controlled by the OS, and if no other process is allowed to use that device, the OS need not check for access control every time the process wants to use it.

Real Example:
Subject X wants to read and write object /tmp/foo.

There are two approaches to solving this question.

- **access control lists** (e.g. list of who can get into the OS nightclub)
- **capabilities** (e.g. a key/ticket to enter the nightclub)

Using ACLs for Access Control

Each file has its own access control list, resulting in simpler, shorter lists and quicker access control checks.

Subject X still trying to get in the nightclub. We now look him up on the list.

- if he's not on the list, no access for him

- if he is on the list, we'll go a step further to determine if the ACL entry for X allows the type of access he's requesting

Where exactly is the ACL peristently stored?
Unless it's cached, it's somewhere on disk, and we need to read it off the disk every time someone tries to open the file.

You need to perform several disk reads to actually obtain any info from a file. Where do we keep it on disk? It needs to be close to something we're already reading, like the file's directory entry, the file's inode, or perhaps the first data block of the file.

The length of the list depends on the file. Some files would only need a couple users, while files like **ls** or **mv** have access control from everyone.

Original ACL Storage for each file
They only could afford 9 bits. They realized that there were effectively three modes of access they cared about (read, write, and execute) and could handle most security policies with only three entries on each access control list.

They partitioned the entries on their access control list into three groups.

- owner of file (whose identity they had already stored away in the inode)
- members of a particular group of users; this group ID could also be stored in the inode
- everybody else; no need to store any bits for this since its just a complement of the user and group

This fixed their problem and also solved the issue of cost of accessing and checking them.

- You already needed to access a file's inode to do almost anything with it, so if the ACL was embedded in the inode, there would be no extra seeks and reads to obtain it.
- And instead of a search of an arbitrary sized list, a bit of simple logic on a few bits would provide the answer to the access control question.

+ ACL

- first, if want to figure out who is allowed to access a resource, you can simply look at the ACL itself
- second, if you want to change the set of subjects who can access an object, you merely need to change the ACL, since nothing else can give the user access.
- third, since the ACL is typically kept either with or near the file itself, if you can get to the file, you get to all relevant access control information.

- ACL

- First, ACLs require you to solve problems we mentioned earlier: having to store the access control information somewhere near the file and dealing with potentially expensive searches of long lists.
- Second, what if you want to figure out the entire set of resources some principal (a process or a user) is permitted to access? You’ll need to check every single ACL in the system, since that principal might be on any of them.
- Third, in a distributed environment, you need to have a common view of identity across all the machines for ACLs to be effective.
  - e.g. if user Ramzi on cs.ucla.edu wants to access a file on cs.wisconsin.edu, is Remzi at UCLA actually referring to the same principal as user Remzi in Wisconsin? If not, you may allow a remote user to access something he shouldn't.

Using Capabilities for Access Control
In capability systems, a running process has some set of capabilities that specify its access permissions.

If you're using a pure capability system, there is no ACL anywhere, and this set is the entire encoding of the access permissions for this process.

What is a capability, exactly?
Capabilities are bunches of bits, data.

Processes and users should not be allowed to touch any capabilities. The OS controls and maintains capabilities, storing them somewhere in its protected memory space. So if we want to rely on capabilities for access control, the operating system will need to maintain its own protected capability list for each process.

There is another option. Capabilities need not be stored in the operating system. Instead, they can be cryptographically protected. If capabilities are relatively long and are created with strong cryptography, they cannot be guessed in a practical way and can be left in the user's hands.

With capabilities, it's easy to determine which system resources a given principal can access. Just look through his capability list. Revoking his access merely requires removing the capability from the list, which is easy enough if the operating system has exclusive access to the capability

On the other hand, determining the entire set of principals who can access a resource becomes more expensive. Any principal might have a capability for the resource, so you must check all principals' capability lists to tell

In practice, user-visible access control mechanisms tend to use access control lists, not capabilities. Instead, e.g. in UNIX, they use a combination of the two to get the best of both worlds.

Mandatory and Discretionary Access Control

Who should decide what the access control on a computer resource should be? Usually, the owner of the resource, but not always.

The common case is called **discretionary access control**, where the permissions are set by the owner.

The more restrictive case is called **mandatory access control**, where at least some elements of the access control divisions in such systems are mandated by an authority, who can override the desires of the owner of the information.

Practicalities of Access Control Mechanisms

RBAC

Cryptography

https://ccle.ucla.edu/pluginfile.php/2953732/mod_resource/content/1/Cryptography.pdf

Cryptography is a set of techniques to convert data from one form to another, in controlled ways with expected outcomes.

The basic idea behind it is to take a piece of data and use an algorithm (often called a **cipher**), usually augmented with a second piece of information (which is called a **key**), to convert the data into a different form.

We start with data P, a key K, and an encryption algorithm E(). We end up with C, called **ciphertext**.
C = E(P,K)

The reverse transformation takes C, which we just produced, a decryption algorithm D(), and the key K:

P=D(C,K)

In many cases, E() and D() are actually the same algorithm, but that is not required. Also, it should be very hard to figure out P from C without knowing K.

We only get assurance if the opponent does not know both D() and our key K. If he does, he'll apply D() and K to C and extract the same information P that we can.

THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY

If you're using a strong ci[her and are careful about maintaining key secrecy, your cryptography is strong.

Symmetric cryptography - using the same key to encrypt and decrypt.

Public Key Cryptography

What if we're Microsoft, and we want to authenticate ourselves to every user who has purchased our software? We can't use just one key to do this, because we'd need to send that key to hundreds of millions of users and, once they had that key, they could pretend to be Microsoft by using it to encrypt information

This is where we use two different keys for cryptography, one to encrypt and one to decrypt.

$C = E(P, K_{encrypt})$

Decryption:
$P = D(C, K_{decrypt})$

Microsoft can now tell everyone their decryption key but keep their encryption key secret.

This form of cryptography is called **public key cryptography (PK)**, since one of the two keys can be widely known to the entire public. The key everyone knows is called the public key, and the key that only the owner knows is called the private key.

The public key must be distributed, but generally we don't care if some third party learns this key, since they can't use it to sign messages.

It works the other way around: you can use the decryption key $K_{decrypt}$ to encrypt, and the encryption key $K_{encrypt}$ to decrypt. Thus, PK allows authentication if you encrypt with the private key and secret communication if you encrypt with the public key.

You can actually have both, but you'll need two different key pairs to do that.

E.g. Let's say Alice wants to use PK to communicate secretly with her pal Bob, and also wants to be sure Bob can authenticate her messages. Let's also say Alice and Bob each have their own PK pair. Each of them knows his or her own private key and the other party's public key. If Alice encrypts her message with her own private key, she'll authenticate the message, since Bob can use her public key to decrypt and will know that only Alice could have created that message. But everyone knows Alice's public key, so there would be no secrecy achieved. However, if Alice takes the authenticated message and encrypts it a second time, this time with Bob's public key, she will achieve secrecy as well. Only Bob knows the matching private key, so only Bob can read the message. Of course, Bob will need to decrypt twice, once with his private key and then a second time with Alice's public key.

This is v computationally expensive and much harder than traditional symmetric cryptography.

Cryptographic Hashes

To ensure integrity, we use **cryptographic hashes**, with properties:

- it is computationally infeasible to find two inputs that will produce the same hash value
- any change to an input will result in an unpredictable change to the resulting hash value
- it is computationally infeasible to infer any properties of the input based only on the hash value

Based on these properties, if we only care about data integrity, rather than secrecy, we can take the cryptographic hash of a piece of data, encrypt only that hash, and send both the encrypted hash and the data to our partner. If an opponent fiddles with the data in transit, when we decrypt the hash and repeat the hashing operation on the data, we'll see a mismatch and detect the tampering.

Without encrypting the cryptographic hash, the opponent can simply change the message, compute a new hash, replace both the original message and the original hash with his own versions, and send the results. If the hash we sent is encrypted though, he can't know what the encrypted version of the altered hash should be.

To formalize, to perform a cryptographic hash we take a plaintext P and a hashing algorithm H().

$$S = H(P)$$

Cracking Cryptography

Hackers cannot read data encrypted with these algorithms without obtaining the key, but they can still exploit software flaws in your system having nothing to do with the cryptography

At-Rest Data Encryption

What if someone can get access to some of our hardware without going through our operating system? If the data is stored on that hardware is encrypted, and the key isn't on that hardware itself, the cryptography will protect the data. This is **at-rest data encryption**.

The data can be encrypted in different ways, using different ciphers (DES, AES, Blowfish), at different granularities (records, data blocks, individual files, entire file systems), by different system components (applications, libraries, file systems, device drivers).

One common general use of at-rest data encryption is called **full disk encryption**. This usually means that the entire contents of the storage device are encrypted. Full disk encryption is usually provided either in hardware (built into the storage device) or by system software (a device driver or some element of a file system).

Things this encryption method **doesn't** protect against:

- **It offers no extra protection against users trying to access data they should not be allowed to see**. Either the standard access control mechanisms that the operating system provides work (and such users can't get to the data because they lack access permissions) or they don't (in which case such users will be given equal use of the decryption key as anyone else).
- **It does not protect against flaws in applications that divulge data**. Such flaws will permit attackers to pose as the user, so if the user can access the unencrypted data, so can the attacker. So, for example, it offers little protection in the face of buffer overflow or SQL injection attacks.
- **It does not protect against dishonest privileged users on the system, such as a system administrator**. If his privileges allow him to pose as the user who owns the data or to install system components that give him access to the user's data, he will be given

decrypted copies of the data on request.
- **It does not protect against security flaws in the operating system itself**. Once the key is provided, it is available (directly in memory, or indirectly by asking the hardware to use it) to the operating system, whether that OS is trustworthy and secure or compromised and insecure.

But, consider this.
If a hardware device storing data is physically moved form one machine to another, the OS on the other machine is not obligated to honor the access control info stored on the device. This benefit would be useful if the hardware in question was stolen and moved to another machine, for example. (e.g. stolen phone)

- Archiving data that might need to be copied and must be preserved, but need not be used. In this case, the data can be encrypted at the time of its creation, and perhaps never decrypted, or only decrypted under special circumstances under the control of the data's owner. If the machine was uncompromised when the data was first encrypted and the key is not permanently stored on the system, the encrypted data is fairly safe. (Note, however, that if the key is not recoverable from some source, you will never be able to decrypt the archived data.)
- Storing sensitive data in a cloud computing facility, a variant of the previous example. If one does not completely trust the cloud computing provider (or one is uncertain of how careful that provider is ¾ remember, when you trust another computing element, you're trusting not only its honesty, but also its carefulness and correctness), encrypting the data before sending it to the cloud facility is wise. Many cloud backup products include this capability. In this case, the cryptography and key use occur before moving the data to the untrusted system, or after it is recovered from that system.
- User-level encryption performed through an application. For example, a user might choose to encrypt an email message, with any stored version of it being in encrypted form. In this case, the cryptography will be performed by the application, and the user will do something to make a cryptographic key available to the application. Ideally, that application will ensure that the unencrypted form of the data and the key used to encrypt it are no longer readily available after encryption is completed. Remember, however, that while the key exists, the operating system can obtain access to it without your application knowing.

The password is forgotten when the user logs out, or the system shuts down, or the application that handles the password vault (e.g. a Chrome) exits.

Cryptographic Capabilities

Capabilities cannot be left in the user's hands, since the users could forge them and grant themselves access to anything they wanted. With cryptography, we can create unforgeable capabilities.

A trusted entity could use cryptography to create a sufficiently long and securely encrypted data structure that indicated that the possessor was allowed to have access to a particular resource. This data structure could then be given to a user, who would present it to the owner of the matching resource to obtain access.

These cryptographic capabilities can be created with either symmetric or public key cryptography
- with **symmetric cryptography**, both the creator of the capability and the system checking it need to share the same key.
    - most feasible when both of those entities are the same system
    - symmetric cryptographic capabilities also make sense when all of the machines creating and checking them are inherently trusted and key distribution is not

problematic

- if **public key cryptography** is used to create the capabilities, then the creator and the resource controller need not be co-located and the trust relationships not be as strong.
  - The creator of the capability needs one key (typically the secret key) and the controller of the resource needs the other
  - if the contents of the capabilities is not itself secret, then a true public key can be used, with no concern over who knows it
  - if secrecy is required, what would otherwise be a public key can be distributed only to the limited set of entities that would need to check the capabilities.

Distributed Systems Goals & Challenges

http://htmlpreview.github.io/?https://github.com/markkampe/Operating-Systems-Reading/blob/master/distsystems.html

Reliablity and Availibility

Combining multiple (very ordinary systems) can obtain better reliability than building systems out of the best possible components. We can even continue providing service, even after one or more of our servers have failed.

The key is to distribute over multiple **independent** servers. The reason they must be independent is so that they have no *single point of failure* - no single component whose failure would take out multiple systems.

Scalability

It is more practical to design systems that can be expanded incrementally, by adding additional computers and storage as they are needed (than to keep buying more powerful computers every time we want to upgrade).

Flexibility

Distributed systems tend to be very flexible in this respect.

Challenges of Distributed Systems

- In a single system it may be very easy to tell that one of the service processes has died. In distributed systems our only indication that a component thas failed might be that we are no longer receiving messages from it. Perhaps:
  - it has failed
  - it is low
  - the network link has failed
  - our own network interface has failed
- If we expect a distributed system to continue operating despite the failures of individual components, all of the components need to be made more robust (e.g. greater error checking, automatic fail-over, recovery and connection reestablishment.

Complexity of Distributed Systems

- Distinct nodes in a distributed system operate completely independently of one-another. Unless operations are performed by message exchanges, it is generally not possible to say whether a particular operation on node X happened before or after a different operation on node Y.
- Because of the independence of parallel events, different nodes may at any given instant, consider a resource to be in different states. Thus a resource does not actually have a single state. Rather its state is a vector of the state that the resource is considered to be in by each node in the system.

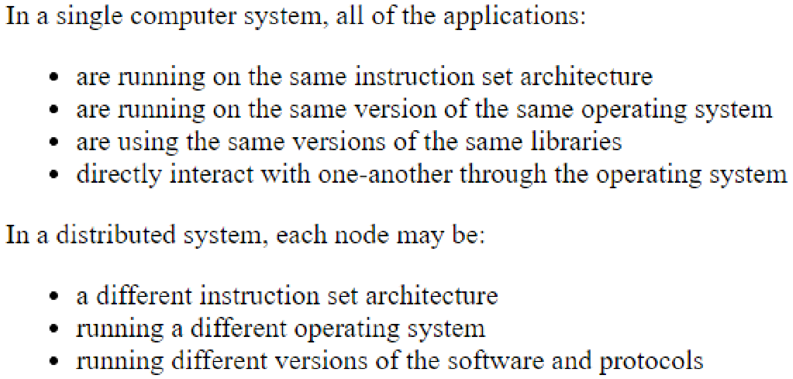
Complexity of Management

- some nodes may not be up when the updates are sent, and so not learn of them
- networking problems may create isolated islands of nodes that are not operating with a different configuration

Much Higher Loads

One of the reasons we build distributed systems is to handle increasing loads. Higher loads often uncover weaknesses that had never caused problems under lighter loads. When a load increases by more than a power of ten, it is common to discover new bottlenecks. Longer (and more variable) delays turn up race-conditions that had previously been highly unlikely.

Heterogeneity



In a single computer system, all of the applications:

- are running on the same instruction set architecture
- are running on the same version of the same operating system
- are using the same versions of the same libraries
- directly interact with one-another through the operating system

In a distributed system, each node may be:

- a different instruction set architecture
- running a different operating system
- running different versions of the software and protocols

"Seven Fallacies" of Distributed Computing

1. The network is reliable.
   Subroutine calls always happen. Messages and responses are not guaranteed to be delivered.
2. Latency is zero.
   The time to make a subroutine call is negligible. The time for a a message exchange can easily be 1,000,000X greater.
3. Bandwidth is Infinite.
   In-memory data copies can be performed at phenomenal rates. Network throughput is limited, and large numbers of clients can easily saturate NICs, switches and WAN links.
4. The network is secure.
   While not perfect, operating systems are sufficiently well protected that we (relatively) seldom have to worry about malicious attacks within our own computer. Once we put a computer on a network it becomes susceptable to penetration attempts, man-in-the-middle attacks, and denial-of-service attacks.
5. Topology does not change.
   In a distributed system, routes change and new clients/servers appear and disappear continuously. Distributed applications must be able to deal with an ever-changing set of connections to an ever-changing set of partners.
6. There is one administrator.
   There may not be a single database of all known clients. Different systems may be administered with different privileges. Independently managed routers and firewalls may block some messages to/from some clients.
7. Transport cost is zero.
   Network infrastructure is not free, and the capital and operational costs of equipment and channels to transport all of our data can dramatically increase the cost of a proposed service.

And, not in Deutsch's original list, but added shortly thereafter ...

8. The network is homogenous.
   Nodes on the network are running different versions, of different operating systems, on machines with different Instruction Set Architectures, word lengths, and byte orders, whose users speak different languages, use different character sets, and have very different means of representing even very standard information (e.g. dates).

Representational State Transfer

https://en.wikipedia.org/wiki/Representational_state_transfer

Representational State Transfer (**REST**) is a software architectural style that defines a set of constraints to be used for creating Web services.

Web services that conform to the REST architectural style, called ***RESTful*** Web services (RWS), provide interoperability between computer systems on the Internet. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations.

In a RESTful Web service, requests made to a resource's URI will elicit a response with a payload formatted in HTML, XML, JSON, or some other format. The response can confirm that some alteration has been made to the stored resource, and the response can provide hypertext links to other related resources or collections of resources.

By using a **stateless protocol** and standard operations, RESTful systems aim for fast performance, reliability, and the ability to grow by reusing components that can be managed and updated without affecting the system as a whole, even while it is running.

Architectural Constraints

Following these guiding constraints, the system gains desirable non-functional properties, such as:

- performance
- scalability
- simplicity
- modifiability
- visibility
- portability
- reliability

1. **Client-server architecture**: Separation of concerns.
    - Separating the user interface concerns from the data storage concern improves the portability of the user interfaces across multiple platforms.
    - It also improves scalability by simplifying the server components.
2. **Statelessness**: the client-server communication is constrained by no client context stored on the server between requests.
    - Each request from any client contains all the information necessary to service the request, and the session state is helt in the client.
    - The session sdtate can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication.
    - The client begins sending requests when it is ready to make the transition to a new state.
    - While one or more requests are outstanding, the client is considered to be **in transition**.
3. **Cacheability**:
    - As on the World Wide Web, clients and intermediaries can cache responses.
    - Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from getting stale or inappropriate data in response to further requests.
    - Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance.
4. **Layered System**:
    - A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.
    - This means that the client doesn’t know if it’s talking with an intermediate or the

actual server.
    - So if a proxy or load balancer is placed between the client and server, it wouldn't affect their communications and there wouldn't be necessities to update the client or server code.
    - Intermediary servers can improve system scalability by enabling load balancing and by providing shared caches.
5. **Code on demand (optional)**:
    - Servers can temporarily extend or customize the functionality of a client by transferring executable code.
6. **Uniform Interface**:
    - This simplifies and decouples the architecture, which enables each part to evolve independently. It has four constraints of its own:
        - **Resource identification in requests**: Individual resources are identified in requests, for example using URIs in RESTful Web services. The resources themselves are conceptually separate from the representations that are returned to the client.
        - **Resource manipulation through representations**: When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.
        - **Self-descriptive messages**: Each message includes enough information to describe how to process the message.
        - **Hypermedia as the engine of application state**: Having accessed an initial URI for the REST application—analogous to a human Web user accessing the home page of a website—a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other actions that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the application

Aside:
a **stateless protocol** is a communication protocol in which no session information is retained by the receiver, usually a server. Relevant session data is sent to the receiver by the client in such a way that every packet of info transferred can be understood in isolation, without context information from previous packets in the session.
This property makes them ideal in high volume applications, increasing performance by removing server load caused by retention of session information.

Lease Based Serialization

http://htmlpreview.github.io/?https://github.com/markkampe/Operating-Systems-Reading/blob/master/leases.html

Locking operations in distributed systems run afoul of all of these:

- In a single node, a compare-and-swap mutex operation might take many tens of nanoseconds. Obtaining a lock through message exchange will likely take at least tens of milliseconds. That is a minimum one-million-X difference in performance.
- In a single node, a mutex operation (whether implemented with atomic instructions or system calls) is guaranteed to complete (tho perhaps unsuccessfully). In a distributed system the request or response could be lost.
- When a single node crashes, it takes all of its applications down with it, and when they restart, all locks will be re-acquired. In a distributed system the node holding the lock can crash without releasing it, and all the other actors will hang indefinitely waiting for a release that will never happen.
- If a process dies, the OS knows it, and has the possibility of automatically releasing all locks held by that process. If the OS dies, all lock-holding processes will also die, and nobody will have to cope with the fact that the OS no longer knows who holds what locks.

> When a node dies in a distributed system, there is no meta-OS to observe the failure and perform the cleanup. If the failed node happens to be the lock-manager, the remaining clients may find that their locks have been "forgotten".

Distributed consensus and multi-phase commits are extremely complex processes, probably far to expensive to be used for every locking operation. It is much easier and more efficient to simply send all locking requests (as messages) to a central server, who will implement them with (simple, efficient, reliable) local locks.

We can replace **locks** with **leases**.

- A ***lock*** grants the owner exclusive access to a resource until the owner releases it.
- A ***lease*** grants the owner exclusive access to a resource until the owner releases it or the lease duration expires

In principle, for normal operation a lease works the same as a lock. Someone who wants exclusive access to a resource requests the lease. As soon as the resource becomes available, the lease is granted, and the requestor can use the resource. When the requestor is done, the lease is released and available for a new owner.

Difference: When a request is sent to a remote server to perform some operation (e.g. update a record in a database), that request includes a copy of the requestor's lease. If the lease has expired, the responding resource manager will refuse to accept the request. It does not matter why a lease may have expired.

Whatever the reason for the expiration, the lease is no longer valid, operations from the previous owner will no longer be accepted, and the lease can be granted to a new requestor. This means that the system can automatically recover from any failure of the lease-holder (including deadlock).

However, there are two challenges:

- An expired lease prevents the previous owner from performing any further operations on the protected resource.
  - But if the tardy owner was part-way through a multi-update transaction, the object may be left in an inconsistent state.
  - If an object is subject to such inconsistencies, updates should be made in all-or-none transactions (i.e**. atomically**)
  - If a lease expires before the operation is committed, the resource should fall back to its last consistent state.
- The lease period is important:
  - too short, and the owner may have to renew it many times to complete a session.
  - too long, and it may take the systema long time to recover from a failure.

Evaluating Leases

- **mutual exclusion**: as good as locks, w/ additional benefit of potential enforcement.
- **fairness**: depends on the policies implemented by the remote lock manager, who could easily implement queued or prioritized requests.
- **performance**: If lease requests are rare and cover large numbers of operations, this can be a very efficient mechanism.
- **progress**: no deadlocks, but if a lease-holder dies, others would be have wait for the lease period to expire.
- **robustness**: more robust than those single-system mechanisms.
- **leases are not without their problems**:
  - while they easily recover from client failures, correct failure recovery from lock-server failures is extremely complex.
  - raises the issue of how to decide "what time it is" in a distributed system without a universal time standard.

Opportunistic Locks

The CIFS protocol supports ***opportunistic locks***. A requestor can ask for a long term lease, enabling that node to handle all future locking as a purely local operation. If another node requests access to the resource, the lock manager will notify the op-lock owner that the lease has been revoked, and subsequent locking operations will have to be dealt with through the centralized lock manager.

Consensus

https://en.wikipedia.org/wiki/Consensus_(computer_science)

A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This often requires processes to agree on some data value that is needed during computation.

The consensus problem requires agreement among a number of processes (or agents) for a single data value. Some of the processes (agents) may fail or be unreliable in other ways, so consensus protocols must be fault tolerant or resilient.

One approach to generating consensus is for all processes (agents) to agree on a majority value.

Protocols that solve consensus problems are designed to deal with limited numbers of faulty processes.

A consensus protocol tolerating halting failures must satisfy the following properties.[1]

- **Termination** - Eventually, every correct process decides some value.
- **Integrity** - If all the correct processes proposed the same value v, then any correct process must decide v.
- **Agreement** - Every correct process must agree on the same value.

In evaluating the performance of consensus protocols two factors of interest are:

- **running time**
- **message complexity**

Crash and Byzantine failures

- a **crash failure** occurs when a process abruptly stops and does not resume.
- **Byzantine failures** are failures in which absolutely no conditions are imposed (e.g. as a result of the malicious actions of an adversary).

Distributed System Security

https://ccle.ucla.edu/pluginfile.php/2953741/mod_resource/content/1/Distributed%20system%20security.pdf

When authenticating a user over a network (someone on a remote OS), you may

- either require the remote machine to provide you with a password
- or you require it to provide evidence using a private key stored only on that machine.

Passwords tend to be useful if there are a vast number of parties who need to authenticate themselves to one party. Public keys tend to be useful if there's one party who needs to authenticate himself to a vast number of parties.

With a password, the authentication provides evidence that somebody knows a password. With a public key, many parties can know the key, but only one party who knows the matching private key can authenticate himself.

- When a website authenticates itself to a user, it's done with PK cryptography. By distributing one single public key (to a vast number of users), the website can be autheticated by all its users.
- When a user authenticates itself to a website, it's done with a pssword.

Each user must be separately authenticated to the website, so we require a unique piece of identifying information for that user, preferably something that's easy for a person to use.

Public Key Authentication for Distributed Systems

We can use the fact that secrecy isn't required to simply create a bunch of bits containing the public key. Anyone who gets a copy of the bits has the key.

What if some other trusted party known to everyone who needs to authenticate our partner used their own publc key to cryptographically sign that bunch of bits, verifying that they do indeed belong to our partner?

If we could check that signature, we could then be sure that bunch of bits really does represent our partner’s public key, at least to the extent that we trust that third party who did the signature.

This is how we actually authenticate websites and many other entities on the internet.

The signed bundle of bits is called a **certificate**, and it contains information about the party that owns the public key. The entire set of information, including the public key, is run through a cryptographic hash, and the result is encrypted with the trusted third party's private key, digitally signing the certificate.

Example:
Frobazz Inc wants to obtain a certificate for its public key K_F.
Go on page 4 to read the rest of the example.

There are some wonderful properties about the above approach to learning public keys:

- first, note that the signing authority did not need to participate in the process of the customer checking the certificate
- second, it only needs to be done once per customer
- third, the customer had no need to trust the party claimining to be Frobazz until that identity had been proven by checking the certificate.

Password Authentication for Distributed Systems

This will work best in situations where only two parties need to deal with any particular password: the party being authenticated and the authenticating party.

SSL/TLS