Operating Systems Principles

1. Introduction

We build increasingly large amounts of increasingly complex software. Size and complexity create numerous problems:

- increasingly sophisticated software means that the typical project involves multiple orders of magnitude more code than projects of 20-30 years ago.
- increasing complexity means that the work involved in creating a piece of software is much worse than linear with the number of lines of code involved.
- increasing complexity makes it very difficult to learn and fully understand our software, which leads to many more errors in its construction.
- systems are now assembled from numerous independently developed and delivered pieces, and small changes in one component often result in failures of other components.
- as the number of independent interacting components increases, we begin to see complex emergent behaviors that could not have been predicted from our experience with smaller systems.
- increasingly rich functionality and interactions with increasingly many other external systems make complete testing combinatorically impossible.
- our increasing dependency on this software means that failures will cause more severe problems, for more people, more often.

We need tools and techniques to tame this complexity. Without such tools, we cannot deliver today's software ... much less tomorrow's.

Operating systems have long been among the largest and complex pieces of software:

- They involve complex interactions among many sub-systems.
- They involve numerous asynchronous interactions and externally originated events.
- They involve the sharing of stateful resources among cooperating parallel processes.
- They involve coordinated actions among heterogenous components in large distributed systems.
- They must evolve to meet ever-changing requirements.
- They must be able to run, without error, for years, despite the regular failures of all components.
- They must be portable to virtually all computer architectures and able to function in heterogenous environments (with many different versions of many different implementations, on many different platforms).

Few of these problems are unique to operating systems, but operating systems encountered them sooner and more intensely than other software domains. As a result, these problems have come to be best understood and solutions developed in the context of Operating Systems. When evolving disciplines encounter these problems, they commonly go back to see how they have been characterized and solved within operating systems.

2. Complexity Management Principles

Software Engineering has developed numerous tools to help us manage the complexity of large software projects. Many of these are software development and project management techniques, but there are also

several fundamental design principles.

2.1 Layered Structure and Hierarchical Decomposition

A complex system can easily have too many components and interactions to be held in a single mind, at a single time. To make the whole system (or even individual components) understandable, we must break the system down into smaller components *that can be understood individually.*

Hierarchical decomposition is the process of decomposing a system in a top-down fashion. If we wanted to describe the University of California, we might

- Start by dividing it into the Regents, the Office of the President, and the campuses.
- The campuses can be divided into ten university campuses (UCLA, UCSB, ...) and three laboratories (Livermore, Los Alamos, ...)
- One campus (e.g. UCLA) can be divided into administration and schools (Letters and Sciences, Engineering, Medicine, ...)
- Administration can be divided into departments (personnel, legal, finance, registrar, deans, ...)
- Each School (e.g. Engineering) can also be divided into departments (Computer Science, Materials, ...)
- A department can be divided into administrative staff, support staff, and faculty
- Administrative staff can be divided into responsibility areas (payroll, purchasing, personnel, ...) ...

At each level we can talk about the mission of each group and interactions with other groups. In this way we are understanding a huge University system one layer at a time. We can also pick a particular group and look at (decompose it into) the sub-groups that comprise it. In this way we can develop an understanding of that one group, without having to understand the internal structure of all the groups with which it interacts. Both of these approaches give us the opportunity to develop (some) understanding of the University in small installments.

Hierarchical decomposition is not merely a technique for studying existing systems, but also a principle for designing new systems. If every person in any part of the University system interacted regularly with every other person in every other part, the processes would be impossible to understand or manage. But if we can devise a structure where, at each level, there is a clear division of responsibilities among components, and most functions can be carried out within a particular component, the responsibilities and relationships are much simplified, and it becomes possible to understand, and to manage the system.

Hierarchical structure is a very effective approach to designing or understanding complex systems. But the operation of these systems is not always strictly hierarchical. If we drill down into the details, we see that not all interaction is constrained to follow the tree. Course planning involves interactions between departmental administration and the registrar's office. Different departments or schools often work together to create programs and events in areas of shared interest. The Academic Senate is comprised of department faculty, but their decisions may have implications for the entire system. Such cross-communication is common in hierarchically structured systems. But, as we will see below, there may be advantages to trying to minimize non-hierarchical communication.

This principle has been described in terms of a University system. But it is highly applicable to complex software systems. In this course we will study and observe the hierarchical design of a complex modern operating system.

2.2 Modularity and Functional Encapsulation

Taking a few million people and arbitrarily assigning them to groups of ten, and then arbitrarily assigning

those groups into super-groups of one hundred ... would not be sufficient to make the system understandable or manageable. We must also require that:

- Each group/component (at each level) have a coherent purpose.
- Most functions (for which a particular group is responsible) can be performed entirely within that group/component.
- The union of the groups/component (within each super-group/system) is able to achieve the system's purpose.

These requirements are the difference between an arbitrary decomposition and a hierarchical decomposition. Such a decomposition makes it possible to opaquely encapsulate the internal operation of a sub-group/component, and to view it only in terms of its external interfaces (the services it performs for other groups/components). This enables us to look a group in two very different ways:

- We can examine its responsibilities, and its role in the system of which it is a part.
- We can examine the internal structure and operating rules by which it fulfills its responsibilities.

As long as a component is able to effectively fulfill its responsibilities, external managers and clients can ignore the internal structure and operating rules. Moreover, it should be possible to change the internal structure and operating rules without impacting external clients and managers. When it is possible to understand and use the functions of a component without understanding its internal structure, we call that component a *module* and say that its implementation details are *encapsulated* within that module. When a system is designed so that all of its components have this characteristic, we say that the system design is *modular*.

We can go a little further in defining the characteristics of good modularity:

- There are numerous ways to break up the responsibilities of a system. Smaller and simpler components are easier to understand and manage than larger and more complex components. Thus, there are clear benefits to dividing a system into a larger number of smaller and simpler components. Going back to the university example, even though the University has over a thousand departments and a hundred-thousand employees, it is not particularly difficult to understand the role and responsibilities of any particular department or administrator.

- Many functions are closely related, typically because they perform related operations on the same resources. If operations on a single resource (e.g. deposits and withdrawals in a bank account) are implemented in separate components, there is a danger that a change in one component might (through unintended *side effects*) break another component. Thus it is a good idea to combine closely related functionality (operations embodying common understandings, or with a high likelihood of affecting one-another) into a single module. Combining this with the previous characteristic: we want the smallest possible modules consistent with the co-location of closely related functionality. This characteristic of modular design is called *cohesion*, and a module that exhibits this characteristic is said to be *cohesive*.

- There are numerous ways to apportion the functionality of a system among its components. If most operations can be accomplished entirely within a single component, they are likely to be accomplished more efficiently. If many operations involve exchanges of services between components:
    - the overhead of communication between components may reduce system efficiency.
    - the increased number of interfaces to service those inter-component requests increases the complexity of the system and the opportunity for misunderstandings.
    - increased dependencies between components increase the likelihood of errors or misunderstandings.

Thus, the scope of component responsibilities must be designed with an eye towards how well it will be possible to compartmentalize operations within that component. This is another type of *cohesion*. As observed above, there are surely many interactions between various administrators, but most of the day-to-day operations and communications for the personnel administrator in the UCLA Computer Science department are with other people in that department.

In this course, we will observe the way in which a modern operating system is structured and how responsibility is apportioned among sub-systems and plug-in components.

2.3 Appropriately Abstracted Interfaces and Information Hiding

There are numerous ways to specify the interface for each piece of component functionality. They are not all equally good. It seems natural (for an implementer) to define interfaces that mate easily with an intended implementation. A faucet, for instance, delivers water at a rate and temperature that is controlled by a chosen mixture of hot and cold water. So it is that, for many years, most faucets have had two valves (for hot and cold water). While this interface is very easy for plumbers to hook up, it is not a particularly good one:

- I, as a user, want a certain flow of water at a certain temperature. I do not want to guess at the (changing over time) amounts of hot and cold water that will provide it for me. An interface that enables a client to specify parameters that are most 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*.

- The two-valve interface is tightly coupled to a hot- and cold-water supply implementation. If water temperature was controlled by a local flash-heater (or flash-cooler), a two-valve set of controls would be both awkward and inefficient to implement. An appropriately abstracted interface that does not reveal the underlying implementation is said to be *opaque* and to exhibit good *information hiding* (not exposing implementation details that clients did not want to be forced to understand).

Two separate controls for temperature and flow rate would more *opaquely encapsulate* the water delivery mechanisms. Changing the temperature control from a simple clockwise-hot knob to a temperature-calibrated dial would create a much more convenient control abstraction for the intended uses. Such an interface better serves the clients, and provides greater flexibility to the implementers.

Obviously a better abstracted interface is easier for the intended clients. But even if the internal implementation was well suited to the intended clients, exposing implementation details would:

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 incompatible interface.

In this course, we will examine many fundamental abstractions of software services and resources, how well those interfaces serve their clients and the need for evolution.

2.4 Powerful Abstractions

If every problem has to be solved from scratch, every problem will be hard to solve, and progress will be slow and difficult. But if we can draw on a library of powerful tools, we may find that most problems are easily solved by existing tools. Most of classical mechanics is based on a few basic machines (wheel, lever, inclined plane, screw, pulley, ...) and tools (hammer, chisel, saw, drill, ...). These are not merely artifacts we

can employ to build new inventions; They actually set the terms in which we visualize our problems. A new technique or tool (e.g. constructive fabrication with 3D-printers) can radically alter the way we approach problems and (perhaps more importantly) the problems we can solve.

Unlike basic machines and tools (which we can observe in nature), virtually all of the tools and resources in an operating system are abstract concepts from the imaginations of system architects. As operating systems have evolved, we continue to devise more powerful abstractions. A powerful abstraction is one that can be profitably applied to many situations:

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

Sufficiently powerful abstractions give us tools to understand, organize and control systems which would otherwise be intractably complex. In this course we will study the emergent phenomena in large and complex systems, concepts in terms of which those phenomena can be understood, and abstractions in terms of which they can be managed.

2.5 Interface Contracts

A complex system is composed of many interacting components. Hierarchical decomposition and modular information hiding moves our attention from implementations to interfaces. Every system, sub-system, or component is primarily understood in terms of the services it provides and the interfaces through which it provides them. An interface specification is a contract: a promise to deliver specific results, in specific forms, in response to requests in specific forms. If the service-providing component operates according to the interface specification, and the clients ensure that all use is within the scope of the interface specifications, then the system should work, no matter what changes are made to the individual component implementations.

These contracts do not merely specify what we will do today, but they represent a commitment for what will be done in the future. This is particularly important in large, complex systems like an operating system. An operating system will be used for a relatively long time, and is assembled from thousands of components developed by thousands of people, most of whom are operating independently from one-another. Technologies and requirements are continuously evolving and component implementations change continuously in response. If a new version of a service-providing component does not fully comply with its interface specifications (e.g. we decide to change the order of the parameters to the *open(2)* system call), or a new client uses a component in an out-of-spec way (e.g. exploiting an undocumented feature), problems are likely to arise, in the future, perhaps resulting in a system failure.

It both logistically and combinatorically infeasible for every developer to test every change with every combination of components (and component versions) from every other developer. Interface contracts are our first line of defense against incompatible changes. As long as component evolution continues to honor long-established interface contracts, the system should continue to function despite the continuous independent evolution of its constituent components.

In this course will examine a variety of interfaces, interface standards and approaches for managing upwards compatible evolution.

2.6 Progressive Refinement

Linux did not start out as the fully-featured, all-platform system that it currently is. It started out as an attempt to complement the Gnu tools with a very humble re-implementation of the most basic UNIX kernel functionality. All of the wonderful features we enjoy today were added incrementally. Software projects that attempt to to grand things, starting from a clean slate, often fail after spending many years without delivering useful results.

- It is very difficult to estimate the work in a large project.
- It is very difficult to anticipate the problems in a large project.
- It is very difficult to get the requirements right in a large project.
- They often take so long that they are obsolete before they are finished.
- They often lose support before they are finished.

Religious wars abound in Software Engineering, but most modern methodologies now seem to embrace some form of iterative or incremental development:

- Add new functionality in smaller, one feature at a time, projects. It is much easier to identify likely problems (and solutions) in a smaller project, and hence to estimate the required work and likely delivery date. Also the work can be done by a smaller team with better communication, and so is likely to be done more efficiently.
- Rather than pursue large speculative (if you build it they will come) projects, identify specific users with specific needs, and build something that addresses those needs (while moving us in the right direction). Having a specific problem to solve leads to better requirements. Delivering needed functionality to actual users yields creates more value more quickly.
- Deliver new functionality as quickly as possible to get feedback before moving on to the next step. Most software requirements are speculative. The best way to clarify them is to deliver something and get feedback.
- It is much easier to plan the next step after we have data from the previous step. Real performance data or user feedback will guide us towards the most important improvements. If we are going down the wrong path, it is best that we find that out as soon as possible.

3. Architectural Paradigms

Over many decades of building ever more powerful operating systems we have found a few very powerful concepts of very broad applicability.

3.1 Mechanism/Policy Separation

The basic concept is that the mechanisms for managing resources should not dictate or greatly constrain the policies according to which those resources are used. The primary motivation for this principle is that a system is likely to be used used for a long time, in many places, environments and ways that the designers never anticipated. A single resource management strategy that made sense to the designer might prove totally inappropriate for future users. As such resource managers should be designed 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.

And, because the policy is independent from the mechanisms, we could also move to a very different mechanism implementation (e.g. RFID tags), and continue to support the exact same access policies. It should be possible to change either mechanism or policy without greatly impacting the other.

Mechanism/policy separation as a design principle that guides us to build systems that will be usable in a wide range of future contexts. In this course we will look at a few examples of services whose designs exhibit this quality.

3.2 Indirection, Federation, and Deferred Binding

Powerful unifying abstractions often give rise to general object classes with multiple implementations. Consider a file system: a collection of files that can be opened, read and written. There can be many different types of file systems (e.g. DOS/FAT file systems on SD cards, ISO 9660 file systems on CD ROMs, Windows NTFS disks, Linux EXT3 disks, etc), but they all implement similar functionality. One could write an operating system that understood all possible file system formats, but the number and range file system formats and rate of evolution dooms that approach to failure. Realistically, we need a way to add support for new file systems independently from the operating system to which they will be connected.

Many services (e.g. device drivers, video codecs, browser plug-ins) have similar needs. The most common way to accommodate multiple implementations of similar functionality is 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 operating system, but accessed *indirectly* (e.g. through a pointer to an 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 operating system until a client requests it.

3.3 Dynamic Equilibrium

It is relatively easy to design and tune a system for a particular load. But most systems will be used in a wide range of different conditions, and their loads (both intensity and mix of activities) change over time. This means that there is probably not a single, one-size-fits all configuration for a complex system. This is widely recognized, and most complex systems have numerous tunable parameters that can be used to optimize behavior for a given load. Unfortunately, a good set of tunable parameters is not enough to ensure that

systems are well tuned:

- 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.

Nature is full of very complex systems, very few of which are perfectly tuned. 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:

- The amount of water in a sealed pot is the net result of evaporation and condensation. When the temperature is raised, more evaporation takes place, which leads to an increase in the steam vapor pressure, which leads to an increased rate of re-condensation.
- The deer population may be the net result of food supply and predation by wolves. If the food supply increases, an increase in the deer population will lead to an increase in the wolf population, which will reduce the deer population.

If our resource allocations can be driven by opposing forces, we may be able to design systems that continuously and automatically adapt to a wide range of conditions. In this course we will examine multiple uses of dynamic equilibrium to achieve stable operation in the face of ever-changing loads.

3.4 The Criticality of Data Structures

Given the amount of code we write, it is natural for programmers to focus much attention on algorithms and their implementation. But the solution to many hard software design problems is found, not so much in algorithms, as in data organization. Most of our program state is stored in data structures, and it is often the case that most of our instruction cycles are spent searching, copying and updating data structures. Consequently, our data structure designs (and their correctness/coherency requirements) determine:

- which operations are fast and which are slow (e.g. because of the number of pointers to be followed)
- which operations are simple and which are complex (e.g. because of how many different data structures have to be updated)
- the locking requirements for each operation (and hence the achievable parallelism)
- the speed (and success probability) of error recovery (because of the number of possible errors and their detectability/repairability)

Often, when confronted with a difficult performance, robustness, or correctness problem, the solution is found in the right data structures. And given those data structures (and their correctness/coherency assertions) the algorithms often become obvious. In this course we will examine the data structures that underlie much complex functionality, and see how the right data structures have contributed to system performance and robustness.

4. Life Principles

Life is made up of huge projects involving a myriad of complex, parallel activities among numerous independent agents (e.g. growing up, getting and education, getting married, finding and keeping a job, raising children, ...). Thus it should not be surprising if many of the lessons we learn from operating systems turn out to be applicable to many other aspects of our lives (and vice versa).

4.1 There Ain't No Such Thing As A Free Lunch

There are no Magic Ponies or perpetual motion machines. Everything comes at a cost. There are common situations with simple and obvious solutions, but those solutions don't correctly handle all situations. We often have multiple conflicting goals, and an approach that optimizes one goal usually compromises another. There is almost always a downside.

Any solution we formulate is likely to involve costs, trade-offs and compromises. Before we can safely change an important system (e.g. an operating system), we must be able to predict (though research, analysis, or prototyping) the likely consequences. The we estimate their impacts, prioritize our goals, and try to optimize our net expected utility.

4.2 The Devil is Usually in the Details

Every once in a while we find a great solution: some beautiful set of paradigms and principles that eliminate nasty problems and greatly simplify the system. But our first articulation of that vision is seldom right, and it must be refined as we try to understand how we will apply it in all of the required situations:

- Enumerating (or even identifying) all of the interesting cases is a great deal of work.
- In many cases, it may not be obvious how to make a particular situation fit in to our beautiful high level structure.
- Sometimes we encounter one nasty case that simply refuses to fit into the new model.
- Sometimes we can extend our model to embrace the troublesome cases.
- Sometimes we can sub-case, preclude or exclude the troublesome cases.
- Sometimes we conclude that, while beautiful, our idea doesn't work.

Having an inspiration is fun, exciting, and rewarding. But it isn't real until you have worked out all of the details. (Ask any String Theorist.)

4.3 Keep It Simple, Stupid and If it ain't broke, don't fix it

Einstein was once quoted as saying:

> *Everything should be made as simple as possible, but no simpler.*

As should be clear from this paper, complexity is the enemy. The problems we face are complex enough, but sometimes we voluntarily add gratuitous complexity to our solutions. It is natural to look at a solution and recognize enhancements that would make it smarter, or more complete, or more elegant. These insights are fun, but they often prove to be counter-productive:

- Many optimizations require extra work to better handle special cases, and that extra work may greatly reduce the benefits of the optimization.
- Many ideas that seem simple at first become more complex as we examine all of the cases that have to be correctly handled.
- More complex solutions are likely to have more complex behavior, including unanticipated consequences. If we cannot predict the behavior associated with changes, we are likely to find that we

have created new problems.
- More complex solutions are more difficult to understand, and hence more likely to have undetected errors and more difficult to maintain.

There are surely problems that call for very smart solutions, but it is often the case that a simpler solution works better than a more clever solution. It is best to avoid more complex solutions unless we have hard data that simpler solutions are not viable and that the smarter solution would fix the problem.

4.4 Be Clear about your Goals

We occasionally build software for fun, but usually we do so with a purpose in mind. It has often been said that if we do not clearly understand our goals, it will only be by accident that we achieve them.

As we pursue a project we will be confronted by many decisions. Some questions are of the form "will this work", and can usually be answered by research, analysis or prototyping. But we will also encounter "which is better" questions. Some of these will have objective (e.g. measurable performance) answers, but some will come down to priorities and perspectives. A clear understanding (and prioritization) of our goals can help us with these questions, by allowing us to ask how each of our alternatives would impact each of our goals.

As we pursue a project we will see many problems and opportunities. We are in a creative problem solving mode, and it is natural to automatically leap to approaches to address problems and exploit opportunities. But not every opportunity we encounter is on the path to our ultimate goals, and not every problem we encounter is actually blocking the path to our ultimate goals. If we expand the scope of our effort to solving non-critical problems or achieving non-goals, we may find that we have over-constrained our requirements ... making it much more difficult to achieve our goals.

It is also important to keep in mind the high level goals, from which our detailed goals may have been derived. Our detailed goals might be to speed up some operation and eliminate a confusing parameter from some interface. But the real goal was to make the system faster and easier to use. If we myopically focus only on our micro-goals, without periodically assessing ourselves against the higher level goals, we may find that we have won all the battles, but lost the war.

Having a clear understanding of our goals will enable us to resolve differences of opinion, make better decisions, and avoid distractions.

4.5 Responsibility and Sustainability

We are going to live for a relatively long time, and when we are gone, our children will live in the world we left them. We need to understand the long-view consequences of our actions, how they will eventually effect us and others, and then act responsibly for the best long-run outcome.

The same principles apply to operating systems. No software-managed resource can ever be lost. All memory and secondary storage will be recycled over and over again. Errors cannot be dismissed as unlikely events, and no error can be left un-handled. Every reasonably anticipatable problem must be correctly detected and handled, so that the system can continue executing. It is commonly observed that the main differences between professional and amateur software are completeness and error handling. We are all responsible for anticipating and dealing with all the consequences of our actions. OS designers are not, by nature, optimists.

5. Conclusions

Despite the fact that Operating Systems encompass ever more responsibility, relatively few people will

actually build operating systems software. But most of us will work on complex software, and we will encounter problems that have already been encountered, understood, and solved by computer scientists, architects, and developers in the field of Operating Systems.

As you read about various mechanisms and issues within operating systems, study them with these principles in mind. As we analyze implications and contrast alternatives, try to pull these principles into the discussion and see if they shed light on the problems. Understand these principles, learn to recognize these problems when you encounter them, and how to apply these solutions to new problems.

> *Powerful tools these are.*
> *Serve you they will.*

Software Interface Standards

Introduction

Over two thousand years ago, Carthage defined standard component sizes for their naval vessels, so that parts from one ship could be used to repair another. Slightly more recently, George Washington gave Eli Whitney a contract to build 12,000 muskets with interchangeable parts. The standardization of component interfaces makes it possible to successfully combine or recombine components made at different times by different people.

In the last half-century we have seen tremendous advances in both electronics and software, many of which are attributable to the availability of powerful, off-the-shelf components (integrated circuits, and software packages) that can be used to design and develop new products. We no longer have to reinvent the wheel for each new project, freeing us to focus our time and energy on the novel elements of each new design. As with ship and musket parts, the keys to this are interchangeability and interoperability: confidence that independently manufactured components will fit and work together. This confidence comes from the availability of and compliance with detailed component specifications.

In the early days of the computer industry, much key software (operating systems, compilers) was developed by hardware manufacturers, for whom compatibility was an anti-goal. If a customer's applications could only run on their platform, it would be very expensive/difficult for that customer to move to a competing platform. Later, with the rise of Independent Software Vendors (ISVs) and *killer applications*, the situation was reversed:

- The cost of building different versions of an application for different hardware platforms was very high, so the ISVs wanted to do as few ports as possible.
- Computer sales were driven by the applications they could support. If the ISVs did not port their applications to your platform, you would quickly lose all of your customers.
- Software Portability became a matter of life and death for both the hardware manufacturers and the software vendors.

If applications are to be readily ported to all platforms, all platforms must support the same services. This means that we need detailed specifications for all services, and comprehensive compliance testing to ensure the correctness and compatibility of those implementations.

The other big change was in who the customers for computers and software were. In the 1960s, most computers were used in business and technology, and the people who used computers were professional programmers: trained and prepared to deal with the adverse side effects of new software releases. Today, most computers are in tablets, phones, cars, televisions, and appliances; and the people who use them are ordinary consumers who expect them to just work. These changes imply that:

- New applications have to work on both old and new devices.
- Software upgrades (e.g. to the underlying operating system) cannot break existing applications.

But the number of available applications, and the independent development and delivery of OS upgrades and new applications make it impossible to test every application with every OS version (or vice versa). If the combinatorics of the problem render complete testing impossible, we have to find another way to ensure that whatever combination of software components wind up on a particular device will work together. And again, the answer is detailed specifications and comprehensive compliance testing.

The last forty years have seen a tremendous growth in software standards organization participation (both by technology providers and consumers) and the the scope, number, and quality of standards.

Challenges of Software Interface Standardization

When a technology or service achieves wide penetration, it becomes important to establish standards. Television would never have happened if each broadcaster and manufacturer had chosen different formats for TV signals. In the short term, a slightly better implementation (e.g. cleaner picture) may give some providers an advantage in the market place. But ultimately the fragmentation of the marketplace ill-serves both producers and consumers.

If the stake-holders get together and agree on a single standard format for TV broadcasts, every TV set can receive every station. Consumers are freed to choose any TV set they want to watch any stations they want. Since all TV sets receive the same signals, set manufacturers must compete on the basis of price, quality, and other features. The cost of TV sets goes down and their quality goes up. Since all stations can reach all viewers, they must compete on the basis of programming. The amount, quality, and variety of programming increases. And the growing market creates opportunities for new products and services that were previously unimaginable.

Standards are a good thing:

- The details are debated by many experts over long periods of time. The extensive review means they will probably be well considered, and broad enough to encompass a wide range of applications.
- Because the contributors work for many different organizations, with different strengths, it is likely that the standards will be relatively *platform-neutral*, not greatly favoring or disadvantaging any particular providers.
- Because they must serve as a basis for compatible implementations, they tend to have very clear and complete specifications and well developed conformance testing procedures.
- They give technology suppliers considerable freedom to explore alternative implementations (to improve reliability, performance, portability, maintainability), as long as they maintain the specified external behavior. Any implementation that conforms to the standard should work with all existing and future clients.

But standardization is the opposite of diversity and evolution, and so is a two-edged sword:

- At the same time, they greatly constrain the range of possible implementations. An interface will specify who provides what information, in what form, at what time. It might be possible to provide a much better implementation of that functionality, if slightly different information could be provided at a different time. But if the new interfaces are not 100% upwards compatible with the old ones, the improved design would be non-compliant. Television was pioneered in the United States, but because we were among the first to standardize (e.g. before color was imagined to be possible), a few decades later we found ourselves locked into having the worst (most primitive) TV signals on earth.
- Interface standards also impose constraints on their consumers. An application that has been written to a well-supported standard can have high confidence of working on any compliant platform. But this warrantee is voided if the application uses any feature in any way that is not explicitly authorized by the standard. An application that uses non-standard features (or standard features in non-standard ways) is likely to encounter problems on new platforms or with new releases of the software providing those features. For many years, Windows users refused to upgrade to new releases because of the near certainty that a few of their devices and applications would stop working.
- When many different stake-holders depend on a standard, it often becomes difficult to evolve those standards to meet changing requirements:

a. There will be many competing opinions about exactly what the new requirements should be and how to best address them.
b. It seems that any change (no matter how beneficial) will adversely affect some stake-holders, and they will object to the improvements.

These trade-offs are fundamental to all standards. But there are additional considerations that make software interface standards particularly difficult.

Confusing interfaces with Implementations

Engineers are inclined to think in terms of design: how we will solve a particular problem? But interface standards should not specify design. Rather they should specify behavior, in as implementation-neutral a way as possible. But true implementation neutrality is very difficult, because existing implementations often provide the context against which we frame our problem statements. It is common to find that a standard developed ten years ago cannot reasonably be extended to embrace a new technology because it was written around a particular set of implementations.

The other way this problem comes about is when the interface specifications are written after the fact (or worse by reverse engineering):

- Since the implementation was not developed to or tested against the interface specification, it may not actually comply with it.
- In the absence of specifications, it may be difficult to determine whether particular behavior is a fundamental part of the interface or a peculiarity of the current implementation.

A common piece of Management 1A wisdom is:

> "If it isn't in writing, it doesn't exist"

This is particularly true of interface specifications.

The rate of evolution, for both technology and applications

The technologies that go into computers are evolving quickly (e.g. Wifi, Bluetooth, USB, power-management, small screen mobile devices), and our software must evolve to exploit them. The types of applications we develop are also evolving dramatically (e.g. from desk-top publishing to social networking). Nobody would expect to be able to pull the engine out of an old truck and drop-in replace it with a new hybrid electric motor. Yet everybody expects to be able to run the latest apps on their ancient phone, laptop or desktop for as long as it keeps running.

Maintaining stable interfaces in the face of fast and dramatic evolution is extremely difficult. Microsoft Windows was designed as a single-user Personal Computer operating system - making it very difficult to extend its OS service APIs to embrace the networked applications and large servers that dominate today's world. 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.

Proprietary vs Open Standards

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). Whenever a technology provider develops a new technology they must make a decision:

- Should they open their interface definitions to competitors, to achieve a better standard, more likely to be widely adopted? This costs of this decision are:
  - Reduced freedom to adjust interfaces to respond to conflicting requirements.
  - Giving up the competitive advantage that might come from being the first/only provider.
  - Being forced to re-engineer existing implementations to bring them into compliance with committee-adopted interfaces.
- Should they keep their interfaces proprietary, perhaps under patent protection, to maximize their competitive advantage? The costs of this decision are:
  - If a competing, open standard evolves, and ours is not clearly superior, it will eventually lose and our market position will suffer as a result.
  - Competing standards fragment the market and reduce adoption.
  - With no partners, we will have to shoulder the full costs of development and evangelization. In an open standards consortium, these costs would be distributed over many more participants.

This dilemma often complicates the development of interface standards. The participants may not only be trying to develop strong standards; Many are also trying to gain market position and protect prior investments.

Application Programming Interfaces

Most of the time, when we look up and exploit some service, we are working with 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

The specifications may also include or refer to sample usage scenarios.

The most important thing to understand about API specifications is that they are written at the source programming level (e.g. C, C++ , Java, Python). They describe how source code should be written in order to exploit these features. API specifications are a basis for software portability, but applications must be recompiled for each platform:

- The benefit for applications 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 application written to supported APIs should easily port

to their platform.

But this promise can only be delivered if the API has been defined in a platform-independent way:

- An API that defined some type as int (implicitly assuming that to be at least 64 bits wide) might not work on a 32 bit machine.
- An API that accessed individual bytes within an int might not work on a big-endian machine.
- An API that assumed a particular feature implementation (e.g. *fork(2)*) might not be implementable on some platforms (e.g. Windows).

One of the many advantages of an Open Standard is that diverse participants will find and correct these problems.

Application Binary Interfaces

Well defined and widely supported Application Programming Interfaces are very good things, but they are not enough. If you are reading this, you are probably fully capable of re-compiling an application for a new platform, given access to the sources. But most users do not have access to the sources for most of the software they use, and probably couldn't successfully compile it if they did. Most people just want to download a program and run it.

But (if we ignore interpreted languages like Java and Python) executable programs are compiled (and linkage edited) for a particular instruction set architecture and operating system. How is it possible to build a binary application that will (with high confidence) run on any Android phone, or any x86 Linux? This problem is not addressed by (source level) Application Programming Interfaces. This problem is addressed by Application Binary Interfaces (ABIs):

> An Application Binary Interface is the binding of an Application Programming Interface to an Instruction Set Architecture.

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 instructions 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).

Who actually uses the Application Binary Interfaces

Most programmers will never directly deal with an ABI, as it is primarily used by:

- the compiler, to generate code for procedure entry, exit, and calls
- the linkage editor, to create load modules
- the program loader, to read load modules into memory
- the operating system (and low level libraries) to process system calls

But it is occasionally used by programmers who need to write assembly language subroutines (for efficiency or privileged operations) that can be called from a higher level language (e.g. C or C++).

Interface Stability

Mark Kampe

Interface Specifications

American History books like to credit Eli Whitney with the invention of manufacturing with interchangeable parts. Actually,

- Whitney's parts weren't all that interchangeable,
- he got the idea from a French gunsmith (Honore Blanc),
- the Swedish clock maker Christopher Polhem did a much better job with clock gears 50 years earlier.

But, be that as it may, interchangeable parts revolutionized, and in fact enabled modern manufacturing. The underlying concept is that there be specifications for every part. 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 same principle applies to software. If you want to open a file on a Posix system, you use the **open** system call. There may be a hundred different implementations of open but this doesn't matter because they all conform to the same interface specification. The rewards for this standardization are:

a. Your work as a programmer is simplified, because you don't have to write your own file I/O routines.
b. Your program is likely to be easily portable to any Posix compliant system, because they all provide the same file I/O services.
c. Training time for new programmers is reduced, because everybody already knows how to use the Posix file I/O functions.

The Criticality of Interfaces in Architecture

A system architecture defines the major components of the system, the functionality of each, and the interfaces between them. Clearly the component interface specifications are a critical element of any architecture. To the extent that these interfaces have been well considered and well defined, it should be possible to (at least semi) independently design and implement those components. In principle, any implementations that satisfy those functional and interface specifications should combine to yield a working system.

The converse is also true. To the extent that interfaces between components were poorly considered or specified, it becomes unlikely that independently developed components will work when combined together, and more likely that subsequent changes to one component will break others.

The Importance of Interface Stability

We write contracts is so that everybody knows what will be expected of them, and what they can expect from the other parties to the agreement. Everybody will depend on you to uphold your obligations under the contract. A software interface specification is a form of 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.

And in return for this they get all the benefits described above.

Suppose that some architectural genius realized that Posix file semantics are too weak to describe the behavior of files in a distributed system, and that this could be solved by adding a new (required) parameter (for a call-back routine) to the Posix open call. What would happen?

- 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 other Posix compliant systems.
- software written to the old semantics would not work on the new systems.
- customers (knowing nothing about this techno-feud) would be faced with inexplicable failures, and would start complaining to their software and system providers (none of whom were responsible for the change).
- programmers would be confused about which version of open they were using, and would probably get around the problem by writing their own file I/O packages ... a bunch of extra work, but at least it would protect them from future such acts of mischief.

And as a result of this, we would lose all of the benefits described above. When you promise somebody that you will do something (e.g. conform to a specified interface), and they depend on you (e.g. by writing code to that interface), and you don't follow through (e.g. make an incompatible change) ... problems are likely to ensue (e.g. failures, bug reports, product gets a bad reputation, going out of business, etc.).

Interfaces vs. Implementations

Suppose that someone writes a handy library to efficiently provide reliable communication over an unreliable network, and I decide I want to use it. I download their development kit and start using it, and find it to be great. A few weeks into the project I realize that I need a feature that is not included in their documentation, but after reading their code I discover that the needed feature is actually available. I use it and my product is a great success.

Six months later, they release a new version of their reliable communications library, and my product immediately breaks. After a little debugging I discover that they have changed the undocumented code that I was depending on. It turns out that I had not designed my product to work with their *interfaces*. My product only worked with a particular *implementation* ... and implementations change.

People often put much more work into designing and maintaining their code than to clarifying and maintaining their documentation. This makes it tempting to reverse engineer the interface specifications from a provided implementation. But an interface should exist (and be defined) independently from any particular implementation. Confusing an implementation with an interface usually ends badly!

Program vs. User Interfaces

Human beings are amazingly robust. We could change the text in dialog boxes, rearrange input forms, add new required input items, and rework all of the menus in a program, and many users would figure out the changes in a few seconds. This inclines many developers to be cavalier in making changes to user interfaces.

Code is nowhere nearly so robust. If I expect my second parameter to be a file name, and you pass me the address of a call-back routine instead, the best we can hope for is a good error message and a quick coredump with no data corruption.

If all users were developers, and only ran their own code, this might be just an irritation. We would get the core dump, track it down, figure out that some idiot had made this change, recode accordingly, and an hour later we'd be good as new. Unfortunately most people run code that they do not understand, and have no way to debug or fix. If a program breaks, all I can do is call support and complain. I am helpless. Users don't care which programmer goofed up. "I bought your product. It doesn't work. I'm taking my business elsewhere!".

If you are going to be delivering binary software to non-developers (that describes 99.999% of all software) you have to trust that what ever platform they run it on will correctly implement all of the interfaces on which your program depends.

The same argument applies, though not quite as strongly, to independent components in a single system. If components exchange services, and I make an incompatible change to the interfaces of one component, this has the potential to break other components in the same system. I can fix the other components in our system to work with the new changes but:

- this complicates the making of the change.
- we have to ensure that mismatched component sets are impossible.

Is every interface carved in stone?

Absolutely not.

You are free to add features that do not change the interface specification (a faster or more robust implementation). In many cases you can add upwards-compatible extensions (all old programs will still work the same way, but new interfaces enable new software to access new functionality). There are a few ways to add incompatible changes to old interfaces without breaking *backwards compatibility*:

Interface Polymorphism

In old-school languages (like FORTRAN and C) a routine had a single interface specification. But more contemporary programming languages support 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

Backwards compatibility requirements do not prevent us from changing interfaces; they merely require us to continue providing old interfaces to old clients. In many cases, it may be possible for an application to call out which version of an interface it requires:

- Java programs are labeled with the run-time-environment versions they require.
- Network protocol (e.g. RPC) sessions often begin with a version negotiation, where each side states the interface versions that it can support, so that a mutually acceptable version can be chosen.

This approach does permit incompatible interface changes, but still requires servers to support old as well as new interface versions.

But even if we have no ability to deliver multiple interface versions, this does not necessarily mean I can never make incompatible changes. It is not the case that all interfaces must be supported for all of time. It is merely necessary that the interface stability be understood, and adequate to achieve the product (or project) goals:

- It is common to provide alpha/evaluation access to proposed new services/interfaces. This is done with the understanding that the final interfaces are likely to be different. Such releases are provided purely for evaluation purposes. Understanding the likelihood of change, developers should only use them for prototyping, and should not base long-lived products on them.
- Many suppliers have interface stability guidelines for their releases. A typical example might be:
  - a micro-release (e.g. 3.4.1) includes only bug fixes.
  - a minor-release (e.g. 3.5) includes new functions, but us upwards compatible with the underlying major release version.
  - a major release (e.g. 4.0) is allowed to make incompatible changes to previously committed interfaces.

  Applications software developers can use this to determine which versions their software will run on, and customers can use this to decide whether or not they can safely upgrade to a new release.
- Many suppliers have make specific support commitments (e.g. we will support release 3.5 for five years from its initial availability date). This gives software developers and customers confidence that they will have a stable software platform for at least that long. At the end of that period, customers may have to choose between continuing to run on a no-longer-supported platform or moving to new versions of their mission-critical applications.

How stable an interface needs to be depends on how you intend to use it. What is important is that:

1. we be honest about our intentions and set realistic expectations.
2. we have a plan for how we will manage change.

Interface Stability and Design

So, if we want to expose an interface to unbundled software with high quality requirements, we are not allowed to change it in non-upwards-compatible ways. That sounds like a bother, but if that is the rules, so be it. What has this got to do with architecture and design?

When we design a system, and the interfaces between the independent components, we need to consider all of the different types of change that are likely to happen over the life of this system. When we specify our external component interfaces we need to have high confidence that will be able to accommodate all of the envisioned evolution while preserving those interfaces.

- We might rearrange the distribution of functionality between components to create a simpler (and hence more preservable) interface between them.
- We might design features that we may never implement, just to make sure that the specified interfaces will accommodate them if we ever do decide to build them.
- We might introduce (seemingly) unnatural degrees of abstraction in our services to ensure that they leave us enough slack for future changes.

We must consider which of our interfaces will need to be how stable, and design our system with those stabilities in mind.

Object Modules, Linkage Editing, Libraries

Introduction

One of the most fundamental abstract resources implemented by an operating system is the process. A process is often defined as an executing instance of a program. If we want to understand what a process is, it is very helpful to understand what a program is. We most often think of a program as one or more files (e.g. C, Java, Python). These are not executable programs, but rather source files that can be translated into machine language and combined with other machine language modules to create executable programs.

From the operating system's perspective, a program is a file full of ones and zeros, that when loaded into memory, become machine-language instructions that can be executed by the computer on which we are running.

- How do our source modules come to be translated into machine language instructions?
- How do they come to be combined with other machine language modules to form complete programs?
- What is the format of a file that contains an executable program, and how does it come to be correctly loaded into memory?

These are a few of the questions we will discuss in this chapter.

The Software Generation Tool Chain

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, assembler, 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 operating system) and executed (by the CPU).

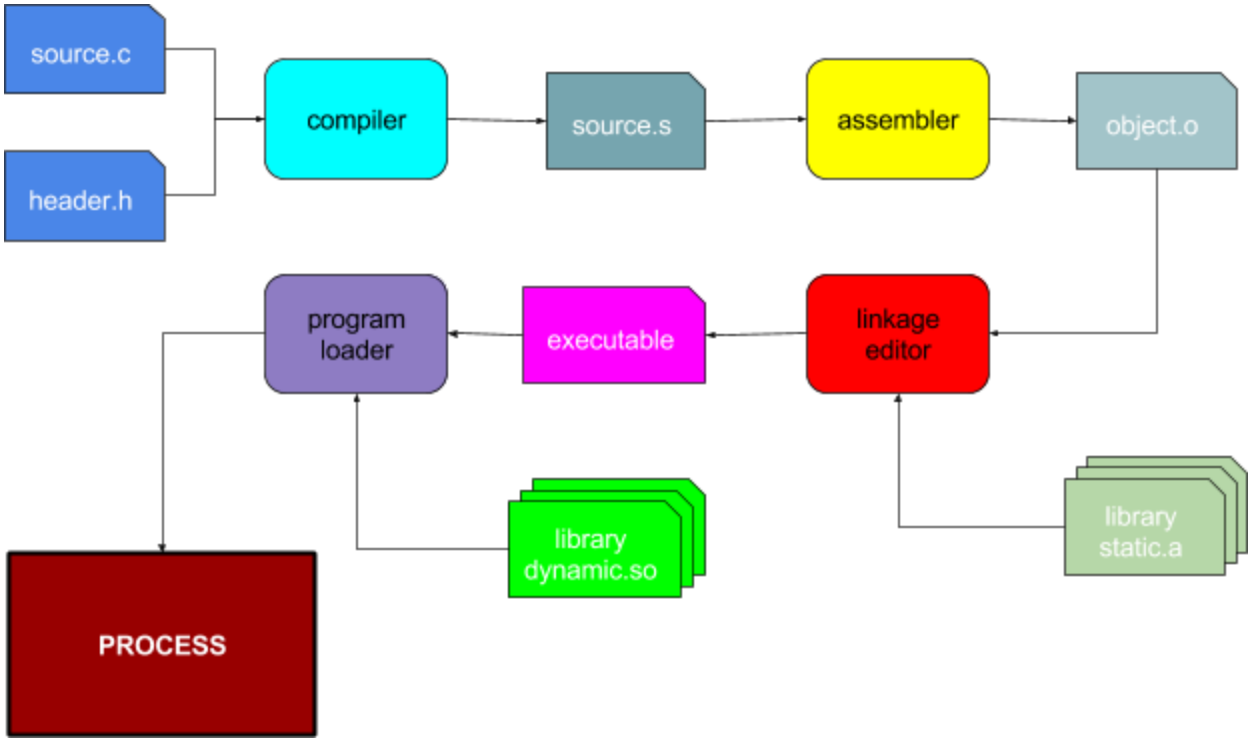


Fig 1. Components of the Software Generation Tool Chain

This figure shows a typical software generation tool chain, with the rounded boxes representing software tools that are run and the rectangles with one corner cut off representing files used (and sometimes created) during the process. The large rectangle in the lower left is the final result, a process that the operating system can schedule and run.

Let's consider the components of this software tool chain in the order they are used.

Compiler

Reads source modules and included header files, parses the input language (e.g. C or Java), and infers the intended computations, for which it will generate lower level code. More often than not, that code will be produced in assembly language code rather than machine language. This provides greater flexibility for further processing (e.g. optimization), may make the compiler more portable, and simplifies the compiler by pushing some of the work out to a subsequent phase. There are, however, languages (e.g. Java, Python) whose compilers directly produce a pseudo-machine language that will be executed by a virtual machine or interpreter.

Assembler

Assembly language is much lower level, with each line of code often translating directly to a single machine language instruction or data item. But the assembly language still allows the declaration of variables, the use of macros, and references to externally defined code and data. Developers occasionally write routines directly in assembly language (e.g. when they need specific code that the compiler is incapable of generating).

In user-mode code, modules written in assembler often include:

- performance critical string and data structure manipulations
- routines to implement calls into the operating system

In the operating system, modules written in assembler often include:

- CPU initialization

- first level trap/interrupt handlers
- synchronization operations

The output of the assembler is an object module containing mostly machine language code. But, because the output corresponds only to a single input module for the linkage editor:

- 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 linkage editor reads a specified set of object modules, placing them consecutively into a virtual address space, and noting where (in that virtual address space) each was placed. It also notes unresolved external references (to symbols referenced by, but not defined by the loaded object modules). It then 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 of the required (specified and implied) object modules, it finds every reference to a relocatable or external symbol and updates 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 into memory and executed, and they are written out into a new file, called an executable load module.

Program loader

The program loader is usually part of the operating system. It examines the information 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.

Once the virtual address space has been created and the required code has been copied into (virtual) memory, the program can be executed by the CPU.

Object Modules

A program must be complete before it is ready to be loaded into memory and be executed; all of the code to be executed must be included in the program. But when we write software, we do not put all of the code that will be executed into a single file:

- A single file containing everything would be huge, difficult to understand, and cumbersome to update. Code is more understandable and maintainable if it different types of functionality are broken out into (relatively) independent modules.
- Many functions (e.g. string manipulation, formatted output, mathematical functions, image decoding) are commonly used. Making these modules available for reuse (from externally supplied libraries) greatly reduces the work associated with writing new programs.

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

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

Obviously the code (machine language instructions) within an object module are Instruction Set Architecture (ISA) specific; The pattern of ones and zeroes that represents an add instruction for an Intel Pentium is quite different than those that represent the same operation on an ARM or PowerPC. But it might surprise you to learn that many contemporary object module formats are common across many Instruction Set Architectures. One very popular format (for Unix and Linux systems) is called ELF (Executable and Linkable Format). The ELF format is described in elf(5), but an ELF module is 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 relocated (e.g. 32 or 64 bit address)
  - the symbol table entry, whose address should be used to perform that relocation

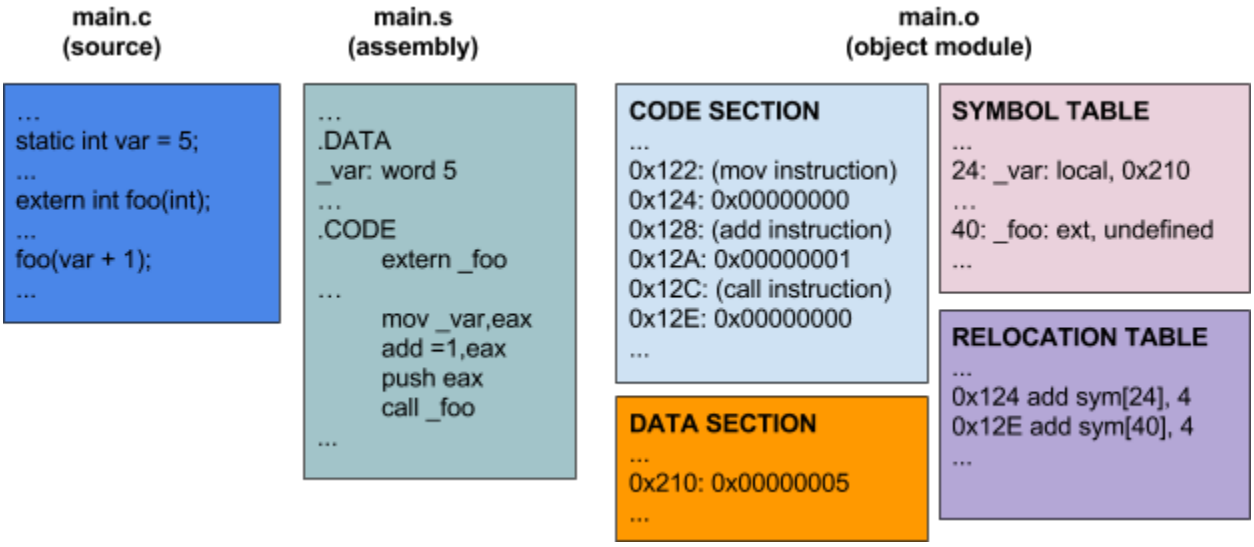


Fig 2. A Program in Various Stages of Preparation for Execution

Libraries

In addition to its own modules, an interesting program could easily use hundreds, or even thousands of standard/reusable functions. Specifying which of thousands of available functions are to be included would be extremely cumbersome. This problem is dealt with by creating libraries of useful functions. A library is simply a collection of (usually related) object modules. One library might contain standard system calls, while another might contain commonly used mathematical functions. Reusable code is often distributed in and obtained from libraries. But not all libraries are public collections of reusable code. If my program were made up of hundreds of modules, I might choose to organize my own code into one or more libraries. Different operating systems (and some languages) may implement libraries in different ways (e.g. Java packages), but the concept of packaging groups of related modules together is common to most software

development tool chains.

The Linux command for creating, updating, and examining libraries is ar(1).

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

Libraries are not always orthogonal and independent:

- It is common to implement higher level libraries (e.g. image file decoding) using functionality from lower level libraries (e.g. mathematical functions and file I/O).
- It is not uncommon to use alternative implementations for some library functionality (e.g. a diagnostic memory allocator) or to intercept calls to standard functions to collect usage data.

This means that the order in which libraries are searched may be very important. If we call a function from library A, and library A calls functions from library B, we may need to search library A before searching library B. If we want to override the standard *malloc(3)* with valgrind's more powerful diagnostic version, we need to search the valgrind library before we search the standard C library.

Linkage Editing

At least 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.

These operations are called Linkage Editing, because we are filling in all of the linkages (connections) between the loaded modules. The program that does this is called a linkage editor. The command to perform linkage editing in UNIX/Linux systems is ld(1). Going back to our previous object module example, the linkage editor would search the specified libraries for a module that could satisfy the external _foo reference,

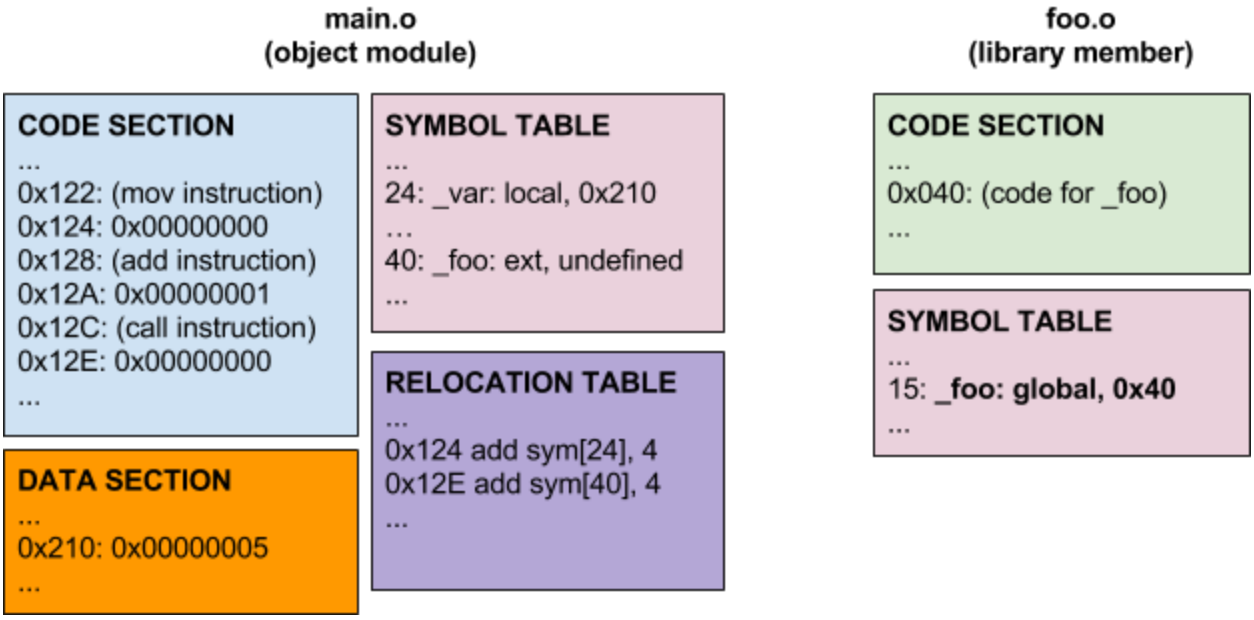


Fig 3. Finding External References in a Library

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

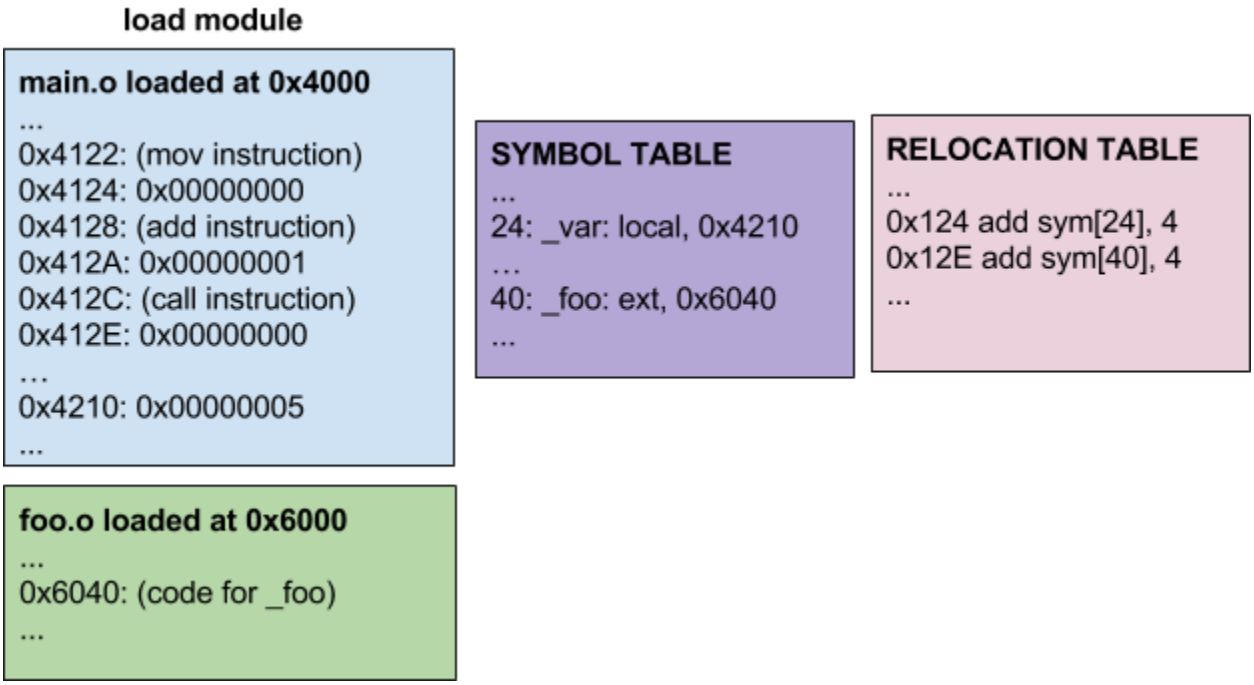


Fig 4. Updating a Process' Virtual Address Space

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.

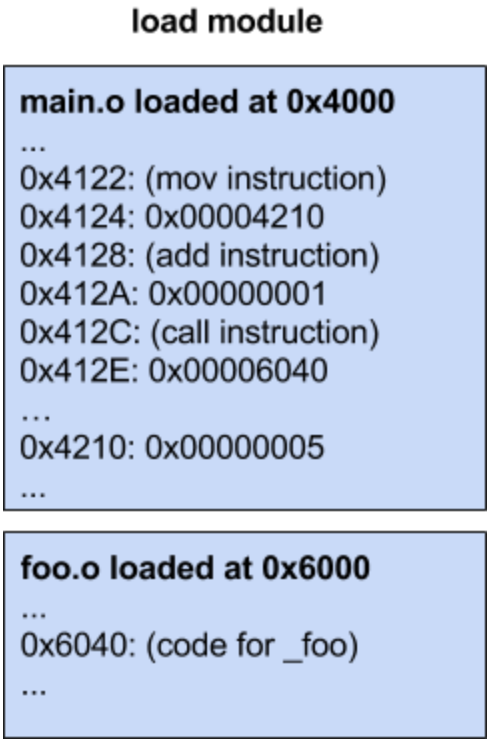


Fig 5. Performing a Relocation in an Load Module

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

A load module is similar in format to an object module, in that it contains multiple sections (code, data, symbol table); But, unlike an object module, it is (a) complete and (b) requires no relocation. Each section specifies the address (in the process' virtual address space) at which it should be loaded. When the operating system is instructed to load a new program (with the exec(2) system call), 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.

At this point, the program is ready to execute, and the operating system transfers control to its entry point (which is also specified in the load module).

You might wonder, if there is no further relocation to be performed, why a load module might still contain a symbol table? Neither loading nor executing the program requires the symbol table. But if the program were to receive an exception (say at address 0x604C), the symbol table would enable us to determine that the error occurred twelve bytes into the _foo routine. Many load modules do not have any symbol tables (to save space, reduce download time, or make it harder for competitors to disassemble their code). Some load modules contain extensive symbol tables, including entry point addresses, data structure descriptions, source code line numbers, and other information to assist intelligent debuggers.

Static vs. Shared Libraries

In the above described linkage editing process, library modules were directly (and permanently) incorporated into the load module.

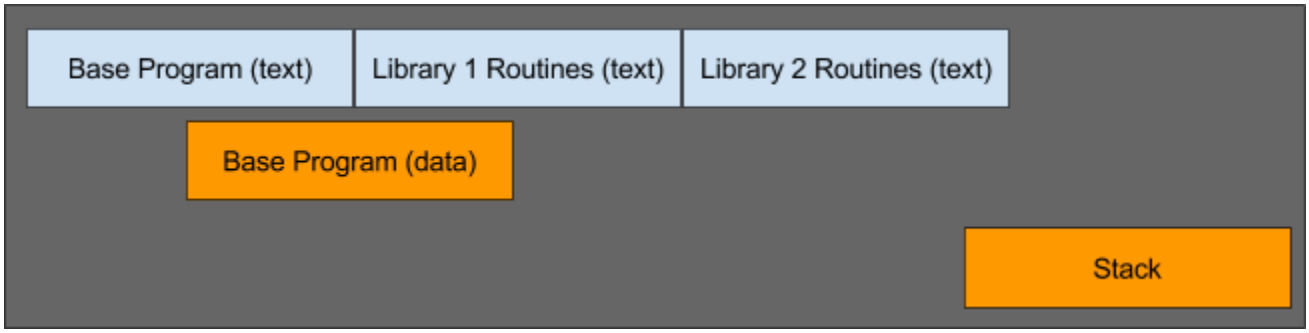


Fig 6. Virtual Address Space with Statically Linked Libraries

Because of this permanence, this process is referred to as static linking. It has at least two significant disadvantages:

- Many libraries (e.g. libc) are used by almost every program on the system. Thousands of identical copies of the same code increase the required down-load time, disk space (to store them), start-up time (to read them into memory) and memory (while they are executing). It would be much more efficient if we could somehow allow all of the programs that used a popular library to share a single copy.
- Popular libraries change over time with enhancements, optimizations, and bug fixes. Some enhancements may be very important (e.g. enabling applications to work with a new version of the operating system). In most cases, a newer version of a library is probably better than an older version. But with static linking, each program is built with a frozen version of each library, as it was at the time the program was linkage edited. It might be better (for software reliability) if it was possible to automatically load the latest library versions each time a program was started.

These issues are addressed by run-time loadable, shared libraries. There are many possible approaches, but a very simple way to implement shared libraries is to:

- Reserve an address for each shared library. This is possible in 32-bit architectures, and easy in 64-bit architectures.
- Linkage edit each shared library into a read-only code segment, 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 linkage 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. The stub library also includes symbol table information that informs the operating system what shared library segment this program requires.
- Linkage edit the client program with the stub library.
- When the operating system loads the program into memory, it will notice that the program requires shared libraries, and will open the associated (sharable, read-only) code segments and map them into the new program's address space at the appropriate location. This process is described in ld.so(8).

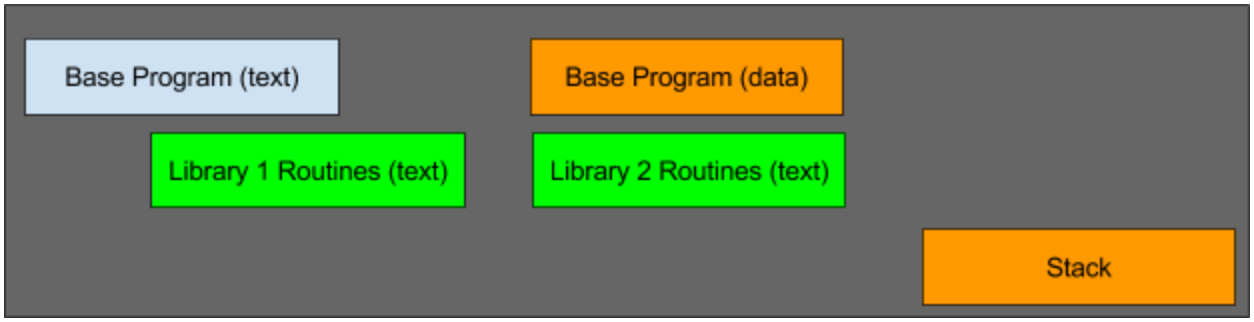


Fig 7. Virtual Address Space with Shared Libraries

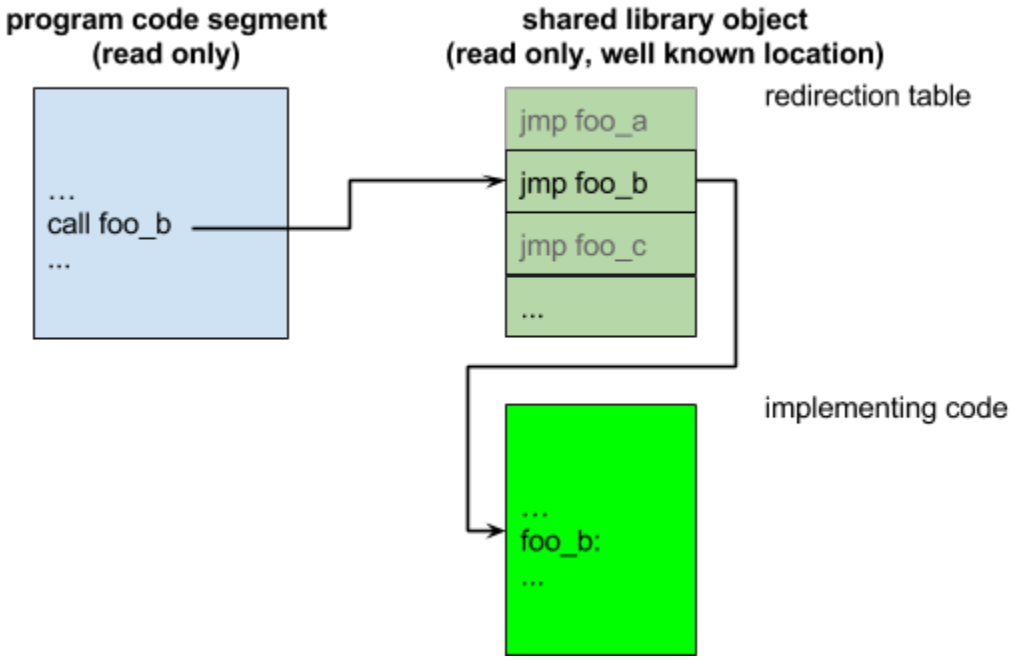


Fig 8. Linking Shared Libraries

In this way:

- A single copy of a shared library implementation (the shared segment) can be shared among all of the programs that use that library.
- 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.

Routines to be included in shared libraries must be designed with these limitations in mind. It may not be

possible to put arbitrary subroutines into a shared library.

Dynamically Loaded Libraries

Shared Libraries are very powerful and convenient, but they too have proved to be too limiting for many applications:

- There may be very large/expensive libraries that are seldom used; Loading/mapping such libraries into the process' address space at program load time unnecessarily slows down program start-up and increases the memory footprint. In some cases, it might be preferable to delay the loading of a module until it is actually needed.
- While loading is delayed until program load time, he 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.

These lead us to Dynamically Loadable Libraries (DLLs): libraries that are not loaded (or perhaps even chosen) until they are actually needed. Again, there are many implementations, but the general model is:

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

The Linux support for user mode Dynamically Loadable Libraries is described in dlopen(3). This sort of mechanism is used in user mode (e.g. to support different MIME times) as well as inside the operating system (to dynamically load device drivers or file system implementations). In either case it makes it possible for pre-compiled applications to exploit plug-ins to support functionality that was not implemented (or even envisioned) at the time the original application was built. This is very powerful.

Calls from the client application into a shared or Dynamically Loaded library are generally handled through a table or vector of entry points, much like object method invocation. There is much more diversity in the handling of calls from a library back into the application:

- An application can register a call-back routine by passing its address to an appropriate library registration method.
- The same approach can be generalized to a large number of functions by having the application call the library to register a vector of entry points to perform standard functions.
- Dynamically loaded kernel modules (e.g. device drivers and file systems) are allowed to access a myriad of functions and data structures within the kernel into which they have been loaded. This is often enabled by a run-time-loader that effectively linkage edits the newly loaded module against the operating system, filling in the addresses of all unresolved external references from the dynamically loaded module into the operating system. The Linux kernel run-time loader is described in ld.so(8).

Implicitly Loaded Dynamically Loadable Libraries

It should be noted that there are Dynamically Loadable Library implementations that are more similar to shared libraries:

- Applications are linkage edited against a set of stubs, which create Program Linkage Table (PLT) entries in the client load module.
- The PLT entries are initialized to point to calls to a Run-Time Loader.
- The first time one of these entry points is called, the stub calls the Run-Time Loader to open and load the appropriate Dynamically Loadable Library.
- After the required library has been loaded, the PLT entry is changed to point to the appropriate routine in the newly loaded library. From then on, all calls through that PLT entry go directly to the now-loaded routine.

Such implicitly loaded DLLs are (from the client perspective) almost indistinguishable from statically loaded libraries or shared libraries. From this perspective, both Dynamically Loadable and shared libraries can reduce the size of load modules (vs. statically linked libraries) and allow a single on-disk/in-memory code segment to be shared by multiple concurrently running programs.

The greater functionality and performance benefits of Dynamically Loadable Libraries are only available when the client applications become aware of them:

- Shared libraries allow delayed (program load time) binding to some version of a library that was chosen at linkage-edit time. Dynamically Loadable Libraries allow the library to be loaded to be chosen at run time.
- Shared libraries consume memory for the entire time that the process is running. Dynamically Loadable Libraries can be unloaded when they are no longer needed.
- Shared libraries impose numerous constraints and simple interfaces on the code they contain. Dynamically Loadable Libraries are capable of performing complex initialization and supporting complex (bi-directional) calls between the client application and library.
- Shared libraries are (from the client applications perspective) indistinguishable from statically loaded libraries, and very simple to use. Dynamically Loadable Libraries may require considerably more work (from the client application) to load the library, register the interfaces, and establish work sessions.

Shared Libraries are a more efficient mechanism for binding libraries to client applications. Dynamically Loadable Libraries are a mechanism to dynamically extend the functionality of a client application based on resources and information that may not be available until the moment they are needed.

Object Modules, Linkage Editing, Libraries

Introduction

One of the most fundamental abstract resources implemented by an operating system is the process. A process is often defined as an executing instance of a program. If we want to understand what a process is, it is very helpful to understand what a program is. We most often think of a program as one or more files (e.g. C, Java, Python). These are not executable programs, but rather source files that can be translated into machine language and combined with other machine language modules to create executable programs.

From the operating system's perspective, a program is a file full of ones and zeros, that when loaded into memory, become machine-language instructions that can be executed by the computer on which we are running.

- How do our source modules come to be translated into machine language instructions?
- How do they come to be combined with other machine language modules to form complete programs?
- What is the format of a file that contains an executable program, and how does it come to be correctly loaded into memory?

These are a few of the questions we will discuss in this chapter.

The Software Generation Tool Chain

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, assembler, 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 operating system) and executed (by the CPU).

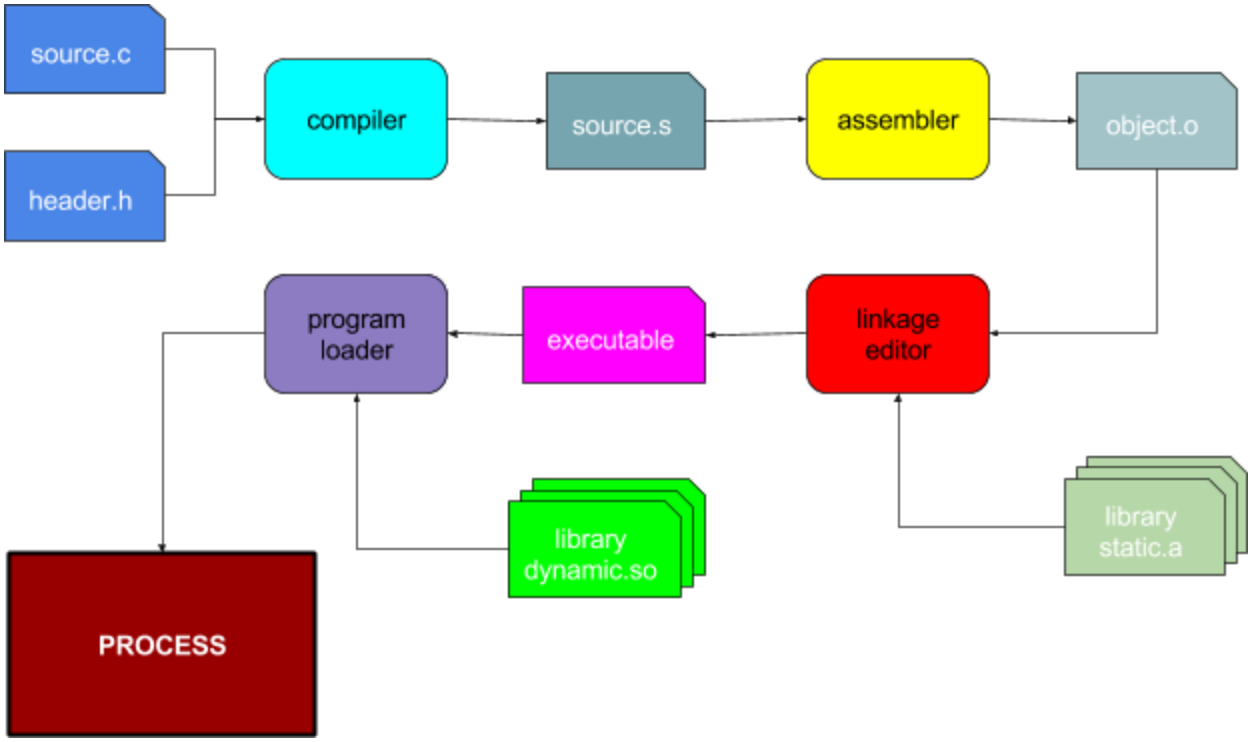


Fig 1. Components of the Software Generation Tool Chain

This figure shows a typical software generation tool chain, with the rounded boxes representing software tools that are run and the rectangles with one corner cut off representing files used (and sometimes created) during the process. The large rectangle in the lower left is the final result, a process that the operating system can schedule and run.

Let's consider the components of this software tool chain in the order they are used.

Compiler

Reads source modules and included header files, parses the input language (e.g. C or Java), and infers the intended computations, for which it will generate lower level code. More often than not, that code will be produced in assembly language code rather than machine language. This provides greater flexibility for further processing (e.g. optimization), may make the compiler more portable, and simplifies the compiler by pushing some of the work out to a subsequent phase. There are, however, languages (e.g. Java, Python) whose compilers directly produce a pseudo-machine language that will be executed by a virtual machine or interpreter.

Assembler

Assembly language is much lower level, with each line of code often translating directly to a single machine language instruction or data item. But the assembly language still allows the declaration of variables, the use of macros, and references to externally defined code and data. Developers occasionally write routines directly in assembly language (e.g. when they need specific code that the compiler is incapable of generating).

In user-mode code, modules written in assembler often include:

- performance critical string and data structure manipulations
- routines to implement calls into the operating system

In the operating system, modules written in assembler often include:

- CPU initialization

- first level trap/interrupt handlers
- synchronization operations

The output of the assembler is an object module containing mostly machine language code. But, because the output corresponds only to a single input module for the linkage editor:

- 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 linkage editor reads a specified set of object modules, placing them consecutively into a virtual address space, and noting where (in that virtual address space) each was placed. It also notes unresolved external references (to symbols referenced by, but not defined by the loaded object modules). It then 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 of the required (specified and implied) object modules, it finds every reference to a relocatable or external symbol and updates 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 into memory and executed, and they are written out into a new file, called an executable load module.

Program loader

The program loader is usually part of the operating system. It examines the information 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.

Once the virtual address space has been created and the required code has been copied into (virtual) memory, the program can be executed by the CPU.

Object Modules

A program must be complete before it is ready to be loaded into memory and be executed; all of the code to be executed must be included in the program. But when we write software, we do not put all of the code that will be executed into a single file:

- A single file containing everything would be huge, difficult to understand, and cumbersome to update. Code is more understandable and maintainable if it different types of functionality are broken out into (relatively) independent modules.
- Many functions (e.g. string manipulation, formatted output, mathematical functions, image decoding) are commonly used. Making these modules available for reuse (from externally supplied libraries) greatly reduces the work associated with writing new programs.

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

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

Obviously the code (machine language instructions) within an object module are Instruction Set Architecture (ISA) specific; The pattern of ones and zeroes that represents an add instruction for an Intel Pentium is quite different than those that represent the same operation on an ARM or PowerPC. But it might surprise you to learn that many contemporary object module formats are common across many Instruction Set Architectures. One very popular format (for Unix and Linux systems) is called ELF (Executable and Linkable Format). The ELF format is described in elf(5), but an ELF module is 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 relocated (e.g. 32 or 64 bit address)
  - the symbol table entry, whose address should be used to perform that relocation

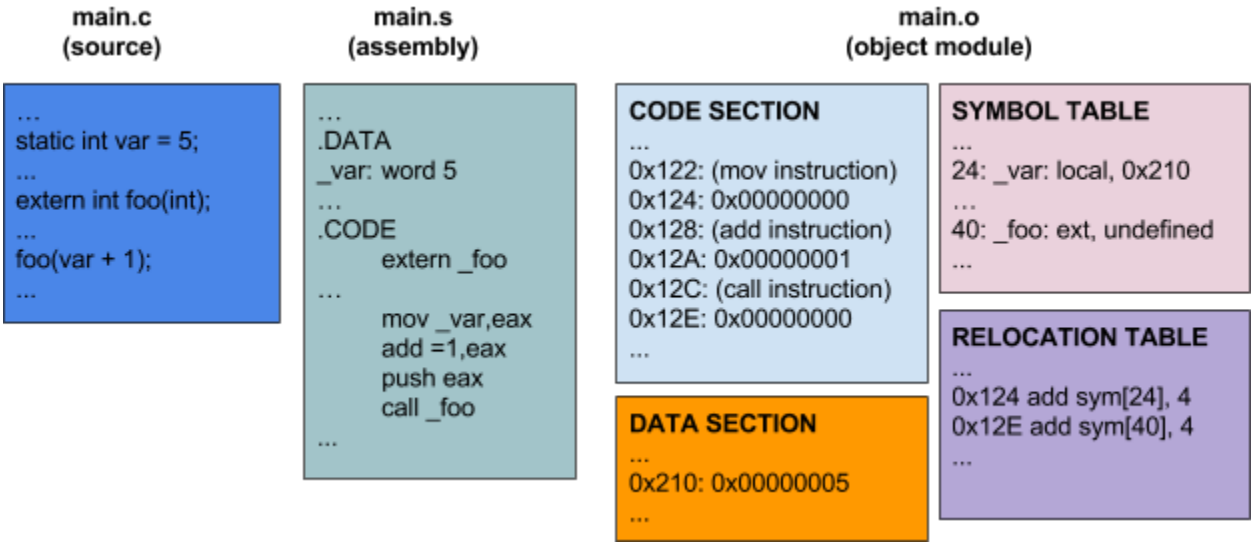


Fig 2. A Program in Various Stages of Preparation for Execution

Libraries

In addition to its own modules, an interesting program could easily use hundreds, or even thousands of standard/reusable functions. Specifying which of thousands of available functions are to be included would be extremely cumbersome. This problem is dealt with by creating libraries of useful functions. A library is simply a collection of (usually related) object modules. One library might contain standard system calls, while another might contain commonly used mathematical functions. Reusable code is often distributed in and obtained from libraries. But not all libraries are public collections of reusable code. If my program were made up of hundreds of modules, I might choose to organize my own code into one or more libraries. Different operating systems (and some languages) may implement libraries in different ways (e.g. Java packages), but the concept of packaging groups of related modules together is common to most software

development tool chains.

The Linux command for creating, updating, and examining libraries is ar(1).

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

Libraries are not always orthogonal and independent:

- It is common to implement higher level libraries (e.g. image file decoding) using functionality from lower level libraries (e.g. mathematical functions and file I/O).
- It is not uncommon to use alternative implementations for some library functionality (e.g. a diagnostic memory allocator) or to intercept calls to standard functions to collect usage data.

This means that the order in which libraries are searched may be very important. If we call a function from library A, and library A calls functions from library B, we may need to search library A before searching library B. If we want to override the standard *malloc(3)* with valgrind's more powerful diagnostic version, we need to search the valgrind library before we search the standard C library.

Linkage Editing

At least 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.

These operations are called Linkage Editing, because we are filling in all of the linkages (connections) between the loaded modules. The program that does this is called a linkage editor. The command to perform linkage editing in UNIX/Linux systems is ld(1). Going back to our previous object module example, the linkage editor would search the specified libraries for a module that could satisfy the external _foo reference,

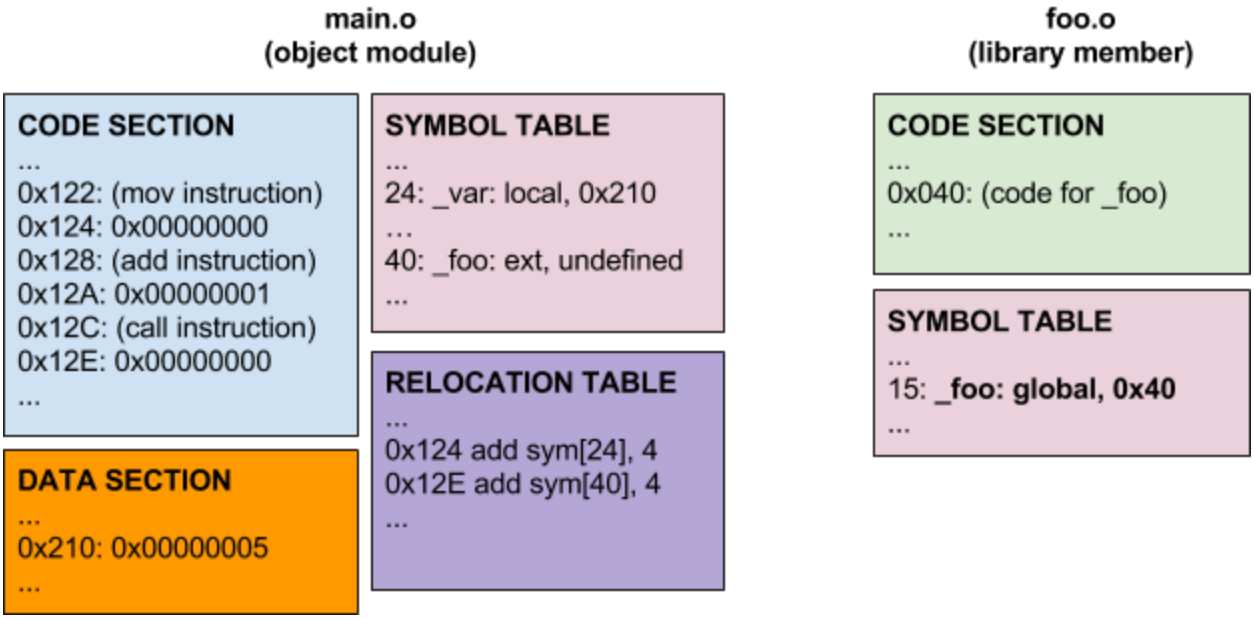


Fig 3. Finding External References in a Library

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

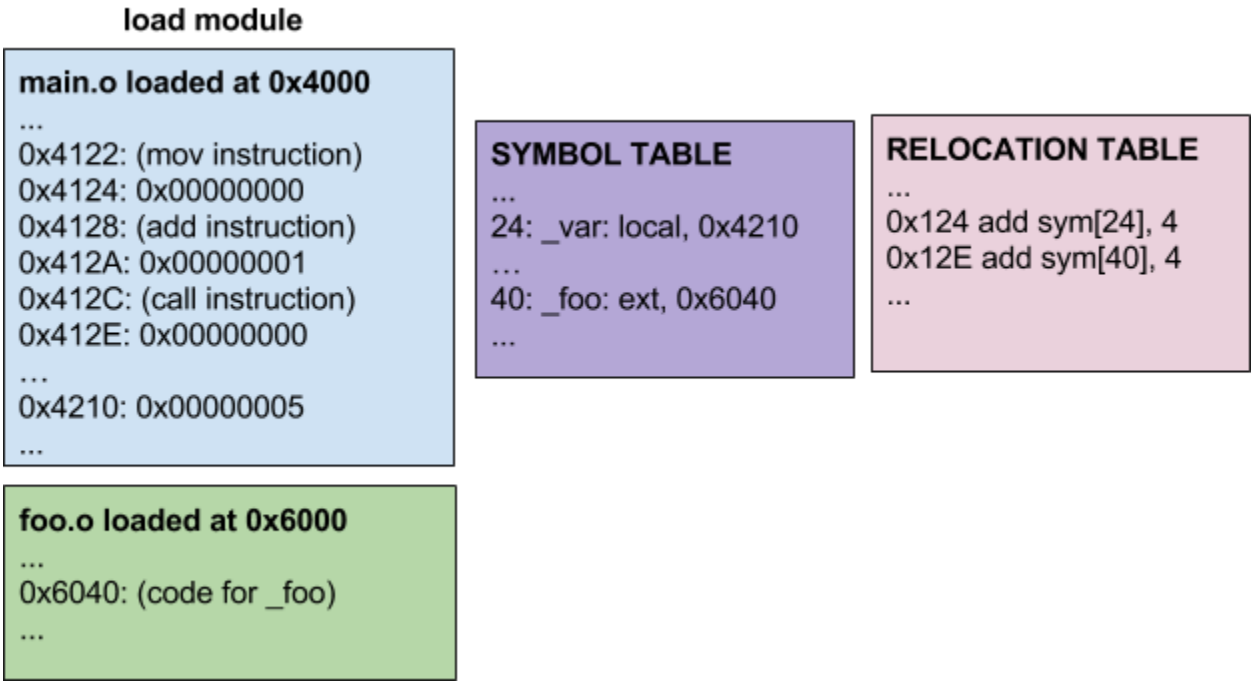


Fig 4. Updating a Process' Virtual Address Space

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.

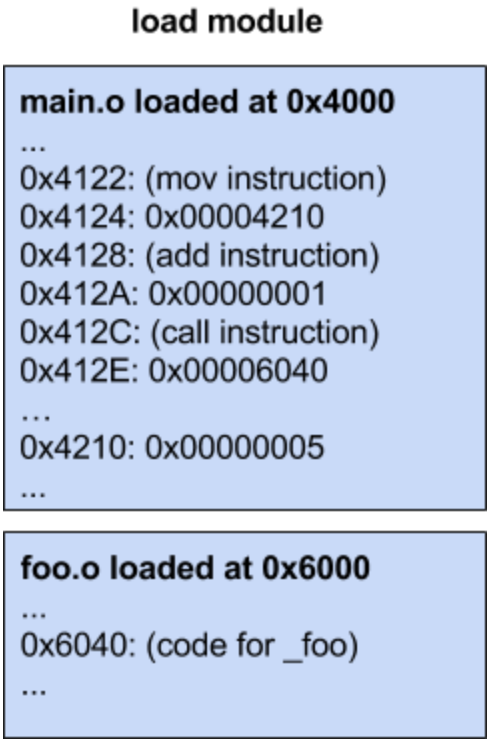


Fig 5. Performing a Relocation in an Load Module

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

A load module is similar in format to an object module, in that it contains multiple sections (code, data, symbol table); But, unlike an object module, it is (a) complete and (b) requires no relocation. Each section specifies the address (in the process' virtual address space) at which it should be loaded. When the operating system is instructed to load a new program (with the exec(2) system call), 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.

At this point, the program is ready to execute, and the operating system transfers control to its entry point (which is also specified in the load module).

You might wonder, if there is no further relocation to be performed, why a load module might still contain a symbol table? Neither loading nor executing the program requires the symbol table. But if the program were to receive an exception (say at address 0x604C), the symbol table would enable us to determine that the error occurred twelve bytes into the _foo routine. Many load modules do not have any symbol tables (to save space, reduce download time, or make it harder for competitors to disassemble their code). Some load modules contain extensive symbol tables, including entry point addresses, data structure descriptions, source code line numbers, and other information to assist intelligent debuggers.

Static vs. Shared Libraries

In the above described linkage editing process, library modules were directly (and permanently) incorporated into the load module.

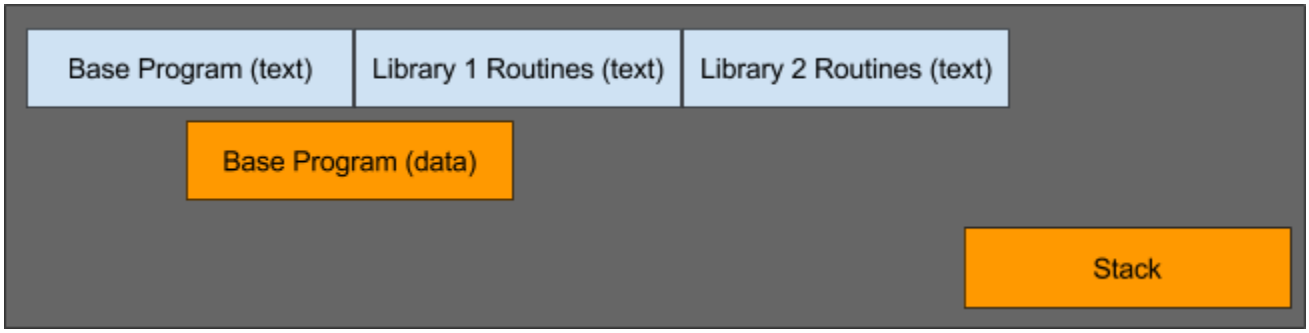


Fig 6. Virtual Address Space with Statically Linked Libraries

Because of this permanence, this process is referred to as static linking. It has at least two significant disadvantages:

- Many libraries (e.g. libc) are used by almost every program on the system. Thousands of identical copies of the same code increase the required down-load time, disk space (to store them), start-up time (to read them into memory) and memory (while they are executing). It would be much more efficient if we could somehow allow all of the programs that used a popular library to share a single copy.
- Popular libraries change over time with enhancements, optimizations, and bug fixes. Some enhancements may be very important (e.g. enabling applications to work with a new version of the operating system). In most cases, a newer version of a library is probably better than an older version. But with static linking, each program is built with a frozen version of each library, as it was at the time the program was linkage edited. It might be better (for software reliability) if it was possible to automatically load the latest library versions each time a program was started.

These issues are addressed by run-time loadable, shared libraries. There are many possible approaches, but a very simple way to implement shared libraries is to:

- Reserve an address for each shared library. This is possible in 32-bit architectures, and easy in 64-bit architectures.
- Linkage edit each shared library into a read-only code segment, 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 linkage 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. The stub library also includes symbol table information that informs the operating system what shared library segment this program requires.
- Linkage edit the client program with the stub library.
- When the operating system loads the program into memory, it will notice that the program requires shared libraries, and will open the associated (sharable, read-only) code segments and map them into the new program's address space at the appropriate location. This process is described in ld.so(8).

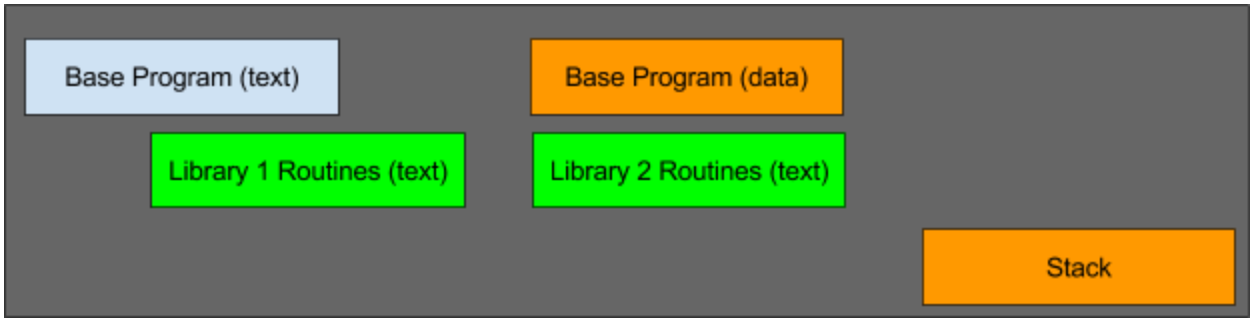


Fig 7. Virtual Address Space with Shared Libraries

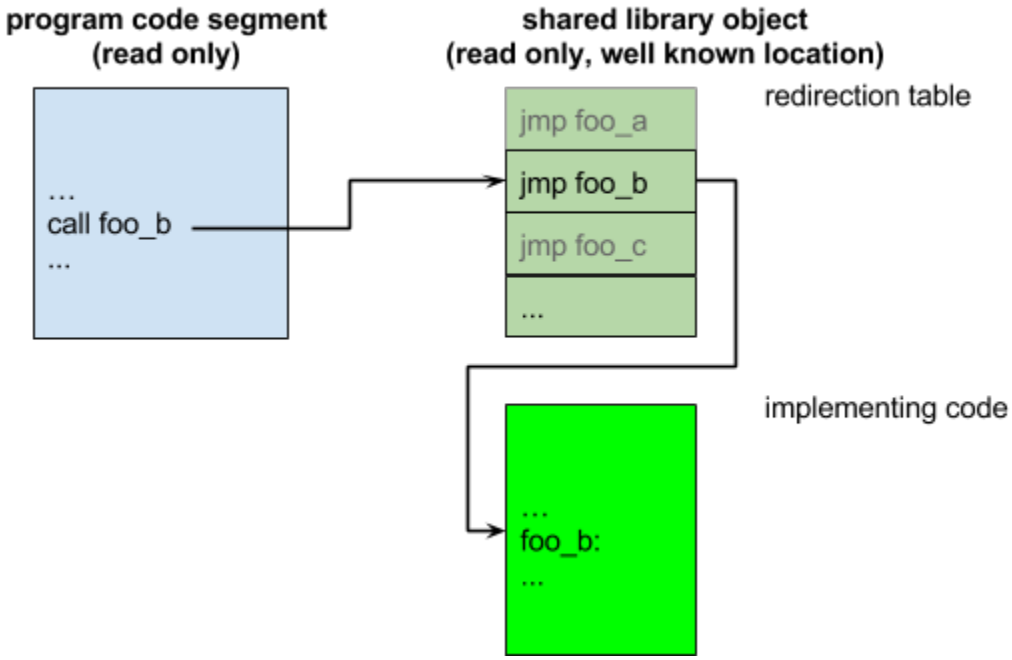


Fig 8. Linking Shared Libraries

In this way:

- A single copy of a shared library implementation (the shared segment) can be shared among all of the programs that use that library.
- 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.

Routines to be included in shared libraries must be designed with these limitations in mind. It may not be

possible to put arbitrary subroutines into a shared library.

Dynamically Loaded Libraries

Shared Libraries are very powerful and convenient, but they too have proved to be too limiting for many applications:

- There may be very large/expensive libraries that are seldom used; Loading/mapping such libraries into the process' address space at program load time unnecessarily slows down program start-up and increases the memory footprint. In some cases, it might be preferable to delay the loading of a module until it is actually needed.
- While loading is delayed until program load time, he 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.

These lead us to Dynamically Loadable Libraries (DLLs): libraries that are not loaded (or perhaps even chosen) until they are actually needed. Again, there are many implementations, but the general model is:

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

The Linux support for user mode Dynamically Loadable Libraries is described in dlopen(3). This sort of mechanism is used in user mode (e.g. to support different MIME times) as well as inside the operating system (to dynamically load device drivers or file system implementations). In either case it makes it possible for pre-compiled applications to exploit plug-ins to support functionality that was not implemented (or even envisioned) at the time the original application was built. This is very powerful.

Calls from the client application into a shared or Dynamically Loaded library are generally handled through a table or vector of entry points, much like object method invocation. There is much more diversity in the handling of calls from a library back into the application:

- An application can register a call-back routine by passing its address to an appropriate library registration method.
- The same approach can be generalized to a large number of functions by having the application call the library to register a vector of entry points to perform standard functions.
- Dynamically loaded kernel modules (e.g. device drivers and file systems) are allowed to access a myriad of functions and data structures within the kernel into which they have been loaded. This is often enabled by a run-time-loader that effectively linkage edits the newly loaded module against the operating system, filling in the addresses of all unresolved external references from the dynamically loaded module into the operating system. The Linux kernel run-time loader is described in ld.so(8).

Implicitly Loaded Dynamically Loadable Libraries

It should be noted that there are Dynamically Loadable Library implementations that are more similar to shared libraries:

- Applications are linkage edited against a set of stubs, which create Program Linkage Table (PLT) entries in the client load module.
- The PLT entries are initialized to point to calls to a Run-Time Loader.
- The first time one of these entry points is called, the stub calls the Run-Time Loader to open and load the appropriate Dynamically Loadable Library.
- After the required library has been loaded, the PLT entry is changed to point to the appropriate routine in the newly loaded library. From then on, all calls through that PLT entry go directly to the now-loaded routine.

Such implicitly loaded DLLs are (from the client perspective) almost indistinguishable from statically loaded libraries or shared libraries. From this perspective, both Dynamically Loadable and shared libraries can reduce the size of load modules (vs. statically linked libraries) and allow a single on-disk/in-memory code segment to be shared by multiple concurrently running programs.

The greater functionality and performance benefits of Dynamically Loadable Libraries are only available when the client applications become aware of them:

- Shared libraries allow delayed (program load time) binding to some version of a library that was chosen at linkage-edit time. Dynamically Loadable Libraries allow the library to be loaded to be chosen at run time.
- Shared libraries consume memory for the entire time that the process is running. Dynamically Loadable Libraries can be unloaded when they are no longer needed.
- Shared libraries impose numerous constraints and simple interfaces on the code they contain. Dynamically Loadable Libraries are capable of performing complex initialization and supporting complex (bi-directional) calls between the client application and library.
- Shared libraries are (from the client applications perspective) indistinguishable from statically loaded libraries, and very simple to use. Dynamically Loadable Libraries may require considerably more work (from the client application) to load the library, register the interfaces, and establish work sessions.

Shared Libraries are a more efficient mechanism for binding libraries to client applications. Dynamically Loadable Libraries are a mechanism to dynamically extend the functionality of a client application based on resources and information that may not be available until the moment they are needed.

Stack Frames and Linkage Conventions

Introduction

Two fundamental questions that quickly arise in the implementation of an operating system are:

1. what constitutes the state of a computation, and how can that state be saved and restored?
2. what are the mechanisms by which one software component can request services, and receive results from another?

We can begin the exploration of these questions by examining subroutine linkage conventions. Most students will have already been exposed to this in previous courses (e.g. computer architecture, programming languages, compiler construction); This brief review is intended to refresh your understanding of the basic concepts so that we can build upon them.

The Stack Model of Programming Languages

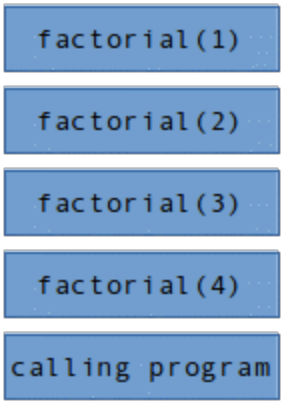
Most modern programming languages support procedure-local variables:

- 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.

These local variables, as well as parameters and intermediate computational results are most commonly stored in a Last-In-First-Out stack: with new call frames being pushed onto the stack whenever a procedure is called (or a block entered) and old frames being popped off the stack whenever a procedure returns (or a block is exited). Consider the recursive factorial implementation:

```
int factorial( int value ) {
        if (value <= 1)
                return(value);
        else
                return(value * factorial(value - 1));
}
```

If I call `factorial(4)`, just before the final invocation of `factorial` returns, there would be four instances of `factorial` on the stack, in addition to the call frames for the code that called `factorial` in the first place.



The stack model for procedure calls is so universal that most processor architectures provide hardware instructions for stack management, and depend on the presence of a stack to make subroutine calls or handle exceptions. It should be remembered, however, that the stack model does not work for all memory allocation needs. Many data items must last longer than the procedure that created them. For example, when read a document into memory, I do not expect that document to disappear as soon as the read function returns. Long lived resources should not be allocated on the stack, but rather from the *heap* (which we will discuss in a few weeks).

Subroutine Linkage Conventions

The details of subroutine linkage conventions are highly Instruction Set Architecture specific, and in some cases language-specific. The following examples will be based on the Intel x86 architecture. The instructions and registers will be very different for other ISAs, but the basic concepts will remain valid.

The basic elements of subroutine linkage are:

- parameter passing ... marshaling the information that will be passed as parameters to the called routine.
- subroutine call ... save the return address (in the calling routine) on the 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 (and perhaps computational temporaries) in the called routine.

When the called routine completes, the process of returning is fairly symmetric:

- 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.

The corresponding x86 code (for the above `factorial` example illustrates the register conventions and respective responsibilities of the caller and callee. The code in green is executed by *callees*. The code in red

is executed by *callers*.

```
_factorial:

        pushl   %ebp            // save previous frame pointer
        movl    %esp, %ebp      // top of stack becomes new frame pointer
        pushl   %ebx            // save non-volatile EBX register

        movl    8(%ebp), %ebx   // copy parameter off stack into EBX
        movl    $1, %eax        // get a constant 1
        cmpl    $1, %ebx        // compare value with 1
        jle     L2              // if less than or equal ...
        leal    -1(%ebx), %eax  // EAX = EBX - 1

        subl    $12, %esp       // extend stack for recursive call
        pushl   %eax            // push parameter onto stack
        call    _factorial      // call factorial
        addl    $12, %esp       // clean off the call frame

        imull   %ebx, %eax      // multiply value by return value
    L2:                         // return value is in eax

        movl    -4(%ebp), %ebx  // restore saved EBX register
        movl    %ebp, %esp      // restore top of stack
        popl    %ebp            // restore saved frame pointer
        ret                     // return to caller
```

X86 Register Conventions:

- `%esp` is the hardware-defined stack pointer.
- the X86 stack grows downwards. A push operation causes the top of stack to be the next lower address. A Pop operation causes the top of stack to be the next higher address.
- `%ebp` is typically used as a frame pointer ... it points at the start of the current stack frame (where the top of stack was at the time of entry).
- `%eax` is a volatile register that is expected to contain 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 return address off of the top of the stack and transfers back to that location.

While these details can be ISA specific, we also observe that:

- Register saving is the responsibility of the called routine. This is often done, because only the called routine knows which registers it will use (and therefore which it needs to save and restore.
- Cleaning parameters off of the stack is the responsibility of the calling routines. This is often done, because only the calling routine knows how many parameters is actually passed.
- The clear delineation of responsibilities between the caller and callee make it possible to have procedures written in one language (e.g. C) called by programs written in another language (e.g. FORTRAN).

There is little point in memorizing the detailed linkage conventions of an obsolete Instruction Set Architecture. There is still, however, value in understanding each step of the above process, because the same basic steps (some, perhaps automated by hardware) can be seen in almost all linkage conventions. After you have understood one set of linkage conventions, you should have little trouble understanding others.

It is also interesting to note how relatively simple it is to save and restore the state of a procedure; It is merely

a stack frame and a few registers ... most of whose values are stored in the next stack frame.

Traps and Interrupts

Most Instruction Set Architectures include support for interrupts (to inform the software than an external event has happened) and traps (to inform the software of an execution fault). These are similar to procedure calls in that:

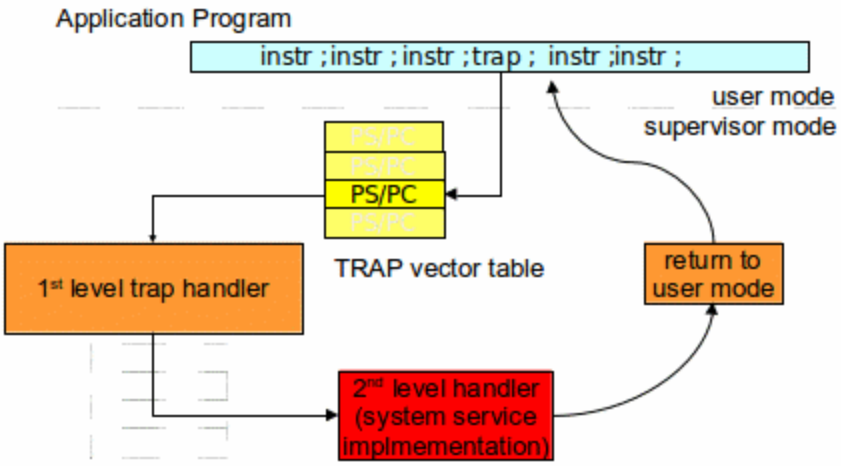
- we want to transfer control to an interrupt or trap handler.
- we need to save the state of the running computation before doing so.
- after the event has been handled, we want to restore the saved state and resume the interrupted computation.

The key differences between a procedure call and an interrupt/trap are:

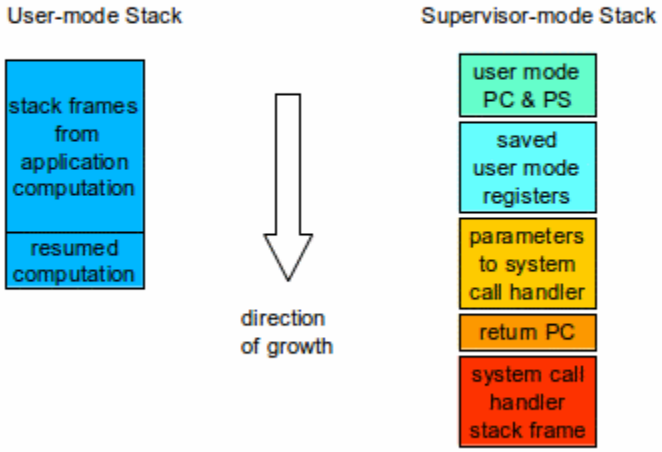
- a procedure call is requested by the running software, and the calling software expects that, upon return, some function will have been performed, and an appropriate value returned.
- because a procedure call is initiated by software, all of the linkage conventions are under software control. Because interrupts and traps are initiated by the 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 had never happened.

A typical interrupt or trap mechanism works as follows:

- there 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:
    1. the CPU uses the associated interrupt/trap number to index into the appropriate interrupt/trap vector table.
    2. the CPU loads a new program counter and processor status word from the appropriate interrupt/trap vector.
    3. 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).
    4. execution continues at the address specified by the new program counter.
    5. 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 ...
    1. the 1st level handler restores all of the saved registers (from the interrupted computation).
    2. the 1st level handler executes a privileged `return from interrupt` or `return from trap` instruction.
    3. the CPU re-loads the program counter and processor status word from the values saved at the time of interrupt/trap.
    4. execution resumes at the point of interruption.



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



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

This interrupt/trap mechanism:

- does a more complete 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 hander.

The similarities might make it seem that an interrupt/trap is only a little bit more expensive than a procedure call (more registers to save, plus the added computation to decide what 2nd level handler to call. But this ignores the fact that an interrupt/trap causes a new processor status word to be loaded into the CPU ... which will probably move us to a new (likely more privileged) processor mode, running in a new (unrelated to the

interrupted program) address space, and almost surely involve a complete loss of the CPU caches. And the same things will happen again upon return. Consequently, taking an interrupt or trap is likely to be somewhere between 100x and 1000x as expensive as making 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 introduction to the basic CPU-supported mechanisms for synchronous and 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.

In the following chapters we will build on these foundations to implement processes and system calls.

kill(2) — Linux manual page

NAME top

kill - send signal to a process

SYNOPSIS top

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
kill():
    _POSIX_C_SOURCE
```

DESCRIPTION top

The **kill**() system call can be used to send any signal to any process group or process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.

If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.

If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.

If *pid* is less than -1, then *sig* is sent to every process in the process group whose ID is *-pid*.

If *sig* is 0, then no signal is sent, but existence and permission

checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

For a process to have permission to send a signal, it must either be privileged (under Linux: have the **CAP_KILL** capability in the user namespace of the target process), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of **SIGCONT**, it suffices when the sending and receiving processes belong to the same session. (Historically, the rules were different; see NOTES.)

RETURN VALUE top

On success (at least one signal was sent), zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS top

EINVAL An invalid signal was specified.

EPERM The calling process does not have permission to send the signal to any of the target processes.

ESRCH The target process or process group does not exist. Note that an existing process might be a zombie, a process that has terminated execution, but has not yet been wait(2)ed for.

CONFORMING TO top

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

NOTES top

The only signals that can be sent to process ID 1, the *init* process, are those for which *init* has explicitly installed signal handlers. This is done to assure the system is not brought down accidentally.

POSIX.1 requires that *kill(-1,sig)* send *sig* to all processes that the calling process may send signals to, except possibly for some implementation-defined system processes. Linux allows a process to signal itself, but on Linux the call *kill(-1,sig)* does not signal the calling process.

POSIX.1 requires that if a process sends a signal to itself, and

the sending thread does not have the signal blocked, and no other thread has it unblocked or is waiting for it in sigwait(3), at least one unblocked signal must be delivered to the sending thread before the **kill**() returns.

Linux notes

Across different kernel versions, Linux has enforced different rules for the permissions required for an unprivileged process to send a signal to another process. In kernels 1.0 to 1.2.2, a signal could be sent if the effective user ID of the sender matched effective user ID of the target, or the real user ID of the sender matched the real user ID of the target. From kernel 1.2.3 until 1.3.77, a signal could be sent if the effective user ID of the sender matched either the real or effective user ID of the target. The current rules, which conform to POSIX.1, were adopted in kernel 1.3.78.

BUGS top

In 2.6 kernels up to and including 2.6.7, there was a bug that meant that when sending signals to a process group, **kill**() failed with the error **EPERM** if the caller did not have permission to send the signal to *any* (rather than *all*) of the members of the process group. Notwithstanding this error return, the signal was still delivered to all of the processes for which the caller had permission to signal.

SEE ALSO top

kill(1), _exit(2), pidfd_send_signal(2), signal(2), tkill(2), exit(3), killpg(3), sigqueue(3), capabilities(7), credentials(7), signal(7)

COLOPHON top

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Pages that refer to this page: capsh(1), fuser(1), kill(1@@coreutils), kill(1), kill(1@@procps-ng), killall(1), pgrep(1), skill(1), strace(1), clone(2), _exit(2), fcntl(2), getpid(2), getrlimit(2), pause(2), pidfd_open(2), pidfd_send_signal(2), ptrace(2), rt_sigqueueinfo(2), setfsgid(2), setfsuid(2), sigaction(2), signal(2), sigpending(2), sigprocmask(2), sigreturn(2), sigsuspend(2), sigwaitinfo(2), syscalls(2), tkill(2), wait(2), gsignal(3), killpg(3), psignal(3), pthread_kill(3), raise(3), sd_event_add_child(3), sigpause(3), sigqueue(3), sigset(3), sigvec(3), systemd.exec(5),

signal(2) — Linux manual page

NAME top

signal - ANSI C signal handling

SYNOPSIS top

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION top

WARNING: the behavior of **signal**() varies across UNIX versions, and has also varied historically across different versions of Linux. **Avoid its use**: use sigaction(2) instead. See *Portability* below.

signal() sets the disposition of the signal *signum* to *handler*, which is either **SIG_IGN**, **SIG_DFL**, or the address of a programmer-defined function (a "signal handler").

If the signal *signum* is delivered to the process, then one of the following happens:

* If the disposition is set to **SIG_IGN**, then the signal is ignored.

* If the disposition is set to **SIG_DFL**, then the default action associated with the signal (see signal(7)) occurs.

* If the disposition is set to a function, then first either the disposition is reset to **SIG_DFL**, or the signal is blocked (see *Portability* below), and then *handler* is called with argument

signum. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.

The signals **SIGKILL** and **SIGSTOP** cannot be caught or ignored.

RETURN VALUE top

signal() returns the previous value of the signal handler On failure, it returns **SIG_ERR**, and *errno* is set to indicate the error.

ERRORS top

EINVAL *signum* is invalid.

CONFORMING TO top

POSIX.1-2001, POSIX.1-2008, C89, C99.

NOTES top

The effects of **signal**() in a multithreaded process are unspecified.

According to POSIX, the behavior of a process is undefined after it ignores a **SIGFPE**, **SIGILL**, or **SIGSEGV** signal that was not generated by kill(2) or raise(3). Integer division by zero has undefined result. On some architectures it will generate a **SIGFPE** signal. (Also dividing the most negative integer by -1 may generate **SIGFPE**.) Ignoring this signal might lead to an endless loop.

See sigaction(2) for details on what happens when the disposition **SIGCHLD** is set to **SIG_IGN**.

See signal-safety(7) for a list of the async-signal-safe functions that can be safely called from inside a signal handler.

The use of *sighandler_t* is a GNU extension, exposed if **_GNU_SOURCE** is defined; glibc also defines (the BSD-derived) *sig_t* if **_BSD_SOURCE** (glibc 2.19 and earlier) or **_DEFAULT_SOURCE** (glibc 2.19 and later) is defined. Without use of such a type, the declaration of **signal**() is the somewhat harder to read:

```
void ( *signal(int signum, void (*handler)(int)) ) (int);
```

Portability

The only portable use of **signal**() is to set a signal's disposition to **SIG_DFL** or **SIG_IGN**. The semantics when using **signal**() to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); **do not use it for this purpose.**

POSIX.1 solved the portability mess by specifying sigaction(2), which provides explicit control of the semantics when a signal handler is invoked; use that interface instead of **signal**().

In the original UNIX systems, when a handler that was established using **signal**() was invoked by the delivery of a signal, the disposition of the signal would be reset to **SIG_DFL**, and the system did not block delivery of further instances of the signal. This is equivalent to calling sigaction(2) with the following flags:

```
    sa.sa_flags = SA_RESETHAND | SA_NODEFER;
```

System V also provides these semantics for **signal**(). This was bad because the signal might be delivered again before the handler had a chance to reestablish itself. Furthermore, rapid deliveries of the same signal could result in recursive invocations of the handler.

BSD improved on this situation, but unfortunately also changed the semantics of the existing **signal**() interface while doing so. On BSD, when a signal handler is invoked, the signal disposition is not reset, and further instances of the signal are blocked from being delivered while the handler is executing. Furthermore, certain blocking system calls are automatically restarted if interrupted by a signal handler (see signal(7)). The BSD semantics are equivalent to calling sigaction(2) with the following flags:

```
    sa.sa_flags = SA_RESTART;
```

The situation on Linux is as follows:

* The kernel's **signal**() system call provides System V semantics.

* By default, in glibc 2 and later, the **signal**() wrapper function does not invoke the kernel system call. Instead, it calls sigaction(2) using flags that supply BSD semantics. This default behavior is provided as long as a suitable feature test macro is defined: **_BSD_SOURCE** on glibc 2.19 and earlier or **_DEFAULT_SOURCE** in glibc 2.19 and later. (By default, these macros are defined; see feature_test_macros(7) for details.) If such a feature test macro is not defined, then **signal**() provides System V semantics.

SEE ALSO top

kill(1), alarm(2), kill(2), pause(2), sigaction(2), signalfd(2), sigpending(2), sigprocmask(2), sigsuspend(2), bsd_signal(3), killpg(3), raise(3), siginterrupt(3), sigqueue(3), sigsetops(3), sigvec(3), sysv_signal(3), signal(7)

COLOPHON top

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Pages that refer to this page: alarm(2), getitimer(2), kill(2), pause(2), prctl(2), sigaction(2), sigpending(2), sigprocmask(2), sigreturn(2), sigsuspend(2), sigwaitinfo(2), syscalls(2), wait(2), wait4(2), bsd_signal(3), gsignal(3), killpg(3), profil(3), raise(3), siginterrupt(3), sigqueue(3), sigset(3), sigvec(3), sleep(3), sysv_signal(3), systemd.exec(5), fifo(7), signal(7), signal-safety(7)

Real Time Scheduling

Introduction

Priority based scheduling enables us to give better service to certain processes. In our discussion of multi-queue scheduling, priority was adjusted based on whether a task was more interactive or compute intensive. But most schedulers enable us to give any process any desired priority. Isn't that good enough?

Priority scheduling is inherently a *best effort* approach. If our task is competing with other high priority tasks, it may not get as much time as it requires. Sometimes best effort isn't good enough:

- During reentry, the space shuttle is aerodynamically unstable. It is not actually being kept under control by the quick reflexes of the well-trained pilots, but rather by guidance computers that are collecting attitude and acceleration input and adjusting numerous spoilers hundreds of times per second.
- Scientific and military satellites may receive precious and irreplaceable sensor data at extremely high speeds. If it takes us too long to receive, process, and store one data frame, the next data frame may be lost.
- More mundanely, but also important, many manufacturing processes are run by computers nowadays. An assembly line needs to move at a particular speed, with each step being performed at a particular time. Performing the action too late results in a flawed or useless product.
- Even more commonly, playing media, like video or audio, has real time requirements. Sound must be produced at a certain rate and frames must be displayed frequently enough or the media becomes uncomfortable to deal with.

There are many computer controlled applications where delays in critical processing can have undesirable, or even disastrous consequences.

What are Real-Time Systems

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

When we discussed more traditional scheduling algorithms, the metrics we looked at were *turn-around time* (or throughput), *fairness*, and mean *response time*. But real-time systems have very different requirements, characterized by different 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

And we introduce a few new concepts:

- *feasibility* ... 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 (perhaps by as little as a fraction of a micro-second) may result in system failure.
- *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.

It sounds like real-time scheduling is more critical and difficult than traditional time-sharing, and in many

ways it is. But real-time systems may have a few characteristics that make scheduling 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. The space shuttle absolutely must sense attitude and acceleration and adjust spolier positions once per millisecond. But it probably doesn't matter if we update the navigational display once per millisecond or once every ten seconds. Telemetry transmission is probably somewhere in-between. 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.

Real-Time Scheduling Algorithms

In the simplest real-time systems, where the tasks and their execution times are all known, there might not even be a scheduler. One task might simply call (or yield to) the next. This model makes a great deal of sense in a system where the tasks form a producer/consumer pipeline (e.g. MPEG frame receipt, protocol decoding, image decompression, display).

In more complex real-time system, with a larger (but still fixed) number of tasks that do not function in a strictly pipeline fashion, it may be possible to do *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.

But for many real-time systems, the work-load changes from moment to moment, based on external events. These require *dynamic* scheduling. For *dynamic* scheduling algorithms, there are 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 may also be a different issue in real-time systems. In ordinary time-sharing, preemption is a means of improving mean response time by breaking up the execution of long-running, compute-intensive tasks. A second advantage of preemptive scheduling, particularly important in a general purpose timesharing system, is that it prevents a buggy (infinite loop) program from taking over the CPU. The trade-off, between improved response time and increased overhead (for the added context switches), almost always favors preemptive scheduling. This may not be true for real-time systems:

- 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 time systems, and the code is often much better tested ... so infinite loop bugs are extremely rare.

For the least demanding real time tasks, a sufficiently lightly loaded system might be reasonably successful in meeting its deadlines. However, this is achieved simply because the frequency at which the task is run happens to be high enough to meet its real time requirements, not because the scheduler is aware of such requirements. A lightly loaded machine running a traditional scheduler can often display a video to a user's satisfaction, not because the scheduler "knows" that a frame must be rendered by a certain deadline, but simply because the machine has enough cycles and a low enough work load to render the frame before the deadline has arrived.

Real-Time and Linux

Linux was not designed to be an embedded or real-time operating system, but many tasks that were once-considered embedded applications now require the capabilities (e.g. file systems, network protocols) of a general purpose operating system. As these requirements have increased and processors have gotten faster, increasingly many embedded and real-time applications have moved to Linux.

To support these applications Linux now supports a real-time scheduler, which can be enabled with sched_setscheduler(2), and is described in this Linux Journal Article. This real-time scheduler does not provide quite the same level of response-time guarantees that more traditional Real-Time-OSs do, but they are adequate for many soft real-time applications.

What about Windows? Conventional wisdom states that Windows is not well suited for real time needs, offering no native real time scheduler and too many ways in which desired real time deadlines might be missed. Windows favors general purpose throughput over meeting deadlines, as a rule. With sufficiently low load, a Windows system may, nonetheless, provide fast enough service for some soft real time requirements, such as playing music or video. One should be careful in relying on Windows for critical real time operations, however, as it is not designed for that purpose.

Garbage Collection and Defragmentation

Introduction

When we study resource allocation in an Operating Systems course, we focus on a few key resources (like space in main memory and secondary storage). But the issues and techniques for addressing those problems apply to a much wider range of divisible, serially reusable resources (e.g. farm land or appointments on a calendar), and so are a valuable foundation for all software developers. Two non-obvious techniques that have evolved (and been widely applied) over the last fifty years are:

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 fragmentation* to create densely allocated resources with large contiguous pools of free space

These two techniques are, in principle, orthogonal ... but we discuss them together for a few reasons:

- both are techniques for remedying space that has been rendered unusable by unfortunate combinations of allocation events.
- both are active resource management techniques, where the resource manager is continuously engaged in resource use analysis and reallocation (as opposed to simply responding to client requests).
- they are often applied in tandem ... start with garbage collection, and then perform defragmentation.

Garbage Collection

How does an allocated resource come to be returned to the free pool?

- With many resources, and in many programming languages, allocated resources are freed by an explicit or implicit action on the client's part. Example include:
  - calling *close(2)* on a file
  - calling *free(3)* on memory obtained from *malloc(3)*
  - invoking the *delete* operator on a C++ object
  - returning from a C/C++ subroutine
  - calling *exit(2)* to terminate a process

- If a resource is sharable by multiple concurrent clients (e.g. an open file or a shared memory segment), we cannot free it simply because one client called close. The resource manager (e.g. the operating system) may maintain a active reference count for each resource:
  - increment the reference count whenever a new reference to the object is obtained.
  - decrement the reference count whenever a reference is released (e.g. by calling close).
  - automatically free the object when the reference count reaches zero.

But these two approaches are not always practical:

- in languages where it is possible to copy resource references (e.g. pointers) references can be copied or destroyed without invoking any operating system services ... and so the OS will not know about these resource references.
- most programming languages seek to make the programmer's job easier, and keeping track of when which resources are no longer needed can be a lot of work. For this reason, many languages (e.g. Lisp,

Java, Python) do not require programs to explicitly free some resources (e.g. memory). This not only reduces developer task loading, but it also eliminates a great many bugs.
- some resources (e.g. DHCP addresses) may never be explicitly freed, and so must be automatically recycled after they have gone unused for a specified period of time.
- for some resources, allocation/release operations may be so frequent that the cost of keeping track of them becomes a significant overhead and performance loss.

In situations like these *Garbage Collection* is a popular alternative to explicit or reference count based freeing. In such a system ...

- 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 resources, is no longer referenced by the process, and can be freed.
    5. after freeing the unused resources, normal program operation can resume.

But Garbage Collection only works if it is possible to identify all active resource references. This might not be possible in an arbitrary program. We could scan all of memory searching for pointers into the allocation heap ... but when we find a heap address in memory, we cannot know if it is a pointer or merely a bit pattern that resembles a pointer. And even if it were clearly a pointer, we cannot know whether or not the program is still going to use it. In languages (or resources) that rely on Garbage Collection the data structures associated with resource references have been designed to be easily enumerated to enable the scan for accessible resources.

Garbage Collection definitely simplifies the developers job, but (like most things) it comes at a cost: The system may run very nicely for a long time, and then suddenly stop (at unpredictable times) for garbage collection. Clever developers have reduced this problem by creating more complex progressive background garbage collectors that run (relatively) continuously, so that continuous resource freeing can keep up with continuous resource allocation. And, even if background garbage collection is able to keep up with resource allocation demands, it often consumes a lot of cycles and must somehow be synchronized with ongoing allocation operations in the address space it is trying to analyze.

Defragmentation

The second problem to be addressed is *external fragmentation*. We observed that all variable partition memory allocation algorithms eventually result in a loss of useful storage due to external fragmentation ... the free memory is broken up into disconnected shards, none of which are large enough to be useful. In discussing those algorithms, we observed that coalescing of adjacent free fragments can counter external fragmentation; But coalescing is only effective if adjacent memory chunks happen to be free at the same time. If short-lived allocations are adjacent to a long-lived allocation, or if allocations and deallocations are extremely frequent, there may be very few such opportunities.

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 resources are allocated. Consider two applications of de-fragmentation:

- Flash Management

  NAND Flash is a pseudo-Write-Once-Read-Many medium. After a block has been written, some serious processing and voltage will be required to erase it before it can be rewritten with new data. When the operating system does a write to an SSD, firmware in the Flash Controller assigns a new block to receive the new data (leaving the old contents no longer referenced). As more and more 4K blocks become stale. Half of all the blocks in the SSD may be available for reuse, but the free blocks are distributed randomly throughout memory.

  To make those blocks writable again, they must first be erased. But erasure can only be done in large (e.g. 64MB units). Therefore we must:

  1. identify a 64MB block, many of whose 4K blocks are no longer in use.
  2. copy the 4K blocks that are still in use into a new (densely filled) 64MB block, and update the resource allocation maps accordingly.
  3. when no more valid data remains in the large block, it can be erased.
  4. once the large block has been erased, all of the 4K blocks within it can be added to the free list for reuse.

- Disk Space Allocation

  File reads and writes can progress orders of magnitude more quickly if logically consecutive blocks are stored contiguously on disk (so they can be read in a single operation with no head movement). But eventually, after many generations of files being created and deleted, the free space, and many files become discontiguous, and the file systems become slower and slower.

  To clean up the free space and restore our previous performance, we need to follow a process that may seem a little bit like the game *Towers of Hanoi*, where moving anything seems to (recursively) require moving everything else:

  1. choose an area of the disk where we want to create contiguous free space and files.
  2. for each file in that area, copy it to some other place, to free up space in the area we want to defragment.
  3. after all files have been copied out of that area, we can coalesce all the free space into one huge contiguous chunk.
  4. choose a set of files to be moved into the newly contiguous free space, and copy them into it.
  5. repeat this process until all of the files and free space are contiguous.

  A graphical illustration of this process can be seen in the Wikipedia article on Defragmentation

In neither of these examples is defragmentation a one-time process. Internal fragmentation (like rust) never sleeps! In the first (flash management) example the process is (like progressive garbage collection) a continuous one. In older file systems (e.g. Windows FAT file systems), the process was more likely periodic;

Once or twice a year we take the file system down to run defragmentation, after which performance should be good for a few more months. But in some newer file systems (e.g. the B-Tree File System) defragmentation is also a continuously ongoing process.

Conclusions

If you write much interesting software, you are likely to find yourself allocating and managing some sorts of divisible, serially reusable resources. Garbage Collection and Defragmentation are sophisticated resource allocation strategies. While the details of how to implement these strategies are highly dependent on the resources to be managed, the general concepts can be applied to a surprisingly wide range of resource allocation problems:

- If it does not make sense, or is not convenient to explicitly free resources, you should consider whether or not Garbage Collection would be a simpler or more efficient solution. If you want to use Garbage Collection, you must figure out how you will make all referenced resources discoverable, how you will trigger the scans, and how you will prevent race conditions between the Garbage Collector and your application.
- If your resource is subject to degradation or loss due to external fragmentation, you should consider whether or not periodic defragmentation can mitigate that process. If you want to use Defragmentation, you must figure out how you will drive the process, and how you will accomplish the required resource reallocation and copying without disrupting the running applications.

Working Sets

> **Annual income twenty pounds, annual expenditure nineteen** [pounds], **nineteen** [shillings] **and six** [pence] **... result happiness.**
> **Annual income twenty pounds, annual expenditure twenty pounds ought and six ... result misery.**
>
> *Wilkins Micawber (Charles Dickens)*

LRU is not enough

The success of LRU (Least Recently Used) replacement algorithms is the stuff of legends. Nobody knows what tomorrow will bring, but for most purposes our best guess is that the near future will be like the recent past. Unless you have inside information (i.e. references are not random, and you know understand the pattern) it is generally held that there is little profit to be gained from trying to out-smart LRU. The reason LRU works so well is that most programs exhibit some degree of temporal and spatial locality:

- if they use some code or data, they are likely to come back to access the same locations again.
- If they access code or data in a particular page, they are likely to reference other code (or data) in the same vicinity.

But these are statements about a single program or process. If we are doing Round-Robin time-sharing with multiple processes we give each process a brief opportunity to run, after which we run all the other processes before running it again. With the exception of shared text segments, separate processes seldom access the same pages.

- 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).

In the absence of temporal and spatial locality, Global LRU will make poor decisions. Global LRU and Round-Robin scheduling are great algorithms, but they do not work well as a team. 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 that might work very well with Round-Robin scheduling.

The concept of a Working Set

As we discovered in our discussion of paging:

- 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.

It is a good thing if a process experiences occasional page faults, and has to replace old (no longer interesting) pages with new ones. What we want to avoid is page faults due to running a process in too little memory. We want to keep the page-faults down to a manageable rate. The ideal mean-time-between-page-faults would be equal to the time-slice length. As we give a process fewer and fewer page frames in which to operate, the page fault rate rises and our performance gets worse (as we spend less time doing useful computation and more time processing page faults).

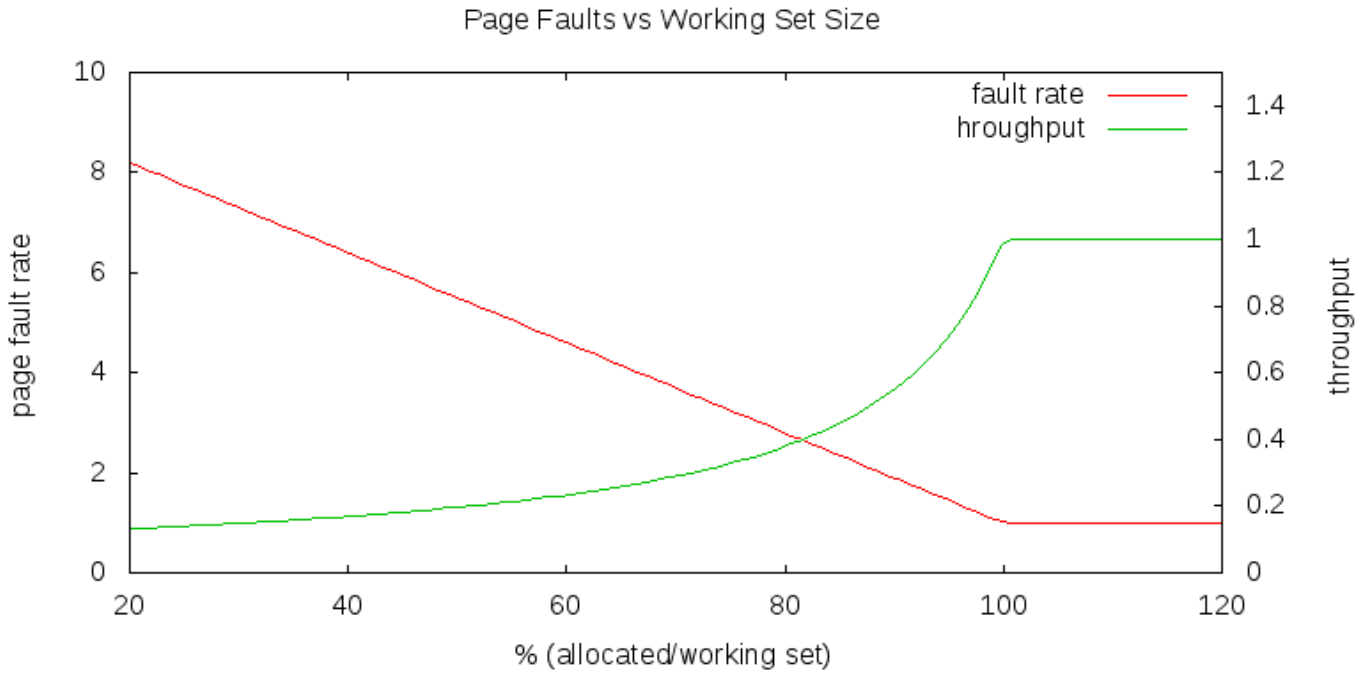
The dramatic degradation in system performance associated with not having enough memory to efficiently run all of the ready processes is called *thrashing*.

- If we think that we can have 25 processes in the ready queue, then we had better have enough memory for all 25 of those processes.
- If we only have enough memory to run 15 of those processes, then we should take the other 10 out of the ready queue (e.g. by swapping them out).
- The improved performance resulting from reducing the number of page faults will more than make up for the delays processes experience while being swapped between primary and secondary storage.

There is, for any given computation, 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.

We will call that number the process' *working set size* (at that particular time).



How large is a working set

Different computations require different amounts of memory. A tight loop operating on a few variables might need only two pages. Complex combinations of functions applied to large amounts of data could require hundreds of thousands of pages. Not only are different programs likely to require different amounts of memory, a single program may require different amounts of memory at different points during its execution.

Requiring more memory is not necessarily a bad thing ... it merely reflects the costs of that particular computation. When we explored scheduling we saw that different processes have different degrees of interactivity, and that if we could characterize the interactivity of each process we could more efficiently schedule it. The same is true of working set size. We can infer a process' working set size by observing its behavior. 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

Regularly scanning all of memory to identify the least recently used page is a very expensive process. With Global LRU, we saw that a clock-like scan was a very inexpensive way to achieve a very similar affect. 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 be referenced since the last time it has been scanned, it must be very old ... even if it is 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 process 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.

The new scan algorithm will be:

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

The key elements of this algorithm are:

- 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 to the rescue

This is often referred to as a *page stealing* algorithm, because a process that needs 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 referenced.
- 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 is a dynamic equilibrium mechanism. The continuously opposing processes of stealing and being stolen from will automatically allocate the available memory to the running processes in proportion to their working set sizes. It does not try to manage processes to any pre-configured notion of a reasonable working set size. Rather it manages memory to minimize the number of page faults, and to avoid thrashing.

User-Mode Thread Implementation

Introduction

For a long time, processes were the only unit of parallel computation. There are two problems with this:

- processes are very expensive to create and dispatch, due to the fact that each has its own virtual address space and ownership of numerous system resources.
- each process operates in its own address space, and cannot share in-memory resources with parallel processes (this was before it was possible to map shared segments into a process' address space).

The answer to these needs was to create threads. 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).

Threads were added to Unix/Linux as an afterthought. In Unix, a process could be scheduled for execution, and could create threads for additional parallelism. Windows NT is a newer operating system, and it was designed with threads from the start ... and so the abstractions are cleaner:

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

A Simple Threads Library

When threads were first added to Linux they were entirely implemented in a user-mode library, with no assistance from the operating system. This is not merely historical trivia, but interesting as an examination of what kinds of problems can and cannot be solved without operating system assistance.

The basic model is:

- Each time a new thread was created:
  - we allocate memory for a (fixed size) thread-private stack from the heap.
  - we create a new thread descriptor that contains identification information, scheduling information, and a pointer to the stack.
  - we add the new thread to a 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.

Eventually people wanted preemptive scheduling to ensure good interactive response and prevent buggy

threads from tying up the application:

- Linux processes can schedule the delivery of (SIGALARM) timer signals and register a handler for them.
- Before dispatching a thread, the 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.

But the addition of preemptive scheduling created new problems for critical sections that required before-or-after, all-or-none serialization. Fortunately Linux processes can temporarily block signals (much as it is possible to temporarily disable an interrupt) via the *sigprocmask(2)* system call.

Kernel implemented threads

There are two fundamental 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.

Both of these problems are solved if threads are implemented by the operating system rather than by a user-mode thread library.

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

Introduction

We can divide process interactions into a two broad categories:

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

The first of these are discussed in other readings and lectures. This is an introduction to the exchange of data between processes.

Simple Uni-Directional Byte Streams

These are easy to create and trivial to use. A pipe can be opened by a parent and inherited by a child, who simply reads standard input and writes standard output. Such pipelines can be used as part of a standard processing model:

```
macro-processor | compiler | assembler > output
```

or as custom constructions for one-off tasks:

```
find . -newer $TIMESTAMP | grep -v '*.o' | tar cfz archive.tgz -T -
```

All such uses have a few key characteristics in common:

- Each program accepts a byte-stream input, and produces a byte-stream output that is a well defined function of the input.
- Each program in the pipeline operates independently, and is unaware that the others exist or what they might do.
- Byte streams are inherently unstructured. If there is any structure to the data they carry (e.g. newline-delimited lines or comma-separated-values) these conventions are implemented by parsers in the affected applications.
- Ensuring that the output of one program is suitable input to the next is the responsibility of the agent who creates the pipeline.
- Similar results could be obtained by writing the output of each program into a (temporary) file, and then using that file as input to the next program.

Pipes are temporary files with a few special features, that recognize the difference between a file (whose contents are relatively static) and an inter-process data stream:

- If the reader exhausts all of the data in the pipe, but there is still an open write file descriptor, the reader does not get an *End of File*(EOF). Rather the reader is blocked until more data becomes available, or the write side is closed.

- The available buffering capacity of the pipe may be limited. If the writer gets too far ahead of the reader, the operating system may block the writer until the reader catches up. This is called *flow control*.
- Writing to a pipe that no longer has an open read file descriptor is illegal, and the writer will be sent an exception signal (SIGPIPE).
- When the read and write file descriptors are both closed, the file is automatically deleted.

Because a pipeline (in principle) represents a closed system, the only data privacy mechanisms tend to be the protections on the initial input files and final output files. There is generally no authentication or encryption of the data being passed between successive processes in the pipeline.

Named Pipes and Mailboxes

A named-pipe (*fifo(7)*) is a baby-step towards explicit connections. It can be thought of as a persistent pipe, whose reader and writer(s) can open it by name, rather than inheriting it from a *pipe(2)* system call. A normal pipe is custom-plumbed to interconnect processes started by a single user. A named pipe can be used as a rendezvous point for unrelated processes. Named pipes are almost as simple to use as ordinary pipes, but ...

- Readers and writers have no way of authenticating one-another's identities.
- Writes from multiple writers may be interspersed, with no indications of which bytes came from whom.
- They do not enable clean fail-overs from a failed reader to its successor.
- All readers and writers must be running on the same node.

Recognizing these limitations, some operating systems have crated more general inter-process communication mechanisms, often called *mailboxes*. While implementations differ, common features include:

- Data is not a byte-stream. Rather each write is stored and delivered as a distinct message.
- Each write is accompanied by authenticated identification information about its sender.
- Unprocessed messages remain in the mailbox after the death of a reader and can be retrieved by the next reader.

But mailboxes still subject to single node/single operating system restrictions, and most distributed applications are now based on general and widely standardized network protocols.

General Network Connections

Most operating systems now provide a fairly standard set of network communications APIs. The associated Linux APIs are:

- *socket(2)* ... create an inter-process communication end-point with an associated protocol and data model.
- *bind(2)* ... associate a *socket* with a local network address.
- *connect(2)* ... establish a connection to a remote network address.
- *listen(2)* ... await an incoming connection request.
- *accept(2)* ... accept an incoming connection request.
- *send(2)* ... send a message over a socket.
- *recv(2)* ... receive a message from a socket.

These APIs directly provide a range of different communications options:

- byte streams over reliable connections (e.g. TCP)
- best effort datagrams (e.g. UDP)

But they also form a foundation for higher level communication/service models. A few examples include:

- Remote Procedure Calls ... distributed request/response APIs.
- RESTful service models ... layered on top of HTTP GETs and PUTs.
- Publish/Subscribe services ... content based information flow.

Using more general networking models enables processes to interact with services all over the world, but this adds considerable complexity:

- Ensuring interoperability with software running under different operating systems on computers with different instruction set architectures.
- Dealing with the security issues associated with exchanging data and services with unknown systems over public networks.
- Discovering the addresses of (a constantly changing set of) servers.
- Detecting and recovering from (relatively common) connection and node failures.

Applications are forced to choose between a simple but strictly local model (pipes) or a general but highly complex model (network communications). But there is yet another issue: performance. Protocol stacks may be many layers deep, and data may be processed and copied many times. Network communication may have limited throughput, and high latencies.

Shared Memory

Sometimes performance is more important than generality.

- The network drivers, MPEG decoders, and video rendering in a set top box are guaranteed to be local. Making these operations more efficient can greatly reduce the required processing power, resulting in a smaller form-factor, reduced heat dissipation, and a lower product cost.
- The network drivers, protocol interpreters, write-back cache, RAID implementation, and back-end drivers in a storage array are guaranteed to be local. Significantly reducing the execution time per write operation is the difference between an industry leader and road-kill.

High performance for Inter-Process Communication means generally means:

- efficiency ... low cost (instructions, nano-seconds, watts) per byte transferred.
- throughput ... maximum number of bytes (or messages) per second that can be transferred.
- latency ... minimum delay between sender write and receiver read.

If we want ultra high performance Inter-Process Communication between two local processes, buffering the data through the operating system and/or protocol stacks is not the way to get it. The fastest and most efficient way to move data between processes is through 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 that it is never paged out.
- the communicating processes agree on a set of data structures (e.g. polled lock-free circular buffers) in

the shared segment.
- anything written into the shared memory segment will be immediately visible to all of the processes that have it mapped in to their address spaces.

Once the shared segment has been created and mapped into the participating process' address spaces, the operating system plays no role in the subsequent data exchanges. Moving data in this way is extremely efficient and blindingly fast ... but (like all good things) this performance comes at a price:

- This can only be used between processes on the same memory bus.
- A bug in one of the processes can easily destroy the communications data structures.
- There is no authentication (beyond access control on the shared file) of which data came from which process.

Network Connections and Out-of-Band Signals

In most cases, event completions can be reported simply by sending a message (announcing the completion) to the waiter. But what if there are megabytes of queued requests, and we want to send a message to abort those queued requests? Data sent down a network connection is FIFO ... and one of the problems with FIFO scheduling is the delays waiting for the processing of earlier but longer messages. Occasionally, we would like to make it possible for an important message to go directly to front of the line.

If the recipient was local, we might consider sending a signal that could invoke a registered handler, and flush (without processing) all of the buffered data. This works because the signals travel over a different channel than the buffered data. Such communication is often called *out-of-band*, because it does not travel over the normal data path.

We can achieve a similar effect with network based services by opening multiple communications channels:

- one heavily used channel for normal requests
- another reserved for out-of-band requests

The server on the far end periodically polls the out-of-band channel before taking requests from the normal communications channel. This adds a little overhead to the processing, but makes it possible to preempt queued operations. The chosen polling interval represents a trade-off between added overhead (to check for out-of-band messages) and how long we might go (how much wasted work we might do) before noticing an out-of-band message.

flock(2) — Linux manual page

NAME top

flock - apply or remove an advisory lock on an open file

SYNOPSIS top

```
#include <sys/file.h>

int flock(int fd, int operation);
```

DESCRIPTION top

Apply or remove an advisory lock on the open file specified by *fd*. The argument *operation* is one of the following:

LOCK_SH Place a shared lock. More than one process may hold a shared lock for a given file at a given time.

LOCK_EX Place an exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

LOCK_UN Remove an existing lock held by this process.

A call to **flock**() may block if an incompatible lock is held by another process. To make a nonblocking request, include **LOCK_NB** (by ORing) with any of the above operations.

A single file may not simultaneously have both shared and exclusive locks.

Locks created by **flock**() are associated with an open file description (see open(2)). This means that duplicate file descriptors (created by, for example, fork(2) or dup(2)) refer to the same lock, and this lock may be modified or released using any of these file descriptors. Furthermore, the lock is released

either by an explicit **LOCK_UN** operation on any of these duplicate file descriptors, or when all such file descriptors have been closed.

If a process uses open(2) (or similar) to obtain more than one file descriptor for the same file, these file descriptors are treated independently by **flock**(). An attempt to lock the file using one of these file descriptors may be denied by a lock that the calling process has already placed via another file descriptor.

A process may hold only one type of lock (shared or exclusive) on a file. Subsequent **flock**() calls on an already locked file will convert an existing lock to the new lock mode.

Locks created by **flock**() are preserved across an execve(2).

A shared or exclusive lock can be placed on a file regardless of the mode in which the file was opened.

RETURN VALUE top

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS top

EBADF *fd* is not an open file descriptor.

EINTR While waiting to acquire a lock, the call was interrupted by delivery of a signal caught by a handler; see signal(7).

EINVAL *operation* is invalid.

ENOLCK The kernel ran out of memory for allocating lock records.

EWOULDBLOCK
The file is locked and the **LOCK_NB** flag was selected.

CONFORMING TO top

4.4BSD (the **flock**() call first appeared in 4.2BSD). A version of **flock**(), possibly implemented in terms of fcntl(2), appears on most UNIX systems.

NOTES top

Since kernel 2.0, **flock**() is implemented as a system call in its own right rather than being emulated in the GNU C library as a call to fcntl(2). With this implementation, there is no interaction between the types of lock placed by **flock**() and fcntl(2), and **flock**() does not detect deadlock. (Note, however, that on some systems, such as the modern BSDs, **flock**() and fcntl(2) locks *do* interact with one another.)

flock() places advisory locks only; given suitable permissions on a file, a process is free to ignore the use of **flock**() and perform I/O on the file.

flock() and fcntl(2) locks have different semantics with respect to forked processes and dup(2). On systems that implement **flock**() using fcntl(2), the semantics of **flock**() will be different from those described in this manual page.

Converting a lock (shared to exclusive, or vice versa) is not guaranteed to be atomic: the existing lock is first removed, and then a new lock is established. Between these two steps, a pending lock request by another process may be granted, with the result that the conversion either blocks, or fails if **LOCK_NB** was specified. (This is the original BSD behavior, and occurs on many other implementations.)

NFS details

In Linux kernels up to 2.6.11, **flock**() does not lock files over NFS (i.e., the scope of locks was limited to the local system). Instead, one could use fcntl(2) byte-range locking, which does work over NFS, given a sufficiently recent version of Linux and a server which supports locking.

Since Linux 2.6.12, NFS clients support **flock**() locks by emulating them as fcntl(2) byte-range locks on the entire file. This means that fcntl(2) and **flock**() locks *do* interact with one another over NFS. It also means that in order to place an exclusive lock, the file must be opened for writing.

Since Linux 2.6.37, the kernel supports a compatibility mode that allows **flock**() locks (and also fcntl(2) byte region locks) to be treated as local; see the discussion of the *local_lock* option in nfs(5).

CIFS details

In Linux kernels up to 5.4, **flock**() is not propagated over SMB. A file with such locks will not appear locked for remote clients.

Since Linux 5.5, **flock**() locks are emulated with SMB byte-range locks on the entire file. Similarly to NFS, this means that fcntl(2) and **flock**() locks interact with one another. Another important side-effect is that the locks are not advisory anymore: any IO on a locked file will always fail with **EACCES** when done

from a separate file descriptor. This difference originates from the design of locks in the SMB protocol, which provides mandatory locking semantics.

Remote and mandatory locking semantics may vary with SMB protocol, mount options and server type. See **mount.cifs**(8) for additional information.

SEE ALSO top

flock(1), close(2), dup(2), execve(2), fcntl(2), fork(2), open(2), lockf(3), lslocks(8)

Documentation/filesystems/locks.txt in the Linux kernel source tree (*Documentation/locks.txt* in older kernels)

COLOPHON top

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Pages that refer to this page: flock(1), chown(2), fcntl(2), fork(2), getrlimit(2), syscalls(2), dbopen(3), flockfile(3), lockf(3), nfs(5), proc(5), tmpfiles.d(5), signal(7), cryptsetup(8), fsck(8), lslocks(8)

lockf(3) — Linux manual page

NAME top

lockf - apply, test or remove a POSIX lock on an open file

SYNOPSIS top

```
#include <unistd.h>

int lockf(int fd, int cmd, off_t len);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
lockf():
    _XOPEN_SOURCE >= 500
        || /* Glibc since 2.19: */ _DEFAULT_SOURCE
        || /* Glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION top

Apply, test, or remove a POSIX lock on a section of an open file. The file is specified by *fd*, a file descriptor open for writing, the action by *cmd*, and the section consists of byte positions *pos*..*pos*+*len*-1 if *len* is positive, and *pos*-*len*..*pos*-1 if *len* is negative, where *pos* is the current file position, and if *len* is zero, the section extends from the current file position to infinity, encompassing the present and future end-of-file positions. In all cases, the section may extend past current end-of-file.

On Linux, **lockf**() is just an interface on top of fcntl(2) locking. Many other systems implement **lockf**() in this way, but note that POSIX.1 leaves the relationship between **lockf**() and fcntl(2) locks unspecified. A portable application should probably avoid mixing calls to these interfaces.

Valid operations are given below:

F_LOCK Set an exclusive lock on the specified section of the file. If (part of) this section is already locked, the call blocks until the previous lock is released. If this section overlaps an earlier locked section, both are merged. File locks are released as soon as the process holding the locks closes some file descriptor for the file. A child process does not inherit these locks.

F_TLOCK
Same as **F_LOCK** but the call never blocks and returns an error instead if the file is already locked.

F_ULOCK
Unlock the indicated section of the file. This may cause a locked section to be split into two locked sections.

F_TEST Test the lock: return 0 if the specified section is unlocked or locked by this process; return -1, set *errno* to **EAGAIN** (**EACCES** on some other systems), if another process holds a lock.

RETURN VALUE top

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS top

EACCES or **EAGAIN**
The file is locked and **F_TLOCK** or **F_TEST** was specified, or the operation is prohibited because the file has been memory-mapped by another process.

EBADF *fd* is not an open file descriptor; or *cmd* is **F_LOCK** or **F_TLOCK** and *fd* is not a writable file descriptor.

EDEADLK
The command was **F_LOCK** and this lock operation would cause a deadlock.

EINTR While waiting to acquire a lock, the call was interrupted by delivery of a signal caught by a handler; see signal(7).

EINVAL An invalid operation was specified in *cmd*.

ENOLCK Too many segment locks open, lock table is full.

ATTRIBUTES top

For an explanation of the terms used in this section, see attributes(7).

Interface	Attribute	Value
lockf()	Thread safety	MT-Safe

CONFORMING TO top

POSIX.1-2001, POSIX.1-2008, SVr4.

SEE ALSO top

fcntl(2), flock(2)

locks.txt and *mandatory-locking.txt* in the Linux kernel source directory *Documentation/filesystems* (on older kernels, these files are directly under the *Documentation* directory, and *mandatory-locking.txt* is called *mandatory.txt*)

COLOPHON top

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Pages that refer to this page: fcntl(2), flock(2), flockfile(3), system_data_types(7), lslocks(8)

Deadlock Avoidance

Introduction

We focus a great deal of attention on the four necessary conditions for deadlock, because these form the basis for deadlock prevention. If we can ensure that any of the four necessary conditions will never be met, we have created a system where deadlock (involving those resources) is impossible. There are, however, common situations in which:

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

For many such cases, deadlock avoidance may be an easy and effective solution. Consider the following deadlock situation:

- main memory is exhausted.
- we need to swap some processes out to secondary storage to free up memory.
- swapping processes out involves the creation of new I/O requests descriptors, which must be allocated from main memory.

The problem, in this case, is not a particular resource dependency graph, but merely that we exhausted a critical resource. There are many similar situations where:

- some process will free up resources when it completes.
- but the process needs more resources in order to complete.

In situations like these, it is common to keep track of free resources, and refuse to grant requests that would put the system into a dangerously resource-depleted state. Making such case-by-case decisions to avoid deadlocks is called *deadlock avoidance*.

Reservations

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.

Consider the *sbrk(2)* system call. It does not actually allocate any more memory to the process. It requests the operating system to change the size of the data segment in the process' virtual address space. The actual memory assignments will not happen until the process begins referencing those newly authorized pages. If we can determine that the requested reservation would over-tax memory, we can return an error from the *sbrk* system call, which the process can then decide how to handle. If, however, we waited until a page was referenced before we decided we did not have sufficient memory, we might have no alternative but to kill the process.

This approach is not limited to memory. For example ...

- we could refuse to create new files when file system space gets 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.

Unlike *malloc(3)*, these other operations do not tell us how much new resource the process wants to consume. But in each of these cases there is a request (to which we can return an error) before we reach actual resource exhaustion. And it is this failable request that gives us the opportunity to consider, and avoid a resource-exhaustion deadlock.

Over-Booking

In most situations, it is unlikely that all clients will simultaneously request their maximum resource reservations. For this reason, it is often considered *relatively safe* to grant somewhat more reservations than 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 might a demand that we cannot gracefully handle.

- Air lines do this all the time ... which is why they occasionally offer cash and free trips to anyone who is willing/able to take a later flight.
- Required network bandwidth is routinely estimated based on the expected traffic distribution. Such a network might be able to handle 25% more traffic while still maintaining its Service Level Agreements 99.9% of the time.
- In operating systems, the notion of killing random processes is so abhorrent that most operating systems simply refuse to over-book. In fact, it is common to under-book (e.g. reserve the last 10% for emergency/super-user use).

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 that the problem is transient).
- A more robust program might return errors for requests 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).

There are many possible responses, and different responses make sense in different situations. But the key here is that, since the allocation request failed with a clean error, the process has the opportunity to try to manage the situation in the most graceful possible way.

Health Monitoring and Recovery

Introduction

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 would we determine that the system might be wedged, so that we could invoke deadlock analysis?

It may not be possible to identify the owners of all of the involved resources, or even all of the resources.

Worse still, a process may not actually be blocked, but merely waiting for a message or event (that has, for some reason, not yet been sent).

If we did determine that a deadlock existed, what would we do? Kill a random process? This might break the circular dependency, but would the system continue to function properly after such an action?

Formal deadlock detection in real systems ...

a. is difficult to perform
b. is inadequate to diagnose most hangs
c. does not enable us to fix the problem

Fortunately there are better techniques that are far more effective at detecting, diagnosing, and repairing a much wider range of problems: health monitoring and managed recovery.

Health Monitoring

We said that we could invoke deadlock detection whenever we thought that the system might not be making progress. How could we know whether or not the system was making progress? There are many ways to do this:

- 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.

Any of these techniques could alert us of a potential deadlock, livelock, loop, or a wide range of other failures. But each of these techniques has different strengths and weaknesses:

- 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.

Many systems use a combination of these methods:

- the first line of defense is an internal monitoring agent that closely watches key applications to detect failures and hangs.
- if the internal monitoring agent is responsible for sending heart-beats (or health status reports) to a central monitoring agent, a failure of the internal monitoring agent will be noticed by the central monitoring agent.
- an external test service that periodically generates test transactions provides an independent assessment that might include external factors (e.g. switches, load balancers, network connectivity) that would not be tested by the internal and central monitoring services.

Managed Recovery

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

- The software should be designed so that 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.
- The software should be designed to support multiple levels of restart. Examples might be:
    - warm-start ... restore the last saved state (from a database or from information obtained from other processes) and resume service where we left off.
    - cold-start ... ignore any saved state (which may be corrupted) and restart new operations from scratch.
    - reset and reboot ... reboot the entire system and then cold-start all of the applications.
- The software might also designed be for progressively escalating scope 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.

Designing software in this way gives us the opportunity to begin with minimal disruption, restarting only the process that seems to have failed. In most cases this will solve the problem, but perhaps:

- process A failed as a result of an incorrect request received from process B.
- the operation that caused process A to fail is still listed in the database, and when process A restarts, it may attempt to re-try the same operation and fail again.
- the operation that caused process A to fail may have been mirrored to other systems, that will also experience or cause additional failures.

For all of these reasons it is desirable to have the ability to escalate to progressively more complete restarts of a progressively wider range of components.

False Reports

Ideally a problem will be found by the internal monitoring agent on the affected node, which will automatically trigger a restart of the affected software on that node. Such prompt local action has the potential to fix the problem before other nodes even notice that there was a problem.

But suppose a central monitoring service notes that it has not received a heart-beat from process A. What might this mean?

- It might mean that process A's node has failed.
- It might mean that the process A has failed.
- It might mean that the process A's system is loaded and the heart-beat message was delayed.
- It might mean that a network error prevented or delayed the delivery of a heart-beat message.
- It might mean there is a problem with the central monitoring service.

Declaring a process to have failed can potentially 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. It might cause other servers to start trying to recover work-in-progress from the failed process or node. This recovery might involve a great deal of network traffic and system activity. We don't want to start an expensive fire-drill unless we are pretty certain that a process has 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.

There is a trade-off here:

- If we do not take the time to confirm suspected failures, we may suffer unnecessary service disruptions from forcing unnecessary fail-overs from healthy servers.
- If we mis-diagnose the cause of the problem and restart the wrong components we may make the problem even worse.
- If we wait too long before initiating fail-overs, we are prolonging the service outage.

These so-called "mark-out thresholds" often require a great deal of tuning. Many systems evolve complex decision algorithms to filter and reconcile potentially conflicting reports to attempt to infer what the most likely cause of of a problem is, and the most effective means of dealing with it.

Other Managed Restarts

As we consider failure and restart, there are two other interesting types of restart to note:

non-disruptive rolling upgrades

If a system is capable of operating without some of its nodes, it is possible to achieve non-disruptive

rolling software upgrades. We take nodes down, one-at-a-time, upgrade each to a new software release, and then reintegrate them into the service. There are two tricks associated with this:

- the new software must be up-wards compatible with the old software, so that new nodes can interoperate with old ones.
- if the rolling upgrade does not seem to be working, there needs to be an automatic *fall-back* option to return to the previous (working) release.

prophylactic reboots

It has long been observed that many software systems become slower and more error prone the longer they run. The most common problem is memory leaks, but there are other types of bugs that can cause software systems to degrade over time. The right solution is probably to find and fix the bugs ... but many organizations seem unable to do this. One popular alternative is to automatically restart every system at a regular interval (e.g. a few hours or days).

If a system can continue operating in the face of node failures, it should be fairly simple to shut-down and restart nodes one at a time, on a regular schedule.

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: Synchronization

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Synchronized Methods

The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*. The more complex of the two, synchronized statements, are described in the next section. This section is about 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;
    }
}
```

If `count` is an instance of `SynchronizedCounter`, then making these methods synchronized has two effects:

- 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 (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a `List` called `instances` containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

But then other threads can use `instances` to access the object before construction of the object is complete.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through `synchronized` methods. (An important exception: `final` fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with liveness, as we'll see later in this lesson.

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: Synchronization

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic 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

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

In this example, the `addName` method needs to synchronize changes to `lastName` and `nameCount`, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking `nameList.add`.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class `MsLunch` has two instance fields, `c1` and `c2`, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with `this`, we create two objects solely to provide locks.

```
public class MsLunch {
    private long c1 = 0;
```

```
        private long c2 = 0;
        private Object lock1 = new Object();
        private Object lock2 = new Object();

        public void inc1() {
            synchronized(lock1) {
                c1++;
            }
        }

        public void inc2() {
            synchronized(lock2) {
                c2++;
            }
        }
    }
```

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

Previous page: Synchronized Methods
Next page: Atomic Access

Monitor (synchronization)

In concurrent programming, 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 false. Monitors also have a mechanism for signaling other threads that their condition has been met. A monitor consists of a mutex (lock) object and **condition variables**. A **condition variable** essentially is a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

Another definition of **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. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: At each point in time, at most one thread may be executing any of its methods. By using one or more condition variables it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor"). For the rest of this article, this sense of "monitor" will be referred to as a "thread-safe object/class/module".

Monitors were invented by Per Brinch Hansen[1] and C. A. R. Hoare,[2] and were first implemented in Brinch Hansen's Concurrent Pascal language.[3]

Mutual exclusion

As a simple example, consider a thread-safe object for performing transactions on a bank account:

```
monitor class Account {
    private int balance := 0
    invariant balance >= 0

    public method boolean withdraw(int amount)
        precondition amount >= 0
    {
        if balance < amount {
            return false
        } else {
            balance := balance - amount
            return true
        }
    }

    public method deposit(int amount)
        precondition amount >= 0
    {
        balance := balance + amount
    }
}
```

While a thread is executing a method of a thread-safe object, it is said to *occupy* the object, by holding its mutex (lock). Thread-safe objects are implemented to enforce that *at each point in time, at most one thread may occupy the object*. The lock, which is initially unlocked, is locked at the start of each public method, and is unlocked at each return from each public method.

Upon calling one of the methods, a thread must wait until no other thread is executing any of the thread-safe object's methods before starting execution of its method. Note that without this mutual exclusion, in the present example, two threads could cause money to be lost or gained for no reason. For example, two threads withdrawing 1000 from the account could both return true, while causing the balance to drop by only 1000, as follows: first, both threads fetch the current balance, find it greater than 1000, and subtract 1000 from it; then, both threads store the balance and return.

The syntactic sugar "monitor class" in the above example is implementing the following basic representation of the code, by wrapping each function's execution in mutexes:

```
class Account {
    private lock myLock

    private int balance := 0
    invariant balance >= 0

    public method boolean withdraw(int amount)
       precondition amount >= 0
    {
        myLock.acquire()
        try {
            if balance < amount {
                return false
            } else {
                balance := balance - amount
                return true
            }
        } finally {
            myLock.release()
        }
    }

    public method deposit(int amount)
       precondition amount >= 0
    {
        myLock.acquire()
        try {
            balance := balance + amount
        } finally {
            myLock.release()
        }
    }
}
```

Condition variables

Problem statement

For many applications, mutual exclusion is not enough. Threads attempting an operation may need to wait until some condition *P* holds true. A busy waiting loop

```
while not ( P ) do skip
```

will not work, as mutual exclusion will prevent any other thread from entering the monitor to make the condition true. Other "solutions" exist such as having a loop that unlocks the monitor, waits a certain amount of time, locks the monitor and checks for the condition *P*. Theoretically, it works and will not deadlock, but issues arise. It is hard to decide an appropriate amount of waiting time:

too small and the thread will hog the CPU, too big and it will be apparently unresponsive. What is needed is a way to signal the thread when the condition *P* is true (or *could* be true).

Case study: classic bounded producer/consumer problem

A classic concurrency problem is that of the **bounded producer/consumer**, in which there is a queue or ring buffer of tasks with a maximum size, with one or more threads being "producer" threads that add tasks to the queue, and one or more other threads being "consumer" threads that take tasks out of the queue. The queue is assumed to be non–thread-safe itself, and it can be empty, full, or between empty and full. Whenever the queue is full of tasks, then we need the producer threads to block until 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.

As the queue is a concurrent object shared between threads, accesses to it must be made atomic, because the queue can be put into an **inconsistent state** during the course of the queue access that should never be exposed between threads. Thus, any code that accesses the queue constitutes a **critical section** that must be synchronized by mutual exclusion. If code and processor instructions in critical sections of code that access the queue could be **interleaved** by arbitrary **context switches** between threads on the same processor or by simultaneously-running threads on multiple processors, then there is a risk of exposing inconsistent state and causing race conditions.

Incorrect without synchronization

A naïve approach is to design the code with **busy-waiting** and no synchronization, making the code subject to race conditions:

```
global RingBuffer queue; // A thread-unsafe ring-buffer of tasks.

// Method representing each producer thread's behavior:
public method producer() {
    while (true) {
        task myTask = ...; // Producer makes some new task to be added.
        while (queue.isFull()) {} // Busy-wait until the queue is non-full.
        queue.enqueue(myTask); // Add the task to the queue.
    }
}

// Method representing each consumer thread's behavior:
public method consumer() {
    while (true) {
        while (queue.isEmpty()) {} // Busy-wait until the queue is non-empty.
        myTask = queue.dequeue(); // Take a task off of the queue.
        doStuff(myTask); // Go off and do something with the task.
    }
}
```

This code has a serious problem in that accesses to the queue can be interrupted and interleaved with other threads' accesses to the queue. The *queue.enqueue* and *queue.dequeue* methods likely have instructions to update the queue's member variables such as its size, beginning and ending positions, assignment and allocation of queue elements, etc. In addition, the *queue.isEmpty()* and *queue.isFull()* methods read this shared state as well. If producer/consumer threads are allowed to be interleaved during the calls to enqueue/dequeue, then inconsistent state of the queue can be exposed leading to race conditions. In addition, if one consumer makes the queue empty in-between

another consumer's exiting the busy-wait and calling "dequeue", then the second consumer will attempt to dequeue from an empty queue leading to an error. Likewise, if a producer makes the queue full in-between another producer's exiting the busy-wait and calling "enqueue", then the second producer will attempt to add to a full queue leading to an error.

Spin-waiting

One naive approach to achieve synchronization, as alluded to above, is to use "**spin-waiting**", in which a mutex is used to protect the critical sections of code and busy-waiting is still used, with the lock being acquired and released in between each busy-wait check.

```
global RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks.

// Method representing each producer thread's behavior:
public method producer() {
    while (true) {
        task myTask = ...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial busy-wait check.
        while (queue.isFull()) { // Busy-wait until the queue is non-full.
            queueLock.release();
            // Drop the lock temporarily to allow a chance for other threads
            // needing queueLock to run so that a consumer might take a task.
            queueLock.acquire(); // Re-acquire the lock for the next call to "queue.isFull()".
        }

        queue.enqueue(myTask); // Add the task to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
    }
}

// Method representing each consumer thread's behavior:
public method consumer() {
    while (true) {
        queueLock.acquire(); // Acquire lock for initial busy-wait check.
        while (queue.isEmpty()) { // Busy-wait until the queue is non-empty.
            queueLock.release();
            // Drop the lock temporarily to allow a chance for other threads
            // needing queueLock to run so that a producer might add a task.
            queueLock.acquire(); // Re-acquire the lock for the next call to "queue.isEmpty()".
        }
        myTask = queue.dequeue(); // Take a task off of the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.
        doStuff(myTask); // Go off and do something with the task.
    }
}
```

This method assures that an inconsistent state does not occur, but wastes CPU resources due to the unnecessary busy-waiting. Even if the queue is empty and producer threads have nothing to add for a long time, consumer threads are always busy-waiting unnecessarily. Likewise, even if consumers are blocked for a long time on processing their current tasks and the queue is full, producers are always busy-waiting. This is a wasteful mechanism. What is needed is a way to make producer threads block until the queue is non-full, and a way to make consumer threads block until the queue is non-empty.

(N.B.: Mutexes themselves can also be **spin-locks** which involve busy-waiting in order to get the lock, but in order to solve this problem of wasted CPU resources, we assume that *queueLock* is not a spin-lock and properly uses a blocking lock queue itself.)

Condition variables

The solution is to use **condition variables**. Conceptually a condition variable is a queue of threads, associated with a mutex, on which a thread may wait for some condition to become true. Thus each condition variable *c* is associated with an assertion P_c. While 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. In most types of monitors, these other threads may signal the condition variable *c* to indicate that assertion P_c is true in the current state.

Thus there are three main operations on condition variables:

- **`wait c, m`**, where *c* is a condition variable and `m` is a mutex (lock) associated with the monitor. This operation is called by a thread that needs to wait until the assertion P_c is true before proceeding. While the thread is waiting, it does not occupy the monitor. The function, and fundamental contract, of the "wait" operation, is to do the following steps:
  1. Atomically:
     a. release the mutex *m*,
     b. move this thread from the "running" to *c*'s "wait-queue" (a.k.a. "sleep-queue") of threads, and
     c. sleep this thread. (Context is synchronously yielded to another thread.)
  2. Once this thread is subsequently notified/signaled (see below) and resumed, then automatically re-acquire the mutex *m*.

  Steps 1a and 1b can occur in either order, with 1c usually occurring after them. While the thread is sleeping and in *c*'s wait-queue, the next program counter to be executed is at step 2, in the middle of the "wait" function/subroutine. Thus, the thread sleeps and later wakes up in the middle of the "wait" operation.
  The atomicity of the operations within step 1 is important to avoid race conditions that would be caused by a preemptive thread switch in-between them. One failure mode that could occur if these were not atomic is a *missed wakeup*, in which the thread could be on *c*'s sleep-queue and have released the mutex, but a preemptive thread switch occurred before the thread went to sleep, and another thread called a signal operation (see below) on *c* moving the first thread back out of *c*'s queue. As soon as the first thread in question is switched back to, its program counter will be at step 1c, and it will sleep and be unable to be woken up again, violating the invariant that it should have been on *c*'s sleep-queue when it slept. Other race conditions depend on the ordering of steps 1a and 1b, and depend on where a context switch occurs.
- **`signal`** `c`, also known as **`notify`** `c`, is called by a thread to indicate that the assertion P_c is true. Depending on the type and implementation of the monitor, this moves one or more threads from `c`'s sleep-queue to the "ready queue", or another queue for it to be executed. It is usually considered a best practice to perform the "signal" operation before releasing mutex *m* that is associated with *c*, but as long as the code is properly designed for concurrency and depending on the threading implementation, it is often also acceptable to release the lock before signalling. Depending on the threading implementation, the ordering of this can have scheduling-priority ramifications. (Some authors instead advocate a preference for releasing the lock before signalling.) A threading implementation should document any special constraints on this ordering.
- **`broadcast`** `c`, also known as **`notifyAll`** `c`, is a similar operation that wakes up all threads in

c's wait-queue. This empties the wait-queue. Generally, when more than one predicate condition is associated with the same condition variable, the application will require **broadcast** instead of **signal** because a thread waiting for the wrong condition might be woken up and then immediately go back to sleep without waking up a thread waiting for the correct condition that just became true. Otherwise, if the predicate condition is one-to-one with the condition variable associated with it, then **signal** may be more efficient than **broadcast**.

As a design rule, multiple condition variables can be associated with the same mutex, but not vice versa. (This is a one-to-many correspondence.) This is because the predicate P_c is the same for all threads using the monitor and must be protected with mutual exclusion from all other threads that might cause the condition to be changed or that might read it while the thread in question causes it to be changed, but there may be different threads that want to wait for a different condition on the same variable requiring the same mutex to be used. In the producer-consumer example described above, the queue must be protected by a unique mutex object, m. The "producer" threads will want to wait on a monitor using lock m and a condition variable c_{full} which blocks until the queue is non-full. The "consumer" threads will want to wait on a different monitor using the same mutex m but a different condition variable c_{empty} which blocks until the queue is non-empty. It would (usually) never make sense to have different mutexes for the same condition variable, but this classic example shows why it often certainly makes sense to have multiple condition variables using the same mutex. A mutex used by one or more condition variables (one or more monitors) may also be shared with code that does *not* use condition variables (and which simply acquires/releases it without any wait/signal operations), if those critical sections do not happen to require waiting for a certain condition on the concurrent data.

Monitor usage

The proper basic usage of a monitor is:

```
acquire(m); // Acquire this monitor's lock.
while (!p) { // While the condition/predicate/assertion that we are waiting for is not true...
    wait(m, cv); // Wait on this monitor's lock and condition variable.
}
// ... Critical section of code goes here ...
signal(cv2); // Or: broadcast(cv2);
             // cv2 might be the same as cv or different.
release(m); // Release this monitor's lock.
```

To be more precise, this is the same pseudocode but with more verbose comments to better explain what is going on:

```
// ... (previous code)
// About to enter the monitor.
// Acquire the advisory mutex (lock) associated with the concurrent
// data that is shared between threads,
// to ensure that no two threads can be preemptively interleaved or
// run simultaneously on different cores while executing in critical
// sections that read or write this same concurrent data. If another
// thread is holding this mutex, then this thread will be put to sleep
// (blocked) and placed on m's sleep queue.  (Mutex "m" shall not be
// a spin-lock.)
acquire(m);
// Now, we are holding the lock and can check the condition for the
// first time.
```

```
// The first time we execute the while loop condition after the above
// "acquire", we are asking, "Does the condition/predicate/assertion
// we are waiting for happen to already be true?"

while (!p())    // "p" is any expression (e.g. variable or
        // function-call) that checks the condition and
        // evaluates to boolean.  This itself is a critical
        // section, so you *MUST* be holding the lock when
        // executing this "while" loop condition!

// If this is not the first time the "while" condition is being checked,
// then we are asking the question, "Now that another thread using this
// monitor has notified me and woken me up and I have been context-switched
// back to, did the condition/predicate/assertion we are waiting on stay
// true between the time that I was woken up and the time that I re-acquired
// the lock inside the "wait" call in the last iteration of this loop, or
// did some other thread cause the condition to become false again in the
// meantime thus making this a spurious wakeup?

{
    // If this is the first iteration of the loop, then the answer is
    // "no" -- the condition is not ready yet. Otherwise, the answer is:
    // the latter.  This was a spurious wakeup, some other thread occurred
    // first and caused the condition to become false again, and we must
    // wait again.

    wait(m, cv);
        // Temporarily prevent any other thread on any core from doing
        // operations on m or cv.
        // release(m)       // Atomically release lock "m" so other
        //          // code using this concurrent data
        //          // can operate, move this thread to cv's
        //          // wait-queue so that it will be notified
        //          // sometime when the condition becomes
        //          // true, and sleep this thread. Re-enable
        //          // other threads and cores to do
        //          // operations on m and cv.
        //
        // Context switch occurs on this core.
        //
        // At some future time, the condition we are waiting for becomes
        // true, and another thread using this monitor (m, cv) does either
        // a signal that happens to wake this thread up, or a
        // broadcast that wakes us up, meaning that we have been taken out
        // of cv's wait-queue.
        //
        // During this time, other threads may cause the condition to
        // become false again, or the condition may toggle one or more
        // times, or it may happen to stay true.
        //
        // This thread is switched back to on some core.
        //
        // acquire(m)       // Lock "m" is re-acquired.

    // End this loop iteration and re-check the "while" loop condition to make
    // sure the predicate is still true.

}

// The condition we are waiting for is true!
// We are still holding the lock, either from before entering the monitor or from
// the last execution of "wait".

// Critical section of code goes here, which has a precondition that our predicate
// must be true.
// This code might make cv's condition false, and/or make other condition variables'
// predicates true.

// Call signal or broadcast, depending on which condition variables'
// predicates (who share mutex m) have been made true or may have been made true,
// and the monitor semantic type being used.
```

```
for (cv_x in cvs_to_signal) {
    signal(cv_x); // Or: broadcast(cv_x);
}
// One or more threads have been woken up but will block as soon as they try
// to acquire m.

// Release the mutex so that notified thread(s) and others can enter their critical
// sections.
release(m);
```

Solving the bounded producer/consumer problem

Having introduced the usage of condition variables, let us use it to revisit and solve the classic bounded producer/consumer problem. The classic solution is to use two monitors, comprising two condition variables sharing one lock on the queue:

```
global volatile RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks. (Not a spin-lock.)
global CV queueEmptyCV; // A condition variable for consumer threads waiting for the queue to
                        // become non-empty. Its associated lock is "queueLock".
global CV queueFullCV; // A condition variable for producer threads waiting for the queue to
                       // become non-full. Its associated lock is also "queueLock".

// Method representing each producer thread's behavior:
public method producer() {
    while (true) {
        // Producer makes some new task to be added.
        task myTask = ...;

        // Acquire "queueLock" for the initial predicate check.
        queueLock.acquire();

        // Critical section that checks if the queue is non-full.
        while (queue.isFull()) {
            // Release "queueLock", enqueue this thread onto "queueFullCV" and sleep this thread.
            wait(queueLock, queueFullCV);
            // When this thread is awoken, re-acquire "queueLock" for the next predicate check.
        }

        // Critical section that adds the task to the queue (note that we are holding "queueLock").
        queue.enqueue(myTask);

        // Wake up one or all consumer threads that are waiting for the queue to be non-empty
        // now that it is guaranteed, so that a consumer thread will take the task.
        signal(queueEmptyCV); // Or: broadcast(queueEmptyCV);
        // End of critical sections.

        // Release "queueLock" until we need it again to add the next task.
        queueLock.release();
    }
}

// Method representing each consumer thread's behavior:
public method consumer() {
    while (true) {
        // Acquire "queueLock" for the initial predicate check.
        queueLock.acquire();

        // Critical section that checks if the queue is non-empty.
        while (queue.isEmpty()) {
            // Release "queueLock", enqueue this thread onto "queueEmptyCV" and sleep this thread.
            wait(queueLock, queueEmptyCV);
            // When this thread is awoken, re-acquire "queueLock" for the next predicate check.
        }

        // Critical section that takes a task off of the queue (note that we are holding "queueLock").
        myTask = queue.dequeue();
```

```
        // Wake up one or all producer threads that are waiting for the queue to be non-full
        // now that it is guaranteed, so that a producer thread will add a task.
        signal(queueFullCV); // Or: broadcast(queueFullCV);
        // End of critical sections.

        // Release "queueLock" until we need it again to take the next task.
        queueLock.release();

        // Go off and do something with the task.
        doStuff(myTask);
    }
}
```

This ensures concurrency between the producer and consumer threads sharing the task queue, and blocks the threads that have nothing to do rather than busy-waiting as shown in the aforementioned approach using spin-locks.

A variant of this solution could use a single condition variable for both producers and consumers, perhaps named "queueFullOrEmptyCV" or "queueSizeChangedCV". In this case, more than one condition is associated with the condition variable, such that the condition variable represents a weaker condition than the conditions being checked by individual threads. The condition variable represents threads that are waiting for the queue to be non-full *and* ones waiting for it to be non-empty. However, doing this would require using *broadcast* in all the threads using the condition variable and cannot use a regular *signal*. This is because the regular *signal* might wake up a thread of the wrong type whose condition has not yet been met, and that thread would go back to sleep without a thread of the correct type getting signaled. For example, a producer might make the queue full and wake up another producer instead of a consumer, and the woken producer would go back to sleep. In the complementary case, a consumer might make the queue empty and wake up another consumer instead of a producer, and the consumer would go back to sleep. Using *broadcast* ensures that some thread of the right type will proceed as expected by the problem statement.

Here is the variant using only one condition variable and broadcast:

```
global volatile RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks.  (Not a spin-lock.)
global CV queueFullOrEmptyCV; // A single condition variable for when the queue is not ready for any thread
                              // i.e. for producer threads waiting for the queue to become non-full
                              // and consumer threads waiting for the queue to become non-empty.
                              // Its associated lock is "queueLock".
                              // Not safe to use regular "signal" because it is associated with
                              // multiple predicate conditions (assertions).

// Method representing each producer thread's behavior:
public method producer() {
    while (true) {
        // Producer makes some new task to be added.
        task myTask = ...;

        // Acquire "queueLock" for the initial predicate check.
        queueLock.acquire();

        // Critical section that checks if the queue is non-full.
        while (queue.isFull()) {
            // Release "queueLock", enqueue this thread onto "queueFullOrEmptyCV" and sleep this thread.
            wait(queueLock, queueFullOrEmptyCV);
            // When this thread is awoken, re-acquire "queueLock" for the next predicate check.
        }

        // Critical section that adds the task to the queue (note that we are holding "queueLock").
        queue.enqueue(myTask);
```

```
        // Wake up all producer and consumer threads that are waiting for the queue to be respectively
        // non-full and non-empty now that the latter is guaranteed, so that a consumer thread will take the task.
        broadcast(queueFullOrEmptyCV); // Do not use "signal" (as it might wake up another producer thread only).
        // End of critical sections.

        // Release "queueLock" until we need it again to add the next task.
        queueLock.release();
    }
}

// Method representing each consumer thread's behavior:
public method consumer() {
    while (true) {
        // Acquire "queueLock" for the initial predicate check.
        queueLock.acquire();

        // Critical section that checks if the queue is non-empty.
        while (queue.isEmpty()) {
            // Release "queueLock", enqueue this thread onto "queueFullOrEmptyCV" and sleep this thread.
            wait(queueLock, queueFullOrEmptyCV);
            // When this thread is awoken, re-acquire "queueLock" for the next predicate check.
        }

        // Critical section that takes a task off of the queue (note that we are holding "queueLock").
        myTask = queue.dequeue();

        // Wake up all producer and consumer threads that are waiting for the queue to be respectively
        // non-full and non-empty now that the former is guaranteed, so that a producer thread will add a task.
        broadcast(queueFullOrEmptyCV); // Do not use "signal" (as it might wake up another consumer thread only).
        // End of critical sections.

        // Release "queueLock" until we need it again to take the next task.
        queueLock.release();

        // Go off and do something with the task.
        doStuff(myTask);
    }
}
```

Synchronization primitives

Implementing mutexes and condition variables requires some kind of synchronization primitive provided by hardware support that provides atomicity. Locks and condition variables are higher-level abstractions over these synchronization primitives. On a uniprocessor, disabling and enabling interrupts is a way to implement monitors by preventing context switches during the critical sections of the locks and condition variables, but this is not enough on a multiprocessor. On a multiprocessor, usually special atomic **read-modify-write** instructions on the memory such as **test-and-set**, **compare-and-swap**, etc. are used, depending on what the instruction set architecture provides. These usually require deferring to spin-locking for the internal lock state itself, but this locking is very brief. Depending on the implementation, the atomic read-modify-write instructions may lock the bus from other cores' accesses and/or prevent re-ordering of instructions in the CPU. Here is an example pseudocode implementation of parts of a threading system and mutexes and Mesa-style condition variables, using **test-and-set** and a first-come, first-served policy. This glosses over most of how a threading system works, but shows the parts relevant to mutexes and condition variables:

Sample Mesa-monitor implementation with Test-and-Set

```
// Basic parts of threading system:
// Assume "ThreadQueue" supports random access.
```

```
public volatile ThreadQueue readyQueue; // Thread-unsafe queue of ready threads.  Elements are (Thread*).
public volatile global Thread* currentThread; // Assume this variable is per-core.  (Others are shared.)

// Implements a spin-lock on just the synchronized state of the threading system itself.
// This is used with test-and-set as the synchronization primitive.
public volatile global bool threadingSystemBusy = false;

// Context-switch interrupt service routine (ISR):
// On the current CPU core, preemptively switch to another thread.
public method contextSwitchISR() {
    if (testAndSet(threadingSystemBusy)) {
        return; // Can't switch context right now.
    }

    // Ensure this interrupt can't happen again which would foul up the context switch:
    systemCall_disableInterrupts();

    // Get all of the registers of the currently-running process.
    // For Program Counter (PC), we will need the instruction location of
    // the "resume" label below.  Getting the register values is platform-dependent and may involve
    // reading the current stack frame, JMP/CALL instructions, etc.  (The details are beyond this scope.)
    currentThread->registers = getAllRegisters(); // Store the registers in the "currentThread" object in memory.
    currentThread->registers.PC = resume; // Set the next PC to the "resume" label below in this method.

    readyQueue.enqueue(currentThread); // Put this thread back onto the ready queue for later execution.

    Thread* otherThread = readyQueue.dequeue(); // Remove and get the next thread to run from the ready queue.

    currentThread = otherThread; // Replace the global current-thread pointer value so it is ready for the next
thread.

    // Restore the registers from currentThread/otherThread, including a jump to the stored PC of the other thread
    // (at "resume" below).  Again, the details of how this is done are beyond this scope.
    restoreRegisters(otherThread.registers);

    // *** Now running "otherThread" (which is now "currentThread")!  The original thread is now "sleeping". ***

    resume: // This is where another contextSwitch() call needs to set PC to when switching context back here.

    // Return to where otherThread left off.

    threadingSystemBusy = false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.
}

// Thread sleep method:
// On current CPU core, a synchronous context switch to another thread without putting
// the current thread on the ready queue.
// Must be holding "threadingSystemBusy" and disabled interrupts so that this method
// doesn't get interrupted by the thread-switching timer which would call contextSwitchISR().
// After returning from this method, must clear "threadingSystemBusy".
public method threadSleep() {
    // Get all of the registers of the currently-running process.
    // For Program Counter (PC), we will need the instruction location of
    // the "resume" label below.  Getting the register values is platform-dependent and may involve
    // reading the current stack frame, JMP/CALL instructions, etc.  (The details are beyond this scope.)
    currentThread->registers = getAllRegisters(); // Store the registers in the "currentThread" object in memory.
    currentThread->registers.PC = resume; // Set the next PC to the "resume" label below in this method.

    // Unlike contextSwitchISR(), we will not place currentThread back into readyQueue.
    // Instead, it has already been placed onto a mutex's or condition variable's queue.

    Thread* otherThread = readyQueue.dequeue(); // Remove and get the next thread to run from the ready queue.

    currentThread = otherThread; // Replace the global current-thread pointer value so it is ready for the next
thread.

    // Restore the registers from currentThread/otherThread, including a jump to the stored PC of the other thread
    // (at "resume" below).  Again, the details of how this is done are beyond this scope.
    restoreRegisters(otherThread.registers);

    // *** Now running "otherThread" (which is now "currentThread")!  The original thread is now "sleeping". ***
```

```
    resume: // This is where another contextSwitch() call needs to set PC to when switching context back here.

    // Return to where otherThread left off.
}

public method wait(Mutex m, ConditionVariable c) {
    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)) {}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be sleeped
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    assert m.held; // (Specifically, this thread must be the one holding it.)

    m.release();
    c.waitingThreads.enqueue(currentThread);

    threadSleep();

    // Thread sleeps ... Thread gets woken up from a signal/broadcast.

    threadingSystemBusy = false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // Context switches may now occur here, making the client caller's predicate false.

    m.acquire();
}

public method signal(ConditionVariable c) {
    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)) {}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be sleeped
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    if (!c.waitingThreads.isEmpty()) {
        wokenThread = c.waitingThreads.dequeue();
        readyQueue.enqueue(wokenThread);
    }

    threadingSystemBusy = false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // The woken thread is not given any priority.
}

public method broadcast(ConditionVariable c) {
    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)) {}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be sleeped
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();
```

```
    while (!c.waitingThreads.isEmpty()) {
        wokenThread = c.waitingThreads.dequeue();
        readyQueue.enqueue(wokenThread);
    }

    threadingSystemBusy = false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // The woken threads are not given any priority.
}

class Mutex {
    protected volatile bool held = false;
    private volatile ThreadQueue blockingThreads; // Thread-unsafe queue of blocked threads.  Elements are
(Thread*).

    public method acquire() {
        // Internal spin-lock while other threads on any core are accessing this object's
        // "held" and "threadQueue", or "readyQueue".
        while (testAndSet(threadingSystemBusy)) {}
        // N.B.: "threadingSystemBusy" is now true.

        // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
        // the thread-switching timer on this core which would call contextSwitchISR().
        // Done outside threadSleep() for more efficiency so that this thread will be sleeped
        // right after going on the lock queue.
        systemCall_disableInterrupts();

        assert !blockingThreads.contains(currentThread);

        if (held) {
            // Put "currentThread" on this lock's queue so that it will be
            // considered "sleeping" on this lock.
            // Note that "currentThread" still needs to be handled by threadSleep().
            readyQueue.remove(currentThread);
            blockingThreads.enqueue(currentThread);
            threadSleep();

            // Now we are woken up, which must be because "held" became false.
            assert !held;
            assert !blockingThreads.contains(currentThread);
        }

        held = true;

        threadingSystemBusy = false; // Must be an atomic assignment.
        systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.
    }

    public method release() {
        // Internal spin-lock while other threads on any core are accessing this object's
        // "held" and "threadQueue", or "readyQueue".
        while (testAndSet(threadingSystemBusy)) {}
        // N.B.: "threadingSystemBusy" is now true.

        // System call to disable interrupts on this core for efficiency.
        systemCall_disableInterrupts();

        assert held; // (Release should only be performed while the lock is held.)

        held = false;

        if (!blockingThreads.isEmpty()) {
            Thread* unblockedThread = blockingThreads.dequeue();
            readyQueue.enqueue(unblockedThread);
        }

        threadingSystemBusy = false; // Must be an atomic assignment.
        systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.
    }
}
```

```
struct ConditionVariable {
    volatile ThreadQueue waitingThreads;
}
```

Semaphore

As an example, consider a thread-safe class that implements a semaphore. There are methods to increment (V) and to decrement (P) a private integer `s`. However, the integer must never be decremented below 0; thus a thread that tries to decrement must wait until the integer is positive. We use a condition variable `sIsPositive` with an associated assertion of $P_{\texttt{sIsPositive}} = (s > 0)$.

```
monitor class Semaphore
{
    private int s := 0
    invariant s >= 0
    private Condition sIsPositive /* associated with s > 0 */

    public method P()
    {
        while s = 0:
            wait sIsPositive
        assert s > 0
        s := s - 1
    }

    public method V()
    {
        s := s + 1
        assert s > 0
        signal sIsPositive
    }
}
```

Implemented showing all synchronization (removing the assumption of a thread-safe class and showing the mutex):

```
class Semaphore
{
    private volatile int s := 0
    invariant s >= 0
    private ConditionVariable sIsPositive /* associated with s > 0 */
    private Mutex myLock /* Lock on "s" */

    public method P()
    {
        myLock.acquire()
        while s = 0:
            wait(myLock, sIsPositive)
        assert s > 0
        s := s - 1
        myLock.release()
    }

    public method V()
    {
        myLock.acquire()
        s := s + 1
        assert s > 0
        signal sIsPositive
        myLock.release()
    }
}
```

Monitor implemented using semaphores

Conversely, locks and condition variables can also be derived from semaphores, thus making monitors and semaphores reducible to one another:

The implementation given here is incorrect. If a thread calls wait() after broadcast() has been called, an originally thread may be stuck indefinitely, since broadcast() increments the semaphore only enough times for threads already waiting.

```
public method wait(Mutex m, ConditionVariable c) {
    assert m.held;

    c.internalMutex.acquire();

    c.numWaiters++;
    m.release(); // Can go before/after the neighboring lines.
    c.internalMutex.release();

    // Another thread could signal here, but that's OK because of how
    // semaphores count.  If c.sem's number becomes 1, we'll have no
    // waiting time.
    c.sem.acquire(); // Block on the CV.
    // Woken
    m.acquire(); // Re-acquire the mutex.
}

public method signal(ConditionVariable c) {
    c.internalMutex.acquire();
    if (c.numWaiters > 0) {
        c.numWaiters--;
        c.sem.release(); // (Doesn't need to be protected by c.internalMutex.)
    }
    c.internalMutex.release();
}

public method broadcast(ConditionVariable c) {
    c.internalMutex.acquire();
    while (c.numWaiters > 0) {
```

```
        c.numWaiters--;
        c.sem.release(); // (Doesn't need to be protected by c.internalMutex.)
    }
    c.internalMutex.release();
}

class Mutex {
    protected boolean held = false; // For assertions only, to make sure sem's number never goes > 1.
    protected Semaphore sem = Semaphore(1); // The number shall always be at most 1.
                                            // Not held <--> 1; held <--> 0.

    public method acquire() {
        sem.acquire();
        assert !held;
        held = true;
    }

    public method release() {
        assert held; // Make sure we never acquire sem above 1.  That would be bad.
        held = false;
        sem.release();
    }
}

class ConditionVariable {
    protected int numWaiters = 0; // Roughly tracks the number of waiters blocked in sem.
                                  // (The semaphore's internal state is necessarily private.)
    protected Semaphore sem = Semaphore(0); // Provides the wait queue.
    protected Mutex internalMutex; // (Really another Semaphore.  Protects "numWaiters".)
}
```

When a **signal** happens on a condition variable that at least one other thread is waiting on, there are at least two threads that could then occupy the monitor: the thread that signals and any one of the threads that is waiting. In order that at most one thread occupies the monitor at each time, a choice must be made. Two schools of thought exist on how best to resolve this choice. This leads to two kinds of condition variables which will be examined next:

- *Blocking condition variables* give priority to a signaled thread.
- *Nonblocking condition variables* give priority to the signaling thread.

Blocking condition variables

The original proposals by C. A. R. Hoare and Per Brinch Hansen were for *blocking condition variables*. With a blocking condition variable, the signaling thread must wait outside the monitor (at least) until the signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition variable. Monitors using blocking condition variables are often called *Hoare-style* monitors or *signal-and-urgent-wait* monitors.

We assume there are two queues of threads associated with each monitor object

- `e` is the entrance queue
- `s` is a queue of threads that have signaled.

In addition we assume that for each condition variable c, there is a queue

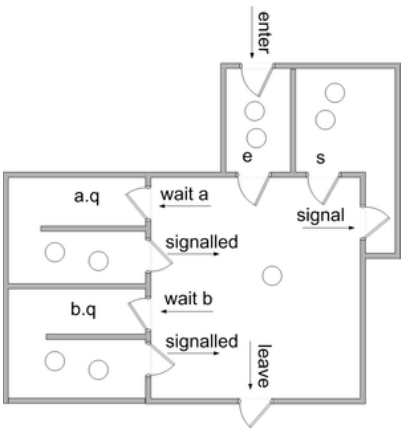
- c`.q`, which is a queue for threads waiting on condition variable c

All queues are typically guaranteed to be fair and, in some implementations, may be guaranteed to

be first in first out.

The implementation of each operation is as follows. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)

```
enter the monitor:
    enter the method
    if the monitor is locked
        add this thread to e
        block this thread
    else
        lock the monitor

leave the monitor:
    schedule
    return from the method

wait c:
    add this thread to c.q
    schedule
    block this thread

signal c:
    if there is a thread waiting on c.q
        select and remove one such thread t from c.q
        (t is called "the signaled thread")
        add this thread to s
        restart t
        (so t will occupy the monitor next)
        block this thread

schedule:
    if there is a thread on s
        select and remove one thread from s and restart it
        (this thread will occupy the monitor next)
    else if there is a thread on e
        select and remove one thread from e and restart it
        (this thread will occupy the monitor next)
    else
        unlock the monitor
        (the monitor will become unoccupied)
```



A Hoare style monitor with two condition variables a and b. After Buhr *et al.*

The `schedule` routine selects the next thread to occupy the monitor or, in the absence of any candidate threads, unlocks the monitor.

The resulting signaling discipline is known as *"signal and urgent wait,"* as the signaler must wait, but is given priority over threads on the entrance queue. An alternative is *"signal and wait,"* in which there is no `s` queue and signaler waits on the `e` queue instead.

Some implementations provide a **signal and return** operation that combines signaling with returning from a procedure.

```
signal c and return:
    if there is a thread waiting on c.q
        select and remove one such thread t from c.q
        (t is called "the signaled thread")
        restart t
        (so t will occupy the monitor next)
    else
        schedule
    return from the method
```

In either case ("signal and urgent wait" or "signal and wait"), when a condition variable is signaled and there is at least one thread waiting on the condition variable, the signaling thread hands occupancy over to the signaled thread seamlessly, so that no other thread can gain occupancy in between. If P_c is true at the start of each **signal** c operation, it will be true at the end of each **wait** c operation. This is summarized by the following contracts. In these contracts, I is the monitor's invariant.

```
enter the monitor:
    postcondition I

leave the monitor:
    precondition I

wait c:
    precondition I
    modifies the state of the monitor
    postcondition P_c and I

signal c:
    precondition P_c and I
    modifies the state of the monitor
    postcondition I

signal c and return:
    precondition P_c and I
```

In these contracts, it is assumed that I and P_c do not depend on the contents or lengths of any queues.

(When the condition variable can be queried as to the number of threads waiting on its queue, more sophisticated contracts can be given. For example, a useful pair of contracts, allowing occupancy to be passed without establishing the invariant, is:

```
wait c:
    precondition I
    modifies the state of the monitor
    postcondition P_c

signal c
    precondition (not empty(c) and P_c) or (empty(c) and I)
    modifies the state of the monitor
    postcondition I
```

(See Howard[4] and Buhr *et al.*[5] for more.)

It is important to note here that the assertion P_c is entirely up to the programmer; he or she simply needs to be consistent about what it is.

We conclude this section with an example of a thread-safe class using a blocking monitor that implements a bounded, thread-safe stack.

```
monitor class SharedStack {
    private const capacity := 10
    private int[capacity] A
    private int size := 0
    invariant 0 <= size and size <= capacity
    private BlockingCondition theStackIsNotEmpty /* associated with 0 < size and size <= capacity */
    private BlockingCondition theStackIsNotFull /* associated with 0 <= size and size < capacity */

    public method push(int value)
    {
        if size = capacity then wait theStackIsNotFull
        assert 0 <= size and size < capacity
        A[size] := value ; size := size + 1
        assert 0 < size and size <= capacity
        signal theStackIsNotEmpty and return
    }

    public method int pop()
    {
        if size = 0 then wait theStackIsNotEmpty
        assert 0 < size and size <= capacity
        size := size - 1 ;
        assert 0 <= size and size < capacity
        signal theStackIsNotFull and return A[size]
    }
}
```

Note that, in this example, the thread-safe stack is internally providing a mutex, which, as in the earlier producer/consumer example, is shared by both condition variables, which are checking different conditions on the same concurrent data. The only difference is that the producer/consumer example assumed a regular non-thread-safe queue and was using a standalone mutex and condition variables, without these details of the monitor abstracted away as is the case here. In this example, when the "wait" operation is called, it must somehow be supplied with the thread-safe stack's mutex, such as if the "wait" operation is an integrated part of the "monitor class". Aside from this kind of abstracted functionality, when a "raw" monitor is used, it will *always* have to include a mutex and a condition variable, with a unique mutex for each condition variable.

Nonblocking condition variables

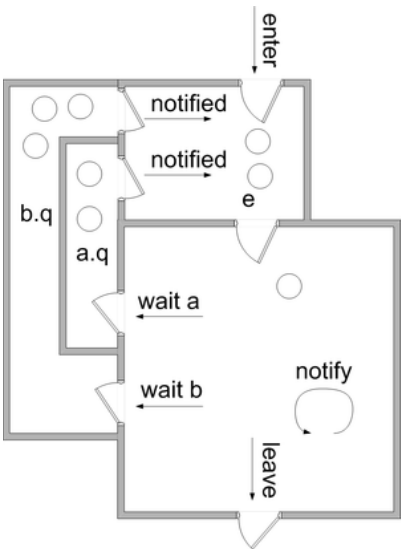
With *nonblocking condition variables* (also called *"Mesa style"* condition variables or *"signal and continue"* condition variables), signaling does not cause the signaling thread to lose occupancy of the monitor. Instead the signaled threads are moved to the `e` queue. There is no need for the `s` queue.

With nonblocking condition variables, the **signal** operation is often called **notify** — a terminology we will follow here. It is also common to provide a **notify all** operation that moves all threads waiting on a condition variable to the `e` queue.

The meaning of various operations are given here. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)

```
enter the monitor:
    enter the method
```

```
    if the monitor is locked
        add this thread to e
        block this thread
    else
        lock the monitor

leave the monitor:
    schedule
    return from the method

wait c:
    add this thread to c.q
    schedule
    block this thread

notify c:
    if there is a thread waiting on c.q
        select and remove one thread t from c.q
        (t is called "the notified thread")
        move t to e

notify all c:
    move all threads waiting on c.q to e

schedule :
    if there is a thread on e
        select and remove one thread from e and restart it
    else
        unlock the monitor
```



A Mesa style monitor with two condition variables a and b

As a variation on this scheme, the notified thread may be moved to a queue called `w`, which has priority over `e`. See Howard[4] and Buhr *et al.*[5] for further discussion.

It is possible to associate an assertion P_c with each condition variable c such that P_c is sure to be true upon return from **`wait`** c. However, one must ensure that P_c is preserved from the time the **notify**ing thread gives up occupancy until the notified thread is selected to re-enter the monitor. Between these times there could be activity by other occupants. Thus it is common for P_c to simply be *true*.

For this reason, it is usually necessary to enclose each **wait** operation in a loop like this

```
while not ( P ) do
    wait c
```

where P is some condition stronger than P_c. The operations **`notify`** c and **`notify all`** c are treated as "hints" that P may be true for some waiting thread. Every iteration of such a loop past the first represents a lost notification; thus with nonblocking monitors, one must be careful to ensure that too many notifications cannot be lost.

As an example of "hinting," consider a bank account in which a withdrawing thread will wait until the account has sufficient funds before proceeding

```
monitor class Account {
    private int balance := 0
    invariant balance >= 0
    private NonblockingCondition balanceMayBeBigEnough

    public method withdraw(int amount)
        precondition amount >= 0
    {
```

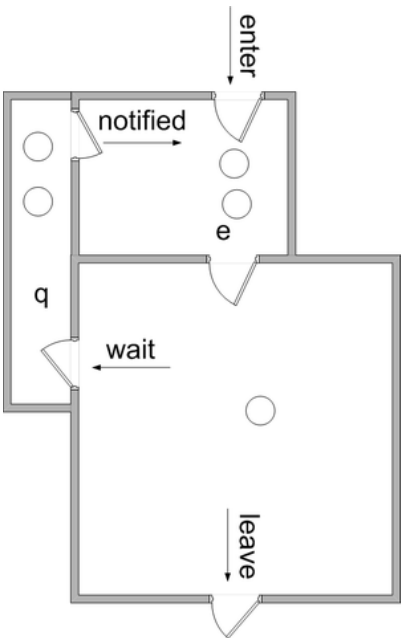
```
        while balance < amount do wait balanceMayBeBigEnough
        assert balance >= amount
        balance := balance - amount
    }

    public method deposit(int amount)
        precondition amount >= 0
    {
        balance := balance + amount
        notify all balanceMayBeBigEnough
    }
}
```

In this example, the condition being waited for is a function of the amount to be withdrawn, so it is impossible for a depositing thread to *know* that it made such a condition true. It makes sense in this case to allow each waiting thread into the monitor (one at a time) to check if its assertion is true.

Implicit condition variable monitors

In the Java language, each object may be used as a monitor. Methods requiring mutual exclusion must be explicitly marked with the **synchronized** keyword. Blocks of code may also be marked by **synchronized**.

Rather than having explicit condition variables, each monitor (i.e., object) is equipped with a single wait queue in addition to its entrance queue. All waiting is done on this single wait queue and all **notify** and **notifyAll** operations apply to this queue. This approach has been adopted in other languages, for example C#.



A Java style monitor

Implicit signaling

Another approach to signaling is to omit the **signal** operation. Whenever a thread leaves the monitor (by returning or waiting), the assertions of all waiting threads are evaluated until one is found to be true. In such a system, condition variables are not needed, but the assertions must be explicitly coded. The contract for wait is

```
wait P:
    precondition I
    modifies the state of the monitor
    postcondition P and I
```

History

Brinch Hansen and Hoare developed the monitor concept in the early 1970s, based on earlier ideas of their own and of Edsger Dijkstra.[6] Brinch Hansen published the first monitor notation, adopting the class concept of Simula 67,[1] and invented a queueing mechanism.[7] Hoare refined the rules of process resumption.[2] Brinch Hansen created the first implementation of monitors, in Concurrent Pascal.[6] Hoare demonstrated their equivalence to semaphores.

Monitors (and Concurrent Pascal) were soon used to structure process synchronization in the Solo operating system.[8][9]

Programming languages that have supported monitors include:

- Ada since Ada 95 (as protected objects)
- C# (and other languages that use the .NET Framework)
- Concurrent Euclid
- Concurrent Pascal
- D
- Delphi (Delphi 2009 and above, via TObject.Monitor)
- Java (via the wait and notify methods)
- Go[10][11]
- Mesa
- Modula-3
- Python (via threading.Condition (https://docs.python.org/library/threading.html#condition-objects) object)
- Ruby
- Squeak Smalltalk
- Turing, Turing+, and Object-Oriented Turing
- μC++
- Visual Prolog

A number of libraries have been written that allow monitors to be constructed in languages that do not support them natively. When library calls are used, it is up to the programmer to explicitly mark the start and end of code executed with mutual exclusion. Pthreads is one such library.

See also

- Mutual exclusion
- Communicating sequential processes - a later development of monitors by C. A. R. Hoare
- Semaphore (programming)

Notes

1. Brinch Hansen, Per (1973). "7.2 Class Concept" (http://brinch-hansen.net/papers/1973b.pdf) (PDF). *Operating System Principles* (https://archive.org/details/operatingsystemp0000brin). Prentice Hall. ISBN 978-0-13-637843-3.
2. Hoare, C. A. R. (October 1974). "Monitors: an operating system structuring concept". *Comm. ACM*. **17** (10): 549–557. CiteSeerX 10.1.1.24.6394 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.6394). doi:10.1145/355620.361161 (https://doi.org/10.1145%2F355620.361161). S2CID 1005769 (https://api.semanticscholar.org/CorpusID:1005769).
3. Hansen, P. B. (June 1975). "The programming language Concurrent Pascal" (https://authors.library.caltech.edu/34677/1/06312840.pdf) (PDF). *IEEE Trans. Softw. Eng.* **SE-1** (2): 199–207. doi:10.1109/TSE.1975.6312840 (https://doi.org/10.1109%2FTSE.1975.6312840). S2CID 2000388 (https://api.semanticscholar.org/CorpusID:2000388).

4. Howard, John H. (1976). "Signaling in monitors" (http://dl.acm.org/citation.cfm?id=807647). *ICSE '76 Proceedings of the 2nd international conference on Software engineering*. International Conference on Software Engineering. Los Alamitos, CA, USA: IEEE Computer Society Press. pp. 47–52.
5. Buhr, Peter A.; Fortier, Michel; Coffin, Michael H. (March 1995). "Monitor classification". *ACM Computing Surveys*. **27** (1): 63–107. doi:10.1145/214037.214100 (https://doi.org/10.1145%2F214037.214100). S2CID 207193134 (https://api.semanticscholar.org/CorpusID:207193134).
6. Hansen, Per Brinch (1993). "Monitors and concurrent Pascal: a personal history". *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*. History of Programming Languages. New York, NY, USA: ACM. pp. 1–35. doi:10.1145/155360.155361 (https://doi.org/10.1145%2F155360.155361). ISBN 0-89791-570-4.
7. Brinch Hansen, Per (July 1972). "Structured multiprogramming (Invited Paper)". *Communications of the ACM*. **15** (7): 574–578. doi:10.1145/361454.361473 (https://doi.org/10.1145%2F361454.361473). S2CID 14125530 (https://api.semanticscholar.org/CorpusID:14125530).
8. Brinch Hansen, Per (April 1976). "The Solo operating system: a Concurrent Pascal program" (http://brinch-hansen.net/papers/1976b.pdf) (PDF). *Software: Practice and Experience*.
9. Brinch Hansen, Per (1977). *The Architecture of Concurrent Programs*. Prentice Hall. ISBN 978-0-13-044628-2.
10. "sync - The Go Programming Language" (https://golang.org/pkg/sync/#Cond). *golang.org*. Retrieved 2021-06-17.
11. "What's the "sync.Cond" | dtyler.io" (https://dtyler.io/articles/2021/04/13/sync_cond/). *dtyler.io*. Retrieved 2021-06-17.

Further reading

- Monitors: an operating system structuring concept, C. A. R. Hoare – Communications of the ACM, v.17 n.10, p. 549–557, Oct. 1974 [1] (http://doi.acm.org/10.1145/355620.361161)
- Monitor classification P.A. Buhr, M. Fortier, M.H. Coffin – ACM Computing Surveys, 1995 [2] (http://doi.acm.org/10.1145/214037.214100)

External links

- Java Monitors (lucid explanation) (http://www.artima.com/insidejvm/ed2/threadsynch.html)
- "Monitors: An Operating System Structuring Concept (https://web.archive.org/web/20060830171518/http://www.acm.org/classics/feb96/)" by C. A. R. Hoare
- "Signalling in Monitors (http://portal.acm.org/citation.cfm?id=807647)"

Device Drivers: Classes and Services

Introduction

Device drivers represent both:

generalizing abstractions

gathering a myriad of very different devices together and synthesizing a few general classes (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 implemention of standard class interfaces while opaquely encapsulating the details of how to effectively and efficiently use a particular device.

For reasons of performance and control, Operating Systems tend not to be implemented in object oriented languages (does anybody remember JavaOS?). Yet despite being implemented in simpler langauges (often C), Operating Systems, in device drivers, offer highly evolved examples of a different realization of class interfaces, derivation, and inheritance.

Whether we are talking about storage, networking, video, or human-interface, the number of available devices is huge and growing. The number and diversity of these devices creates tremendous demands for object oriented code reuse:

- We want the system to behave similarly, no matter what the underlying devices were being used to provide storage, networking, etc. To ensure this, we would like most of the higher level functionality to be implemented in common, higher level modules.
- We would like to minimize the cost of developing drivers to support new devices. This is most easily done if the majority of the functionality is implemented in common code that can be inherrited by the individual drivers.
- As system functionality and performance are improved, we would like to ensure that those benefits accrue not only to new device drivers, but also to older device drivers.

These needs can be satisfied by implementing the higher level functionality (associated with each general class of device) in common code that uses per-device implementations of a standard sub-class driver to operate over a particular device. This requires:

- deriving device-driver sub-classes for each of the major classes of device.
- defining sub-class specific interfaces to be implemented by the drivers for all devices in each of those classes.
- creating per-device implementations of those standard sub-class interfaces.

Major Driver Classes

In the earliest versions of Unix, all devices were divided into two fundamental classes, which are still present in all Unix derivatives:

block devices

> These are random-access devices, addressable in fixed size (e.g. 512 byte, 4K byte) blocks. Their drivers implement a *request* method to enqueue asynchronous DMA requests. The request descriptor included information about the desired operation (e.g. byte count, target device, disk address and in-memory buffer address) as well as completion information (how much data was transferred, error indications) and a condition variable the requestor could use to await the eventual completion of the request.
>
> A read or write request could be issued for any number of blocks, but in most cases a large request would be broken into multiple single-block requests, each of which would be passed, block at a time, through the system buffer cache. For this reason block device drivers also implement a *fsync* method to flush out any buffered writes.

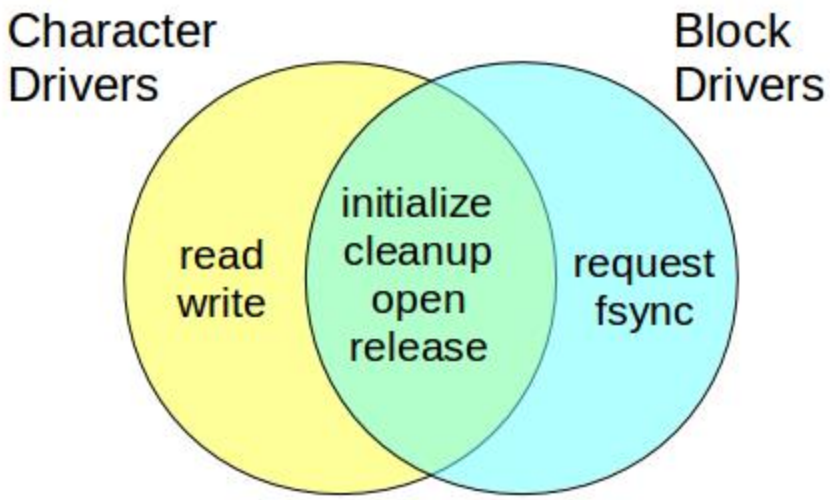
character devices

> These devices may be sequential access, or may be byte-addressable. They support the standard synchronous *read(2)*, *write(2)* and (indirectly) *seek(2)* operations.
>
> For devices that supported 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.

A key point here is that, even in the oldest Unix systems, device drivers were divided into distinct classes (implementing different interfaces) based on the needs of distinct classes of clients:

- Block devices were designed to be used, within the operating system, by file systems, to access disks. Forcing all I/O to go through the system buffer cache is almost surely the right thing to do with file system I/O.
- Character devices were designed to be used directly by applications. The potential for large DMA transfers directly between the device and user-space buffers meant that character (or *raw*) I/O operations might be much more efficient than the corresponding requests to a block device.

These two major classes of device were not mutually exclusive. A single driver could export 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 (potentially much more efficient) character device*. All device drivers support *initialize* and *cleanup* methods (for dynamic module loading and unloading), *open* and *release* methods (roughly corresponding to the *open(2)* and *close(2)* system calls), and an optional catch-all *ioctl(2)* method.

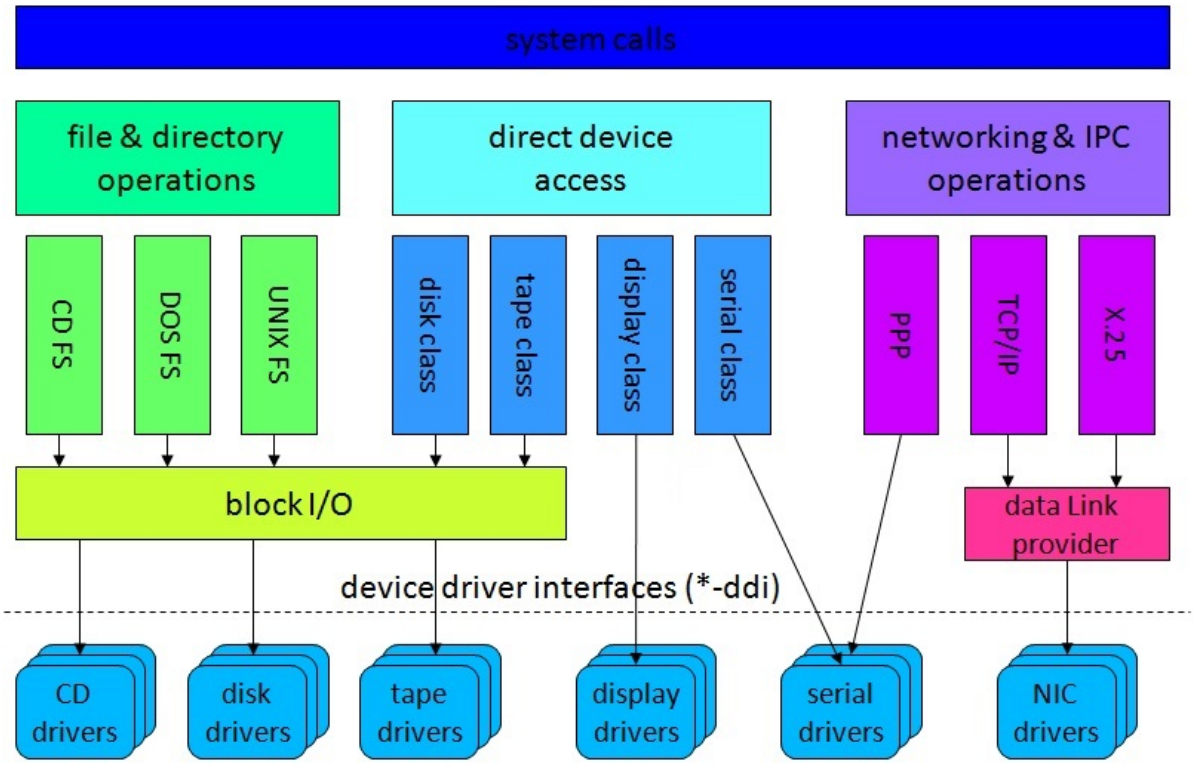


*In more contemporary systems, it is possible for a client to specify that block I/O should not be passed through the system buffer cache.

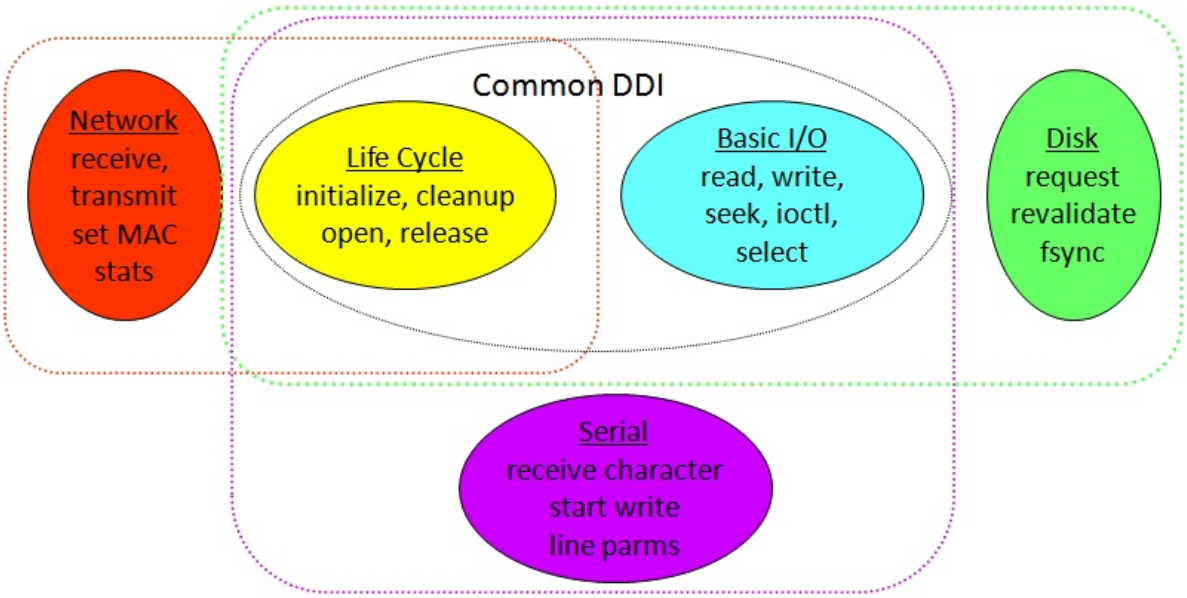
Driver sub-classes

Given the fundamental importance of file systems and disks to operating systems, it is not surprising that block device drivers would have been singled out as a special sub-class in even the earliest versions of Unix. But as system functionality evolved, the operating system began to implement higher level services for other sub-classes of devices:

- 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



And as each of these sub-systems evolved, new device driver methods* were defined to enable more effective communication between the higher level frameworks and the lower level device drivers in each sub-class. Each of these sub-class specific interfaces is referred to as a *Device Driver Interface* (DDI).



*It should be noted that in some cases, the higher frameworks have been implemented in user-mode, so that some of the new interfaces have been specified as behavior rather than new methods.

The rewards for 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.

But the price for these rewards is that all device drivers must implement exactly the same interfaces:

- 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

It is nearly impossible to implement a completely self-contained device driver. Most device drivers are likely to require a range of resources and services from the operating system:

- 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 operating system for use by device drivers is sometimes referred to as the *Driver-Kernel Interface* (DKI). Interface stability for DKI functions is every bit as important as it is for the DDI entry points. If an operating system eliminates or incompatibly changes a DKI function, device drivers that depend on that function may cease working. The requirement to maintain stable DKI entry points may greatly constrain our ability to evolve our operating system implementation. Similar issues arrise for other classes of dynamically loadable kernel modules (such as network protocols and file systems).

Conclusion

Device drivers demonstrate an evolution from a basic super-class (character devices) into an ever-expanding hierarchy of derived sub-classes. But unlike traditional class derivation, where sub-class implementations inherit most of their implemntation from their parent, we see a different sort of inheritance. While each new sub-class and instance is likely to be a new implementation, what they inherit is pre-existing higher level frameworks that do do most of their work for them.

Dynamically Loadable Kernel Modules

Introduction

We often design systems to provide a common framework, but that expect problem-specific implementations to be provided at a later time. This approach is embraced by several standard design patterns (e.g. Strategy, Factory, Plug-In). These approaches have a few key elements in common:

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

In most programs the set of available implementations is locked in at build-time. But many systems have the ability to select and load new implementations at run time as *dynamically loadable modules*. We see this with browser plug-ins. Operating systems may also support many different types of dynamically loadable modules (e.g., file systems, network protocols, device drivers). Device drivers are a particularly important and rich class of dynamically loadable modules ... and therefore a good example to study.

There are several compelling reasons for wanting device drivers to be dynamically loadable:

- the number of possible I/O devices is far too large to build all of them in to the operating system.
- if we want an operating system 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 the associated device is plugged in to the computer.
- new devices become available long after the operating system has been shipped, and so must be *after market* addable.
- most device drivers are developed by the hardware manufacturers, and delivered to customers independently from the operating system.

Choosing Which Module to Load

In the abstract, a program needs an implementation and calls a *Factory* to obtain it. But how does the Factory know what implementation class to instantiate?

- for a browser plugin, there might be a MIME-type assocated with the data to be handled, and the browser can consult a registry to find the plug-in associated with that MIME-type. This is a very general mechanism ... but it presumes that data is tagged with a type, and that somebody is maintaining a tag-to-plugin registery.
- at the other extreme, the Factory could load all of the known plug-ins and call a *probe* method in each to see which (if any) of the plug-ins could identify the data and claim responsibility for handling it.

Long ago, dynamically loadable device drivers used the probing process, but this was both unreliable (might incorrectly accept the wrong device) and dangerious (touching random registers in random devices). Today most I/O busses support self-identifying devices. Each device has type, model, and even serial number information that can be queried in a standard way (e.g., by *walking the configuration space*). This information can be used, in combination with a device driver registry, to automatically select a driver for a

given device. These registries may support precedence rules that can chose the best from among multiple competing drivers (e.g., a generic VGA driver, a GeForce driver, and a GeForce GTX 980 driver).

Loading a New Module

In many cases, 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 many cases (including device drivers and file systems) the dynamically loaded module may need to make use of other functions (e.g., memory allocation, synchronization, I/O) in the program into which it is being added. This means that the module to be loaded will (like an object module) have unresolved external references, and requires a *run-time loader* (a simplified linkage editor) that can look up and adjust all of those references as the new module is being loaded.

Note, however, that these references can only be from the dynamically loaded module into the program into which it is loaded (e.g., the operating system). The main program can never have any direct references into the dynamically loaded module ... because the dynamically loaded module may not always be there.

Initialization and Registration

If the operating system is not allowed to have any direct references into a dynamically loadable module, how can the new module be used? When the run-time loader is invoked to load a new dynamically loadable module, it is common for it to return a vector that contains a pointer to at least one method: an initialization function.

After the module has been loaded into memory, the main program calls its initialization method. For a dynamically loaded device driver, the initialization method might:

- allocate memory and initialize driver data structures.
- allocate I/O resources (e.g., bus addresses, interrupt levels, DMA channels) and assign them to the devices to be managed.
- register all of the device instances it supports. Part of this registration would involve providing a vector (of pointers to standard device drvier entry points) that client software could call in order to use these device instances.

Device instance configuration and initialization is another area where self-identifying devices have made it much easier to implement dynamically loaded device drivers:

- Long ago, devices were configured for particular bus addresses and interrupt levels with mechanical switches, that would be set before the card was plugged in to the bus. These resource assignments would be recorded in a system configuration table, which would be compiled (or read during system start-up) into the operating system, and used to select and configure corresponding device driver instances.
- More contemporary busses (like PCIe or USB) provide mechanisms to discover all of the available devices and learn what resources (e.g., bus addresses, DMA channels, interrupt levels) they require. The device driver can then allocate these resources from the associated bus driver, and assign the required resources to each device.

Using a Dynamically Loaded Module

The operating system will provide some means by which processes can open device instances. In Linux the OS exports a pseudo file system (/dev) that is full of *special files*, each one associated with a registered device instance. When a process attempts to open one of those special files, the operating system creates a reference from the open file instance to the registered device instance. From then on, when ever the process issues a *read(2)*, *write(2)*, or *ioctl(2)* system call on that file descriptor, the operating system forwards that call to the appropriate device driver entry point.

A similar approach is used when higher level frameworks (e.g., terminal sessions, network protocols or file systems) are implemented on top of a device. Each of those services maintains open references to the underlying devices, and when they need to talk to the device (e.g., to queue an I/O request or send a packet) the OS forwards that call to the appropriate device driver entry point.

The system often maintains a table of all registered device instances and the associated device driver entry points for each standard operation. When ever a request is to be made for any device, the operating system 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 implementations 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 Criticality of Stable Interfaces

All of this is completely dependent on stable and well specified interfaces:

- the set of entry-points for any class of device driver must be well defined, and all drivers must compatably implement all of the relevant interfaces.
- the set of functions within the main program (OS) that the dynamically loaded modules are allowed to call must be well defined, and the interfaces to each of those functions must be stable.

If one device driver did not implement a standard entry point in the standard way, clients of that device would not work. Some functionality may be optional, and it may be acceptable for a device driver to refuse some requests. But this may make the application responsible for dealing with some version incompatabilities.

If an operating system does not implement some standard service function (e.g., memory allocation) in the standard way, a device driver written to that interface standard may not work when loaded into the non-compliant operating system.

There is often a tension between the conflicting needs to support new hardware and software features while

retaining compatability with old device drivers.

Hot-Pluggable Devices and Drivers

One of the major advantages of dynamically loadable modules is that they can be loaded at any time; not merely during start-up. Hot-plug busses (e.g., USB) can generate events whenever a device is added to, or removed from the bus. In many systems a 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-plugable busses often have multiple power levels, and a newly inserted device may receive only enough power to enable it to be queried and configured. When the driver is ready to start using the device, it can instruct the bus to fully power the device. Some hot-pluggable busses also have mechanical safety interlocks to prevent a device from being removed while it is still in use. In these cases the driver must shut down and release the device before it can be removed.

Summary

This discussion has focused on dynamically loadable device drivers, but most of the issues (selecting the module to be loaded, dependencies of the loaded module on the host program, initializing and shutting down dynamically loaded modules, binding clients to dynamically loaded modules, and defining and managing the interfaces between the loaded and hosting modules) are applicable to a much wider range of dynamically loadable modules.

Stable and well standardized interfaces are critical to any such framework:

- the methods to be implemented by the dynamically loaded modules.
- the services that can be used by the dynamically loaded modules.

File Types and Attributes

1. Introduction

Many operating systems (including Unix derivatives) attempt to represent all data sources and sinks as files. Files are most commonly discussed as byte streams or one-dimensional arrays, and this is a fair description of much of the file processing done by applications software. But many of those byte-streams contain highly structured data (e.g. a load module, an MPEG-4 video or a database). Some very interesting sources of data (e.g. directories and key-value stores), while serializable, are are not even intended to be processed as byte streams, but with very different APIs. Many files have interesting attributes beyond the data they contain.

This is a brief exploration of the range of different types of data that are commonly gathered under the general heading of *files*.

2. Ordinary Files

Even the simplest files are understood by imposing structure and semantics on a binary byte stream:

- A text file is a byte stream, but when we process it we generally break it into lines (delimited by \n or \r\n), and render it as characters (e.g. according to the ASCII or ISO 8859 character set).
- 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 (in that it is an alternating sequence of section headers and contents) but the different sections represent different parts of a program (code, initialized data, symbol table, ...).
- an MPEG stream is a sequence of audio, video, frames, containing compressed (e.g. Discrete Cosine Transform) program information, which require considerable processing in order to reconstruct the encoded program.

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

2.1 Data Types and Associated Applications

If a file can only be interpreted by a program that understands the *meaning* that has been encoded in the byte stream, our first problem is finding the right program to interpret each file (or byte stream). There are a few general approaches:

- Require the user to specifically invoke the correct command to process the data. This is common in Unix-derived systems:
  - to edit a file you type `vi` *filename*
  - to compile a program you type `gcc` *filename*
- Consult a registry that associates 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 solution presupposes that we know what the type of the file is. There are a few approaches to *classing* files:

- The simplest approach is based on file name suffix (e.g. `.c`, `.png`, `.txt`). This may be simply an organizing convention (e.g. in Unix-derived systems) or a hard-rule (in Windows it may be impossible to process a file that has been renamed to the wrong suffix).
- Another common approach (very popular with Unix-derived systems) is a *magic number* at the start of the file. Each type of file (or even file system) begins with a reserved and registered magic number that identifies the file's type. For more information on this approach, look at the file(1) command.
- In systems that support extended attributes the file type can be an attribute of the file, not dependent on a suffix or magic-number registry.

2.2 File Structure and Operations

Even ASCII text byte streams have structure, and some streams (e.g. an MPEG-4 video) have a 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 (e.g. a text editor or video player). It is meaningful to talk about doing *read(2)* and *write(2)* operations on these files. But not all files are intended to be accessed via these operations. In some cases, however, the structure of the data is not merely an implementation decision, but fundamental to the manner in which the data is intended to be used:

- the earliest databases were indexed sequential files. These were organized into records, each with a unique *index key*. While these files could be processed sequentially (one record at a time), it was more common to *get* and *put* records based on their keys.
- these evolved into Relational databases, accessed via Structured Query Languages.
- the complexity (and non-scalability) of SQL databases gave rise to much simpler (and more scalable) Key-Value Stores accessed only by *get, put,* and *delete* operations.

While all of these wind up being implemented (usually by some middle-ware layer) on top of byte streams, that is certainly not how they are accessed by their clients.

3. Other types of files

It can be argued that all of the above types of ordinary files are still just blobs of data, differing in their binary representations and the operations in terms of which they are written and read back. But there are other types of files that are not merely blobs of data, to be written and re-read.

3.1 Directories

Directories do not contain blobs of client data. Rather they represent name-spaces, the association of names with blobs of data. They are, in this respect, somewhat similar to key-value stores, but they the namespace is much more highly structured, and the operations are quite different. 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 files owned by numerous users, each of whom wants to impose different sharing/privacy constraints on the access to each referenced file.

Because of how important directories are to the system operation and integrity, all directory operations tend to be implemented within the operating system. To ensure the correctness and security of the directory

structure there are only a few supported update operations (e.g. *mkdir(2), rmdir(2), link(2), unlink(2), create(2)* and *open(2)*), each of which involves considerable permission checking and integrity assurance.

But despite these differences, directories exist in same namespace as files, and have the same notions of user/group ownership, and file protection. They can even be accessed (if you know what you are doing) with *open(2)* and *read(2)*.

3.2 Inter-Process Communications Ports

An inter-process communications port (e.g. a pipe) is not so much a container in which data is stored, as it is a channel through which data is passed. That data is exchanged via *write(2)* and *read(2)* system calls on file descriptors that can be manipulated with the *dup(2)* and *close(2)* operations. And in the case of named pipes they can be accessed via the *open(2)* system call.

With directories we saw something that was implemented as on-disk byte streams, but accessed with very different operations. With inter-process communications ports, we have something with a very different implementation, but that is accessed (as a byte stream) with normal file I/O operations.

3.3. I/O Devices

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

Many sequential access devices (e.g. keyboards and printers) are fit naturally into the byte-stream *read(2)/write(2)* model. Other random access devices (e.g. disks) easily fit into a *read(2)/write(2)/seek(2)* access model. Communications interfaces often behave like byte streams (or perhaps message streams) that also support additional control functions (like controlling line speed, setting MAC address, etc), which we can handle with *ioctl(2)* operations.

But not all I/O devices can be fit into a byte-stream access model. Consider a 3D rendering engine comprised of a few thousand GPUs. Rather than deal with this as a byte stream, we simply map gigabytes of display memory and control registers into our address space and maniuplate them directly. With more primitive devices, we may choose to deal directly with digital and analog signals that sample the state of the external world and control external actuators. Beyond the fact that we used the *open(2)* system call to get access to them, these devices may bear no relation to persistent byte streams.

4. File Attributes

Most of the above have treated files as containers (or access portals) for data. But even a file of ASCII text (e.g. this HTML) is more than than the bytes it contains. In addition to *data*, files also have *meta-data* ... data that describes data.

4.1 System Attributes

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

- type: regular, directory, pipe, device, symbolic link, ...
- ownership: identity of the owning user and owning group
- protection: permitted access (read, write execute), by the owner, the owning group, and others

- when the file was created, last updated, last accessed
- size (for regular files) ... the number of bytes in the file (which may be sparse).

Other operating systems may support more or different attributes. What is important about this list is:

- all files have these attributes
- the operating system depends on these attributes (e.g. to correctly implement access control)
- the operating system maintains these attributes

4.2 Extended Attributes

There may be other information (beyond the basic system attributes) that is vitally important to correct file processing. Examples might be:

- 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?

All of the above examples are *meta-data*. They are not part of the file's contents ... but rather descriptive information that may be necessary to properly process the file's contents. There have been two basic approaches taken to supporting extended attributes:

- associate a limited number/size of *name=value* attributes with each file.
- pair each file with one or more shadow files (sometimes called *resource forks*) that contain additional resources and information.

The operating system may even need to make use of extended attributes. If we were required to extend Unix/Linux owner/group/other protection to support generalized Access Control Lists we might choose store the additional information in (protected) extended attributes.

5. Diversity of Semantics

The Posix operations for file and directory operations are standardized and relatively universal (although some file systems do not support some types of links). The Posix standards even include a *readdir(3)* operation for file-system and operating system independent scanning of directories. It is not difficult to write portable software that operates on ordinary files and directories.

The operations on and behavior of other types of files (e.g. inter-process communications ports and devices) are not at all standardized.

While the need for extended attributes is widely recognized, there are many different implementations, and not (yet) any widely accepted standard.

An Introduction to DOS FAT Volume and File Structure

Mark Kampe markk@cs.ucla.edu

1. Introduction

When the first personal computers with disks became available, they were very small (a few megabytes of disk and a few dozen kilobytes of memory). A file system implementation for such machines had to impose very little overhead on disk space, and be small enough to fit in the BIOS ROM. BIOS stands for **BASIC I/O Subsystem**. Note that the first word is all upper-case. The purpose of the BIOS ROM was to provide run-time support for a BASIC interpreter (which is what Bill Gates did for a living before building DOS). DOS was never intended to provide the features and performance of real timesharing systems.

Disk and memory size have increased in the last thirty years, People now demand state-of-the-art power and functionality from their PCs. Despite the evolution that the last decades have seen, old standards die hard. Much as European train tracks maintain the same wheel spacing used by Roman chariots, most modern OSs still support DOS FAT file systems. DOS file systems are not merely around for legacy reasons. The ISO 9660 CDROM file system format is a descendent of the DOS file system.

The DOS FAT file system is worth studying because:

- It is heavily used all over the world, and is the basis for more modern file system (like 9660).
- It provides reasonable performance (large transfers and well clustered allocation) with a very simple implementation.
- It is a very successful example of "linked list" space allocation.

2. 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. MVS volumes reserve the entire first track of the first cylinder for the boot strap.

- 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.

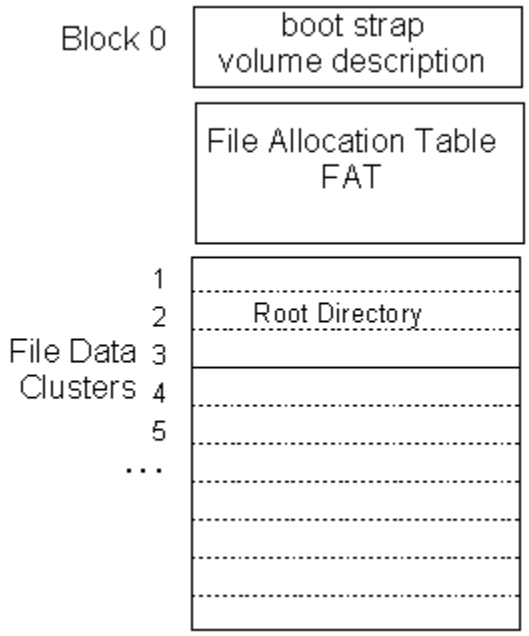
- file name descriptors

data structures that user-chosen names with each file.

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

The first block of DOS FAT volume contains the bootstrap, along with some volume description information. After this comes a much longer **File Allocation Table** (FAT from which the file system takes its name). The File Allocation Table is used, both as a free list, and to keep track of which blocks have been allocated to which files.

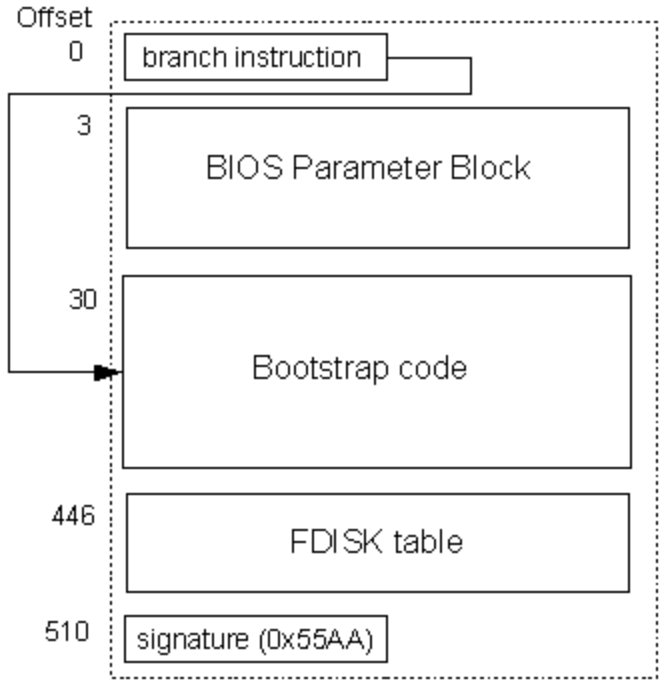
The remainder of the volume is data clusters, which can be allocated to files and directories. The first file on the volume is the root directory, the top of the tree from which all other files and directories on the volume can be reached.



3. Boot block BIOS Parameter Block and FDISK Table

Most file systems separate the first block (pure bootstrap code) from volume description information. DOS file systems often combine these into a single block. The format varies between (partitioned) hard disks and (unpartitioned) floppies, and between various releases of DOS and Windows ... but 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 disk partitioning table
- followed by a signature (for error checking).



3.1 BIOS Parameter Block

After the first few bytes of the bootstrap comes the BIOS parameter block, which 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 laid out on the volume:

- number of sectors per (logical) cluster
- the number of reserved sectors (not part of file system)
- the number of Alternate File Allocation Tables
- the number of entries in the root directory

These parameters enable the OS to interpret the remainder of the file system.

3.2 FDISK Table

As disks got larger, the people at MicroSoft figured out that their customers might want to put multiple file systems on each disk. This meant they needed some way of partitioning the disk into logical sub-disks. To do this, they added a small partition table (sometimes called the FDISK table, because of the program that managed it) to the end of the boot strap block.

This FDISK table has four entries, each capable of describing one disk partition. Each entry includes

- A partition type (e.g. Primary DOS partition, UNIX partition).
- An ACTIVE indication (is this the one we boot from).
- The disk address where that partition starts and ends.

- The number of sectors contained within that 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					

In older versions of DOS the starting/ending addresses were specified as cylinder/sector/head. As disks got larger, this became less practical, and they moved to logical block numbers.

The addition of disk partitioning also changed the structure of the boot record. The first sector of a disk contains the Master Boot Record (MBR) which includes the FDISK table, and a bootstrap that finds the active partition, and reads in its first sector (Partition Boot Record). Most people (essentially everyone but Bill Gates :-) make their MBR bootstrap ask what system you want to boot from, and boot the active one by default after a few seconds. This gives you the opportunity to choose which OS you want to boot. Microsoft makes this decision for you ... you want to boot Windows.

The structure of the Partition Boot Record is entirely operating system and file system specific ... but for DOS FAT file system partitions, it includes a BIOS Parameter block as described above.

4. File Descriptors (directories)

In keeping with their desire for simplicity, 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 (8 characters of base name, plus a 3 character extension).
- a byte of attribute bits for the file, which include:
  - Is this a file, or a 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. (This field is only 16 bits wide, and so when Microsoft introduced the FAT32 file system, they had to put the high order bits in a different part of the directory entry).
- the length (number 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

If the first character of a files name is a NULL (0x00) the directory entry is unused. The special character

(0xE5) in the first character of a file name is used to indicate that a directory entry describes a deleted file. (See the section on Garbage collection below)

Note on times and dates:

DOS stores file modification times and dates as a pair of 16-bit numbers:

- 7 bits of year, 4 bits of month, 5 bits of day of month
- 5 bits of hour, 6 bits of minute, 5 bits of seconds (x2).

All file systems use dates relative to some epoch (time zero). For DOS, the epoch is midnight, New Year's Eve, January 1, 1980. A seven bit field for years means that the the DOS calendar only runs til 2107. Hopefully, nobody will still be using DOS file systems by then :-)

5. Links and Free Space (File Allocation Table)

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 File Allocation Table is a relatively unique design. It 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.

5.1 Cluster Size and Performance

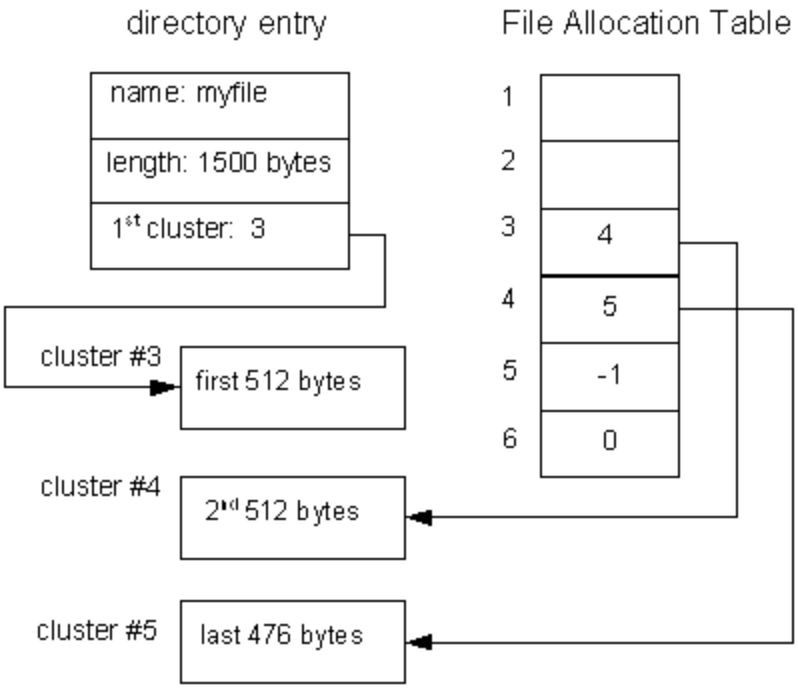
Space is allocated to files, not in (physical) blocks, but in (logical) multi-block clusters. The number of clusters per block is determined when the file system is created.

Allocating space to files in larger chunks improves I/O performance, by 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). As disks have grown larger, people have become less concerned about internal fragmentation losses, and cluster sizes have increased.

The maximum number of clusters a volume can support depends on the width of the FAT entries. In the earliest FAT file systems (designed for use on floppies, and small hard drives). An 8-bit wide FAT entry would have been too small (256 * 512 = 128K bytes) to describe even the smallest floppy, but a 16-bit wide FAT entry would have been ludicrously large (8-16 Megabytes) ... so Microsoft compromised and adopted 12-bit wide FAT entries (two entries in three bytes). These were called FAT-12 file systems. As disks got larger, they created 2-byte wide (FAT-16) and 4-byte wide (FAT-32) file systems.

5.2 Next Block Pointers

A file's directory entry contains a pointer to the first cluster of that file. The File Allocation Table entry for that cluster tells us the cluster number **next** cluster in the file. When we finally get to the last cluster of the file, its FAT entry will contain a -1, indicating that there is no next block in the file.



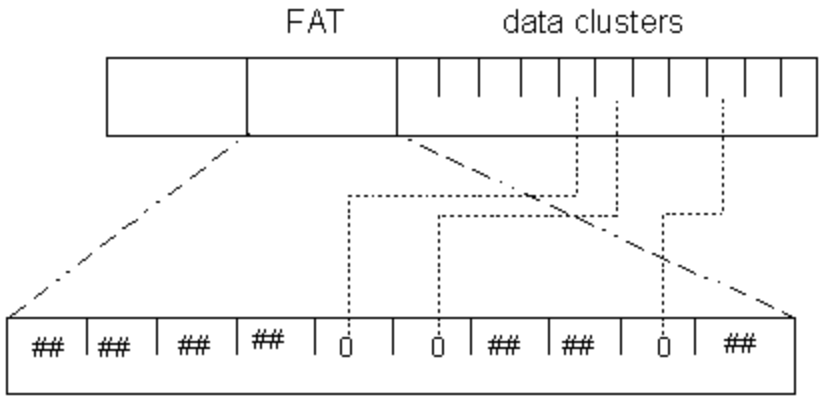
The "next block" organization of the FAT means that in order to figure out what physical cluster is the third logical block of a file, must know the physical cluster number of the second logical block. This is not usually a problem, because almost all file access is sequential (reading the first block, and then the second, and then the third ...).

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 (e.g. 512 bytes per megabyte of file system) that the entire FAT can be kept in memory as long as a file system is in use. This means that successor block numbers can be looked up without the need to do any additional disk I/O. It is easy to imagine

5.3 Free Space

The notion of "next block" is only meaningful for clusters that are allocated to a file ... which leaves us free to use the FAT entries associated with free clusters as a free indication. Just as we reserved a value (-1) to mean **end of file** we can reserve another value (0) to mean **this cluster is free**.

To find a free cluster, one has but to search the FAT for an entry with the value -2. If we want to find a free cluster near the clusters that are already allocated to the file, we can start our search with the FAT entry after the entry for the first cluster in the file.



Each FAT entry corresponds to one data cluster

A FAT entry of 0 means the corresponding data cluster is not allocated to any file.

5.4 Garbage Collection

Older versions of FAT file systems did not bother to free blocks when a file was deleted. Rather, they merely crossed out the first byte of the file name in the directory entry (with the reserved value 0xE5). This had the advantage of greatly reducing the amount of I/O associated with file deletion ... but it meant that DOS file systems regularly ran out of space.

When this happened, they would initiate garbage collection. Starting from the root directory, they would find every "valid" entry. They would 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 File Allocation Table.

This "feature" was probably motivated by a combination of laziness and a desire for performance. It did, however, have an advantage. Since clusters were not freed when files were deleted, they could not be reallocated until after garbage collection was performed. This meant that it might be possible to recover the contents of deleted files for quite a while. The opportunity this created was large enough to enable Peter Norton to start a very successful company.

6. Descendants of the DOS file system

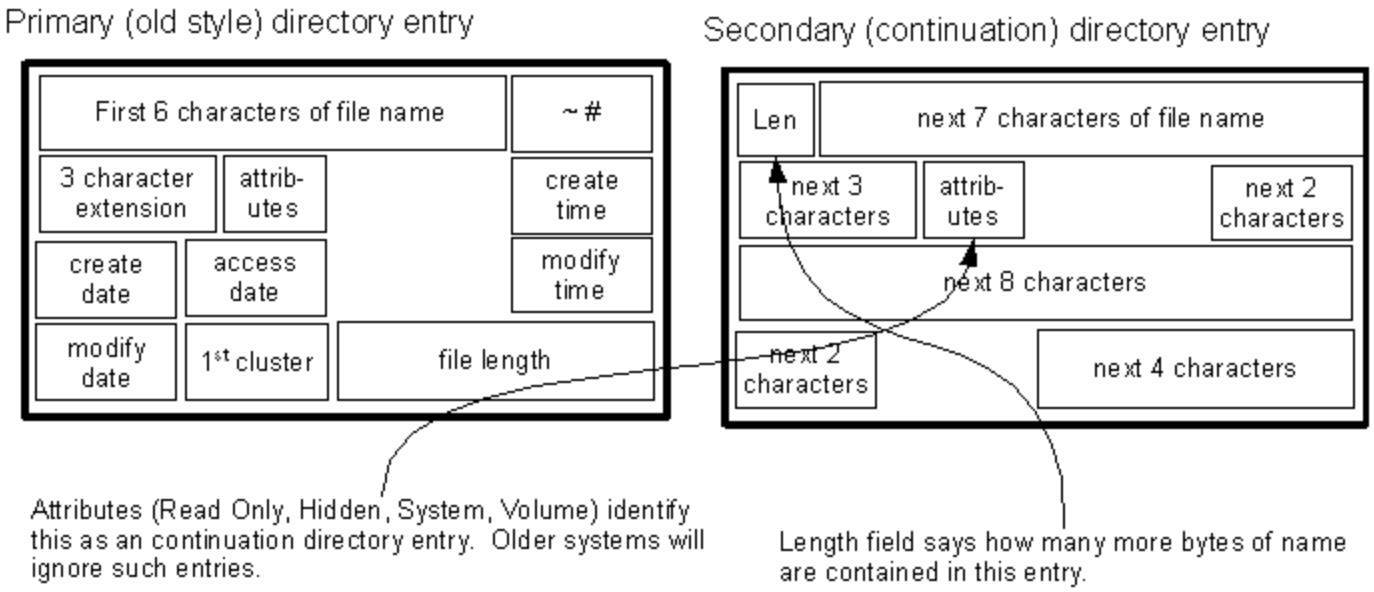
The DOS file system has evolved with time. Not only have wider (16- and 32-bit) FAT entries been used to support larger disks, but other features have been added. The last stand-alone DOS product was DOS 6.x. After this, all DOS support was under Windows, and along with the change to Windows came an enhanced version of the FAT file system called Virtual FAT (or simply VFAT).

6.1 Long File Names

Most DOS and Windows systems were used for personal productivity, and their users didn't demand much in the way of file system features. Their biggest complaints were about the 8+3 file names. Windows users demanded longer and mixed-case file names.

The 32 byte directory entries didn't have enough room to hold longer names, and changing the format of DOS directories would break hundreds or even thousands of applications. It wasn't merely a matter of importing files from old systems to new systems. DOS diskettes are commonly used to carry files between various systems ... which means that old systems still had to be able to read the new directories. They had to find a way to support longer file names without making the new files unreadable by older systems.

The solution they came up was to put the extended filenames in additional (auxiliary) directory entries. Each file would be described by an old-format directory entry, but supplementary directory entries (following the primary directory entry) could provide extensions to the file name. To keep older systems from being confused by the new directory entries, they were tagged with a very unusual set of attributes (hidden, system, read-only, volume label). Older systems would ignore new entries, but would still be able to access new files under their 8+3 names in the old-style directory entries. New systems would recognize the new directory entries, and be able to access files by either the long or the short names.



The addition of long file names did create one problem for the old directory entries. What would you do if you had two file names that differed only after the 8th character (e.g. datafileread.c and datafilewrite.c)? If we just used the first 8 characters of the file name in the old directory entries (datafile), we would have two files with the same name, and this is illegal. For this reason, the the short file names are not merely the first eight characters of the long file names. Rather, the last two bytes of the short name were merely made unambiguous (e.g. "~1", "~2", etc).

6.2 Alternate/back-up FATs

The File Allocation Table is a very concise way of keeping track of all of the next-block pointers in the file system. If anything ever happened to the File Allocation Table, the results would be disastrous. The directory entries would tell us where the first blocks of all files were, but we would have no way of figuring out where the remainder of the data was.

Events that corrupt the File Allocation Table are extremely rare, but the consequences of such an incident are catastrophic. To protect users from such eventualities, MicroSoft added support for alternate FATs. Periodically, the primary FAT would be copied to one of the pre-reserved alternate FAT locations. Then if something bad happened to the primary FAT, it would still be possible to read most files (files created before

the copy) by using the back-up FAT. This is an imperfect solution, as losing new files is bad ... but losing all files is worse.

6.3 ISO 9660

When CDs were proposed for digital storage, everyone recognized the importance of a single standard file system format. Dueling formats would raise the cost of producing new products ... and this would be a lose for everyone. To respond to this need, the International Standards Organization chartered a sub-committee to propose such a standard.

The failings of the DOS file system were widely known by this time, but, as the most widely used file system on the planet, the committee members could not ignore it. Upon examination, it became clear that the most idiomatic features of the DOS file system (the File Allocation Table) were irrelevant to a CDROM file system (which is written only once):

- We don't need to keep track of the free space on a CD ROM. We write each file contiguously, and the next file goes immediately after the last one.
- Because files can be written contiguously, we don't need any "next block" pointers. All we need to know about a file is where its first block resides.

It was decided that ISO 9660 file systems would (like DOS file systems) have tree structured directory hierarchies, and that (like DOS) each directory entry would describe a single file (rather than having some auxiliary data structure like and I-node to do this). 9660 directory entries, like DOS directory entries, contain:

- file name (within the current directory)
- file type (e.g. file or directory)
- location of the file's first block
- number of bytes contained in the file
- time and date of creation

They did, however, learn from DOS's mistakes:

- Realizing that new information would be added to directory entries over time, they made them variable length. Each directory entry begins with a length field (giving the number of bytes in this directory entry, and thus the number of bytes until the next directory entry).
- Recognizing the need to support long file names, they also made the file name field in each entry a variable length field.
- Recognizing that, over time, people would want to associate a wide range of attributes with files, they also created a variable length extended attributes section after the file name. Much of this section has been left unused, but they defined several new attributes for files:
    - file owner
    - owning group
    - permissions
    - creation, modification, effective, and expiration times
    - record format, attributes, and length information

But, even though 9660 directory entries include much more information than DOS directory entries, it remains that 9660 volumes resemble DOS file systems much more than they resemble any other file system format. And so, the humble DOS file system is reborn in a new generation of products.

7. Summary

DOS file systems are very simple, They don't support multiple links to a file, or symbolic links, or even multi-user access control. They are also and very economical in terms of the space they take up. Free block lists, and file block pointers are combined into a single (quite compact) File Allocation Table. File descriptors are incorporated into the directory entries. And for all of these limitations, they are probably the most widely used file system format on the planet. Despite their primitiveness, DOS file systems were used as the basis for much newer CD ROM file system designs.

What can we infer from this? That most users don't need a lot of fancy features, and that the DOS file system (primitive as it may be) covers their needs pretty well.

It is also noteworthy that when Microsoft was finally forced to change the file system format to get past the 8.3 upper case file name limitations, they chose to do so with a kludgy (but upwards compatible) solution using additional directory entries per file. The fact that they chose such an implementation clearly illustrates the importance of maintaining media interchangeability with older systems. This too is a problem that all (successful) file systems will eventually face.

8. References

DOS file system information

- PC Guide's Overview of DOS FAT file systems. (this is a pointer to the long filename article ... but the entire library is nothing short of excellent).
- Free BSD sources, PCFS implementation, BIOS Parameter block format, and (Open Solaris) FDISK table format.
- Free BSD sources, PCFS implementation, Directory Entry format

9660 file system information

- Wikipedia Introduction to ISO 9660 file systems
- Free BSD sources, ISOFS implementation, ISO 9660 data structures.

Filesystem in Userspace

Filesystem in Userspace (**FUSE**) is a software interface for Unix and Unix-like computer operating systems 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 the FUSE module provides only a bridge to the actual kernel interfaces.

FUSE is available for Linux, FreeBSD, OpenBSD, NetBSD (as puffs), OpenSolaris, Minix 3, macOS,[2] and Windows.[3]

FUSE is free software originally released under the terms of the GNU General Public License and the GNU Lesser General Public License.

Filesystem in Userspace

Stable release	3.14.0[1] / 17 February 2023
Repository	github.com/libfuse/libfuse (https://github.com/libfuse/libfuse)
Written in	C
Operating system	Unix, Unix-like
Type	File system driver
License	GPL for Linux kernel part, LGPL for Libfuse, Simplified BSD on FreeBSD, ISC license on OpenBSD; proprietary for macOS
Website	github.com/libfuse/libfuse (https://github.com/libfuse/libfuse)

History

The FUSE system was originally part of *AVFS* (*A Virtual Filesystem*), a filesystem implementation heavily influenced by the translator concept of the GNU Hurd.[4] It superseded Linux Userland Filesystem, and provided a translational interface using `lufis` in libfuse1.

FUSE was originally released under the terms of the GNU General Public License and the GNU Lesser General Public License, later also reimplemented as part of the FreeBSD base system[5] and released under the terms of Simplified BSD license. An ISC-licensed re-implementation by Sylvestre Gallon was released in March 2013,[6] and incorporated into OpenBSD in June 2013.[7]

FUSE was merged into the mainstream Linux kernel tree in kernel version 2.6.14.[8]

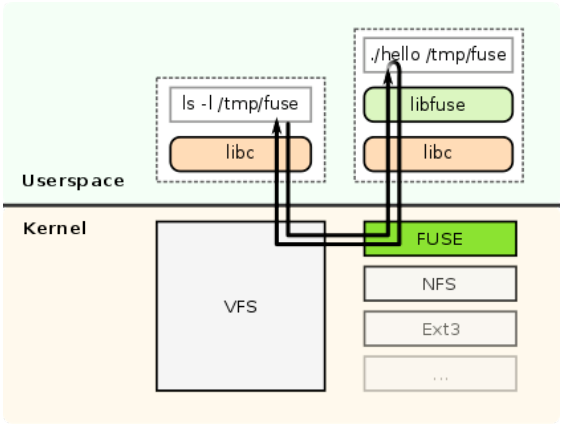
The userspace side of FUSE, the `libfuse` library, generally followed the pace of Linux kernel development while maintaining "best effort" compatibility with BSD descendants. This is possible because the kernel FUSE reports its own "feature levels", or versions. The exception is the FUSE fork for macOS, OSXFUSE, which has too many differences for sharing a library.[9] A break in libfuse history is libfuse3, which includes some incompatible improvements in the interface and performance, compared to the older libfuse2 now under maintenance mode.[10]

As the kernel-userspace protocol of FUSE is versioned and public, a programmer can choose to use a different piece of code in place of `libfuse` and still communicate with the kernel's FUSE facilities. On the other hand, `libfuse` and its many ports provide a portable high-level interface that may be

implemented on a system without a "FUSE" facility.

Operation and usage

To implement a new file system, a handler program linked to the supplied `libfuse` library needs to be written. The main purpose of this program is to specify how the file system is to respond to read/write/stat requests. 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 for this newly mounted file system, the kernel forwards these IO-requests to the handler and then sends the handler's response back to the user.

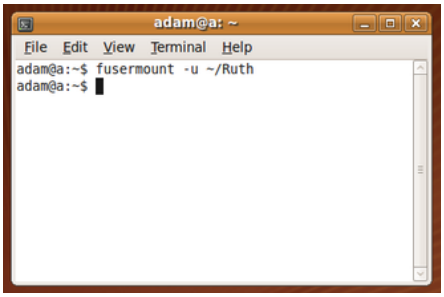


A flow-chart diagram showing how FUSE works: Request from userspace to list files (ls -l /tmp/fuse) gets redirected by the Kernel through VFS to FUSE. FUSE then executes the registered handler program (./hello) and passes it the request (ls -l /tmp/fuse). The handler program returns a response back to FUSE which is then redirected to the userspace program that originally made the request.

FUSE is particularly useful for writing virtual file systems. Unlike traditional file systems that essentially work with data on mass storage, virtual filesystems don't actually store data themselves. They act as a view or translation of an existing file system or storage device.

In principle, any resource available to a FUSE implementation can be exported as a file system.

Applications



Unmounting a FUSE-based file system with the `fusermount` command

On-disk file systems

Conventional on-disk file systems can be implemented in user space with FUSE, e.g. for compatibility or licensing reasons.

- Linear Tape File System: Allows files stored on magnetic tape to be accessed in a similar fashion to those on disk or removable flash drives.
- NTFS-3G and Captive NTFS, allowing access to NTFS filesystems.
- retro-fuse (https://github.com/jaylogue/retro-fuse): retro-fuse is a user-space filesystem that provides a way to mount filesystems created by ancient Unix systems on modern OSes. The current version of retro-fuse supports mounting filesystems created by Fifth, Sixth and Seventh Edition of Research Unix from BTL, as well as 2.9BSD and 2.11BSD based systems.

Layering file systems

FUSE filesystems can create a view of an underlying file system, transforming the files in some way.

Distributed Systems Goals & Challenges

Before we start discussing distributed systems architectures it is important to understand why we have been driven to build distributed systems, and the fundamental problems associated with doing so.

Goals: Why Build Distributed Systems

Distributed systems could easily be justified by the simple facts of collaboration and sharing. The world-wide web is an obvious and compelling example of the value that is created when people can easily expose and exchange information. Much of the work we do is done in collaboration with others, and so we need often need to share work products. But there are several other powerful forces driving us towards distributed systems.

Client/Server Usage Model

Long ago, all the equipment I needed to use in connection with my computer was connected to my computer. But this does not always make sense:

- I may only use a high resolution color scanner for a few minutes per month.
- I may make regular use of a high speed color printer, but not enough to justify buying one just for myself.
- I could store my music and videos on my own computer, but if I store them on a home NAS server, the entire family can have access to our combined libraries.
- I could store all my work related files on my own computer, but if I store them on a work-group server, somebody else will manage the back-ups, and ensure that everyone on the project has access to them.

There are many situations where we can get better functionality and save money by using remote/centralized resources rather than requiring all resources to be connected to a client computer.

Reliability and Availability

As we come to depend more and more on our digital computers and storage we require higher reliability and availability from them. Long ago people tried to improve reliability by building systems out of the best possible components. But, as with RAID, we have learned that we can obtain better reliability by combining multiple (very ordinary systems). If we have more computing and storage capacity than we actually need, we may be able to continue providing service, even after one or more of our servers have failed.

The key is to distribute service 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. If the client and server instances are to be distributed across multiple independent computers, then we are building a distributed system.

Scalability

It is common to start any new project on a small system. If the system is successful, we will probably add more work to it over time. This means we will need more storage capacity, more network bandwidth, and more computing power. System manufacturers would be delighted if, each time we needed more capacity and power, we bought a new (larger, more expensive) computer (and threw away the old one). But

a. This is highly inefficient, as we are essentially throwing away old capacity in order to buy new capacity.
b. If we are successful, our needs for capacity and power will eventually exceed even the largest computer.

A more practical approach would be to design systems that can be expanded incrementally, by adding additional computers and storage as they were needed. And, again, if our growth plan is to *scale-out* (rather than *scale-up*) we are going to be building our system out of multiple independent computers, and so we are building a distributed system.

Flexibility

We may start building and testing all the parts of a new service on a notebook or desktop, but later we may decide that we need to run different parts on different computers, or a single part on multiple computers. If our the components of our service interact with one-another through network protocols, it will likely be very easy to change the deployment model (which services run on which computers). Distributed systems tend to be very flexible in this respect.

Challenges: Why are Distributed Systems Hard to Build

The short answer is that there are two reasons:

- Many solutions that work on single systems, do not work in distributed systems.
- Distributed systems have new problems that were never encountered in single systems.

New and More Modes of Failure

If something bad happens to a single system (e.g. the failure of a disk or power supply) the whole system goes down. Having all the software fail at the same time is bad for service availability, but we don't have to worry about how some components can continue operating after others have failed. Partial failures are common in distributed systems:

- one node can crash while others continue running
- occasional network messages may be delayed or lost
- a switch failure may interrupt communication between some nodes, but not others

Distributed systems introduce many new problems that we might never have been forced to address in single systems:

- In a single system it may be very easy to tell that one of the service processes has died (e.g. the process exited with a fatal signal or error return code). In a distributed system our only indication that a component has failed might be that we are no longer receiving messages from it. Perhaps it has failed, or perhaps it is only slow, or perhaps the network link has failed, or perhaps our own network interface has failed. Problems are much more difficult to diagnose in a distributed system, and if we incorrectly diagnose a problem we are likely to choose the wrong solution.
- 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 (eg. greater error checking, automatic fail-over, recovery and connection reestablishment). One particularly tricky part of recovery is how to handle situations where a failed component was holding resource locks. We must find some way of recognizing the problem and breaking the locks. And after we have broken the locks we need some

way of (a) restoring the resource to a clean state and (b) preventing the previous owner from attempting to continue using the resource if he returns.

Complexity of Distributed State

Within a single computer system all system resource updates are correctly serialized and we can:

- place all operations on a single time-time (a total ordering)
- at any moment, say what the state of every resource in the system is.

Neither of these is true in a distributed system:

- 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 A happened before or after a different operation on node B. And even when operations are performed via message exchanges, two nodes may disagree on the relative ordering of two events (depending on the order in which each node received the messages).
- Because of the independence of parallel events, different nodes may at any given instant, consider a single 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.

In single systems, when we needed before-or-after atomicity, we created a single mutex (perhaps in the operating system, or in memory shared by all contending threads). A similar effect can be achieved by sending messages to a central coordinator ... except that those messages are roughly a million times as expensive as operations on an in-memory mutex. This means that serialization approaches that worked very well in a single system can become prohibitively expensive in a distributed system.

Complexity of Management

In a single computer system has a single configuration. A thousand different systems may each be configured differently:

- they may have different databases of known users
- their services may be configured with different options
- they may have different lists of which servers perform which functions
- their switches may be configured with different routing and fire-wall rules

And even if we create a distributed management service to push management updates out to all nodes:

- 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 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. More nodes mean more messages, which may result in increased overhead, and longer delays. Increased overhead may result in poor scaling, or even in performance that drops as the system size grows. Longer (and more variable) delays often 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

and the components interact with one-another through a variety of different networks and file systems. The combinatorics and constant evolution of possible component versions and interconnects render exhaustive testing impossible. These challenges often give rise to interoperability problems and unfortunate interactions that would never happen in a single (homogeneous) system.

Emergent phenomena

The human mind renders complex systems understandable by constructing simpler abstract models. But simple models (almost by definition) cannot fully capture the behavior of a complex system. Complex systems often exhibit *emergent behaviors* that were not present in the constituent components, but arise from their interactions at scale (e.g. delay-induced oscillations in under-damped feed-back loops). If these phenomena do not happen in smaller systems, we can only learn about them through (hard) experience.

Peter Deutsch's "Seven Fallacies" of Distributed Computing

In 1994, Peter Deutsch looked back at a decade of failed distributed computing projects and attempted to distill lessons. He came to the conclusion that many projects had failed because of a few naive assumptions that proved to be disastrously false. He enumerated them in an often cited paper:

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 susceptible 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).

Lease-Based Serialization

Solutions developed for single systems often prove inadequate to embrace the added complexities of distributed systems. This is certainly the case with critical-section synchronization in systems with distributed actors. Leases represent a very different, and much more robust approach to serialization.

Challenges of Distributed Locking

Among Deutsch's Seven Fallacies of distributed computing were:

- zero latency
- reliable delivery
- stable topology
- consistent management

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".

Other issues include:

- In a single node it was possible to use some combination of atomic instructions and interrupt disables to prevent parallelism within critical sections. There are no WAN-scale atomic instructions or interrupt disables.
- In a single system, we might understand the resources well enough to be able to assign a total ordering to all resources, and so prevent circular dependencies (and thereby deadlocks). In a distributed system the set of possible resources may not be orderable, or even known, which eliminates ordering as a practical means of deadlock prevention.

Addressing these Challenges

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.

The more complex failure cases and greater deadlock risks can be dealt with by replacing *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. But in practice, there is one other very important difference: Locks work on an *honor system*. An actor who does not yet have a lock will not enter the critical section that the lock protects. Leases are often enforced. 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:

- the release message was lost in transit
- the owning process has crashed
- the node on which the owner was running has crashed
- the owning process is running slowly
- the network connection to the owning node has failed

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). But we do have (at least) two issues:

1. 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. If a lease expires before the operation is committed, the resource should fall back to its last consistent state.
2. The choice of the lease period may involve some careful tuning. If the lease period is short, an owner may have to renew it many times to complete a session. If the lease period is long, it may take the system a long time to recover from a failure.

Sending a network message for every entry into a short critical section would be disastrous. On the other hand, using leases for long lived resources (like DHCP allocated IP addresses) can be extremely economical. It comes down to:

- the number of operations that can be performed under a single lease-grant
- the ratio of the costs of obtaining the lease to the costs of the operations that will be performed under that lease

A three second delay to obtain an IP address is trivial if the IP address can be used for 24 hours thereafter. A one second delay to get a lock on a file might amortize down to nothing if we could then use that lock to do one million writes.

Evaluating Leases

Mutual Exclusion
: Leases are at least as good as locks, with the additional benefit of potential enforcement.

Fairness
: This depends on the policies implemented by the remote lock manager, who could easily implement

either queued or prioritized requests.

Performance
: Remote operations are, by their very nature expensive ... but we probably aren't going to network leases for local objects. If lease requests are rare and cover large numbers of operations, this can be a very efficient mechanism.

Progress
: The good news is that automatic preemption makes leases immune to deadlocks! But, if a lease-holder dies, other would-be lessees must wait for the lease period to expire.

Robustness
: This was never mentioned as a criterion when we were evaluating single system synchronization mechanisms, but leases are clearly more robust than those single-system mechanisms.

Leases are not without their problems:

- while they easily recover from client failures, correct (highly stateful) recovery from lock-server failures is extremely complex.
- automatic lease expiration is a very powerful feature, but it raises the issue of how to decide "what time it is" in a distributed system without a universal time standard.

These are interesting problems, with interesting solutions, but well beyond the scope of this introductory course.

Opportunistic Locks

For most resources, contention is rare, and the locking code is there only to ensure correct behavior in unlikely situations. It seems unfair to have to pay the high cost of remote lock requests to deal with an unlikely problem. 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.

Summary

Locks are a fundamental concept, but one from a simpler time. They are ignorant of Deutsch's Seven Fallacies, and do not work well in distributed systems. Leases are a richer and more expensive mechanism that more robustly embraces these more complex situations. These capabilities make them interesting even in single-node applications where locking must be robust in the face of an ever-changing set of clients.

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 coordinating processes to reach **consensus**, or agree on some data value that is needed during computation. Example applications of consensus include agreeing on what transactions to commit to a database in which order, state machine replication, and atomic broadcasts. Real-world applications often requiring consensus include cloud computing, clock synchronization, PageRank, opinion formation, smart power grids, state estimation, control of UAVs (and multiple robots/agents in general), load balancing, blockchain, and others.

Problem description

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. The processes must somehow put forth their candidate values, communicate with one another, and agree on a single consensus value.

The consensus problem is a fundamental problem in control of multi-agent systems. One approach to generating consensus is for all processes (agents) to agree on a majority value. In this context, a majority requires at least one more than half of available votes (where each process is given a vote). However, one or more faulty processes may skew the resultant outcome such that consensus may not be reached or reached incorrectly.

Protocols that solve consensus problems are designed to deal with limited numbers of faulty processes. These protocols must satisfy a number of requirements to be useful. For instance, a trivial protocol could have all processes output binary value 1. This is not useful and thus the requirement is modified such that the output must somehow depend on the input. That is, the output value of a consensus protocol must be the input value of some process. Another requirement is that a process may decide upon an output value only once and this decision is irrevocable. A process is called correct in an execution if it does not experience a failure. 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.

Variations on the definition of *integrity* may be appropriate, according to the application. For example, a weaker type of integrity would be for the decision value to equal a value that some correct process proposed – not necessarily all of them.[1] There is also a condition known as **validity** in the literature which refers to the property that a message sent by a process must be delivered.[1]

A protocol that can correctly guarantee consensus amongst n processes of which at most t fail is said to be *t-resilient*.

In evaluating the performance of consensus protocols two factors of interest are *running time* and *message complexity*. Running time is given in Big O notation in the number of rounds of message exchange as a function of some input parameters (typically the number of processes and/or the size of the input domain). Message complexity refers to the amount of message traffic that is generated by the protocol. Other factors may include memory usage and the size of messages.

Models of computation

Varying models of computation may define a "consensus problem". Some models may deal with fully connected graphs, while others may deal with rings and trees. In some models message authentication is allowed, whereas in others processes are completely anonymous. Shared memory models in which processes communicate by accessing objects in shared memory are also an important area of research.

Communication channels with direct or transferable authentication

In most models of communication protocol participants communicate through *authenticated channels*. This means that messages are not anonymous, and receivers know the source of every message they receive. Some models assume a stronger, *transferable* form of authentication, where each *message* is signed by the sender, so that a receiver knows not just the immediate source of every message, but the participant that initially created the message. This stronger type of authentication is achieved by digital signatures, and when this stronger form of authentication is available, protocols can tolerate a larger number of faults.[2]

The two different authentication models are often called *oral communication* and *written communication* models. In an oral communication model, the immediate source of information is known, whereas in stronger, written communication models, every step along the receiver learns not just the immediate source of the message, but the communication history of the message.[3]

Inputs and outputs of consensus

In the most traditional **single-value** consensus protocols such as Paxos, cooperating nodes agree on a single value such as an integer, which may be of variable size so as to encode useful metadata such as a transaction committed to a database.

A special case of the single-value consensus problem, called **binary consensus**, restricts the input, and hence the output domain, to a single binary digit {0,1}. While not highly useful by themselves, binary consensus protocols are often useful as building blocks in more general consensus protocols, especially for asynchronous consensus.

In **multi-valued** consensus protocols such as Multi-Paxos and Raft, the goal is to agree on not just a single value but a series of values over time, forming a progressively-growing history. While multi-valued consensus may be achieved naively by running multiple iterations of a single-valued consensus protocol in succession, many optimizations and other considerations such as reconfiguration support can make multi-valued consensus protocols more efficient in practice.

Crash and Byzantine failures

There are two types of failures a process may undergo, a crash failure or a Byzantine failure. A *crash failure* occurs when a process abruptly stops and does not resume. *Byzantine failure*s are failures in which absolutely no conditions are imposed. For example, they may occur as a result of the malicious actions of an adversary. A process that experiences a Byzantine failure may send contradictory or conflicting data to other processes, or it may sleep and then resume activity after a lengthy delay. Of the two types of failures, Byzantine failures are far more disruptive.

Thus, a consensus protocol tolerating Byzantine failures must be resilient to every possible error that can occur.

A stronger version of consensus tolerating Byzantine failures is given by strengthening the Integrity constraint:

Integrity

If a correct process decides v, then v must have been proposed by some correct process.

Asynchronous and synchronous systems

The consensus problem may be considered in the case of asynchronous or synchronous systems. While real world communications are often inherently asynchronous, it is more practical and often easier to model synchronous systems,[4] given that asynchronous systems naturally involve more issues than synchronous ones.

In synchronous systems, it is assumed that all communications proceed in *rounds*. In one round, a process may send all the messages it requires, while receiving all messages from other processes. In this manner, no message from one round may influence any messages sent within the same round.

The FLP impossibility result for asynchronous deterministic consensus

In a fully asynchronous message-passing distributed system, in which at least one process may have a *crash failure*, it has been proven in the famous 1985 **FLP impossibility result** by Fischer, Lynch and Paterson that a deterministic algorithm for achieving consensus is impossible.[5] This impossibility result derives from worst-case scheduling scenarios, which are unlikely to occur in practice except in adversarial situations such as an intelligent denial-of-service attacker in the network. In most normal situations, process scheduling has a degree of natural randomness.[4]

In an asynchronous model, some forms of failures can be handled by a synchronous consensus protocol. For instance, the loss of a communication link may be modeled as a process which has suffered a Byzantine failure.

Randomized consensus algorithms can circumvent the FLP impossibility result by achieving both safety and liveness with overwhelming probability, even under worst-case scheduling scenarios such as an intelligent denial-of-service attacker in the network.[6]

Permissioned versus permissionless consensus

Consensus algorithms traditionally assume that the set of participating nodes is fixed and given at the outset: that is, that some prior (manual or automatic) configuration process has **permissioned** a particular known group of participants who can authenticate each other as members of the group. In the absence of such a well-defined, closed group with authenticated members, a Sybil attack against an open consensus group can defeat even a Byzantine consensus algorithm, simply by creating enough virtual participants to overwhelm the fault tolerance threshold.

A **permissionless** consensus protocol, in contrast, allows anyone in the network to join dynamically and participate without prior permission, but instead imposes a different form of artificial cost or barrier to entry to mitigate the Sybil attack threat. Bitcoin introduced the first permissionless consensus protocol using proof of work and a difficulty adjustment function, in which participants compete to solve cryptographic hash puzzles, and probabilistically earn the right to commit blocks and earn associated rewards in proportion to their invested computational effort. Motivated in part by the high energy cost of this approach, subsequent permissionless consensus protocols have proposed or adopted other alternative participation rules for Sybil attack protection, such as proof of stake, proof of space, and proof of authority.

Equivalency of agreement problems

Three agreement problems of interest are as follows.

Terminating Reliable Broadcast

A collection of n processes, numbered from 0 to $n-1$, communicate by sending messages to one another. Process 0 must transmit a value v to all processes such that:

1. if process 0 is correct, then every correct process receives v
2. for any two correct processes, each process receives the same value.

It is also known as The General's Problem.

Consensus

Formal requirements for a consensus protocol may include:

- *Agreement*: All correct processes must agree on the same value.
- *Weak validity*: For each correct process, its output must be the input of some correct process.
- *Strong validity*: If all correct processes receive the same input value, then they must all output that value.
- *Termination*: All processes must eventually decide on an output value

Weak Interactive Consistency

For n processes in a partially synchronous system (the system alternates between good and bad periods of synchrony), each process chooses a private value. The processes communicate with each other by rounds to determine a public value and generate a consensus vector with the following

requirements:[7]

1. if a correct process sends v, then all correct processes receive either v or nothing (integrity property)
2. all messages sent in a round by a correct process are received in the same round by all correct processes (consistency property).

It can be shown that variations of these problems are equivalent in that the solution for a problem in one type of model may be the solution for another problem in another type of model. For example, a solution to the Weak Byzantine General problem in a synchronous authenticated message passing model leads to a solution for Weak Interactive Consistency.[8] An interactive consistency algorithm can solve the consensus problem by having each process choose the majority value in its consensus vector as its consensus value.[9]

Solvability results for some agreement problems

There is a t-resilient anonymous synchronous protocol which solves the Byzantine Generals problem,[10][11] if $\frac{t}{n} < \frac{1}{3}$ and the Weak Byzantine Generals case[8] where t is the number of failures and n is the number of processes.

For systems with n processors, of which f are Byzantine, it has been shown that there exists no algorithm that solves the consensus problem for $n \leq 3f$ in the *oral-messages model*.[12] The proof is constructed by first showing the impossibility for the three-node case $n = 3$ and using this result to argue about partitions of processors. In the *written-messages model* there are protocols that can tolerate $n = f + 1$.[2]

In a fully asynchronous system there is no consensus solution that can tolerate one or more crash failures even when only requiring the non triviality property.[5] This result is sometimes called the FLP impossibility proof named after the authors Michael J. Fischer, Nancy Lynch, and Mike Paterson who were awarded a Dijkstra Prize for this significant work. The FLP result has been mechanically verified to hold even under fairness assumptions.[13] However, FLP does not state that consensus can never be reached: merely that under the model's assumptions, no algorithm can always reach consensus in bounded time. In practice it is highly unlikely to occur.

Some consensus protocols

The Paxos consensus algorithm by Leslie Lamport, and variants of it such as Raft, are used pervasively in widely deployed distributed and cloud computing systems. These algorithms are typically synchronous, dependent on an elected leader to make progress, and tolerate only crashes and not Byzantine failures.

An example of a polynomial time binary consensus protocol that tolerates Byzantine failures is the Phase King algorithm[14] by Garay and Berman. The algorithm solves consensus in a synchronous message passing model with n processes and up to f failures, provided $n > 4f$. In the phase king algorithm, there are $f + 1$ phases, with 2 rounds per phase. Each process keeps track of its preferred output (initially equal to the process's own input value). In the first round of each phase each process broadcasts its own preferred value to all other processes. It then receives the values from all processes and determines which value is the majority value and its count. In the second round of

the phase, the process whose id matches the current phase number is designated the king of the phase. The king broadcasts the majority value it observed in the first round and serves as a tie breaker. Each process then updates its preferred value as follows. If the count of the majority value the process observed in the first round is greater than $n/2 + f$, the process changes its preference to that majority value; otherwise it uses the phase king's value. At the end of $f + 1$ phases the processes output their preferred values.

Google has implemented a distributed lock service library called Chubby.[15] Chubby maintains lock information in small files which are stored in a replicated database to achieve high availability in the face of failures. The database is implemented on top of a fault-tolerant log layer which is based on the Paxos consensus algorithm. In this scheme, Chubby clients communicate with the Paxos *master* in order to access/update the replicated log; i.e., read/write to the files.[16]

Many peer-to-peer online Real-time strategy games use a modified Lockstep protocol as a consensus protocol in order to manage game state between players in a game. Each game action results in a game state delta broadcast to all other players in the game along with a hash of the total game state. Each player validates the change by applying the delta to their own game state and comparing the game state hashes. If the hashes do not agree then a vote is cast, and those players whose game state is in the minority are disconnected and removed from the game (known as a desync.)

Another well-known approach is called MSR-type algorithms which have been used widely from computer science to control theory.[17][18][19]

Source	Synchrony	Authentication	Threshold	Rounds	Notes
Pease-Shostak-Lamport [10]	Synchronous	Oral	$n > 3f$	$f + 1$	total communication $O(n^f)$
Pease-Shostak-Lamport [10]	Synchronous	Written	$n > f + 1$	$f + 1$	total communication $O(n^f)$
Ben-Or [20]	Asynchronous	Oral	$n > 5f$	$O(2^n)$ (expected)	expected $O(1)$ rounds when $f < \sqrt{n}$
Dolev et al.[21]	Synchronous	Oral	$n > 3f$	$2f + 3$	total communication $O(f^3 \log f)$
Dolev-Strong [2]	Synchronous	Written	$n > f + 1$	$f + 1$	total communication $O(n^2)$
Dolev-Strong [2]	Synchronous	Written	$n > f + 1$	$f + 2$	total communication $O(nf)$
Feldman-Micali [22]	Synchronous	Oral	$n > 3f$	$O(1)$ (expected)	
Katz-Koo [23]	Synchronous	Written	$n > 2f$	$O(1)$ (expected)	Requires PKI
PBFT [24]	Asynchronous (safety)-- Synchronous (liveness)	Oral	$n > 3f$		
HoneyBadger [25]	Asynchronous	Oral	$n > 3f$	$O(\log n)$ (expected)	per tx communication $O(n)$ - requires public-key encryption
Abraham et al.[26]	Synchronous	Written	$n > 2f$	8	
Byzantine Agreement Made Trivial [27][28]	Synchronous	Signatures	$n > 3f$	9 (expected)	Requires digital signatures

[Permissionless consensus protocols]

Bitcoin uses proof of work, a difficulty adjustment function and a reorganization function to achieve permissionless consensus in its open peer-to-peer network. To extend Bitcoin's blockchain or distributed ledger, *miners* attempt to solve a cryptographic puzzle, where probability of finding a solution is proportional to the computational effort expended in hashes per second. The node that first solves such a puzzle has their proposed version of the next block of transactions added to the ledger and eventually accepted by all other nodes. As any node in the network can attempt to solve the proof-of-work problem, a Sybil attack is infeasible in principle unless the attacker has over 50% of the computational resources of the network.

Other cryptocurrencies (i.e. NEO, STRATIS, ...) use proof of stake, in which nodes compete to append blocks and earn associated rewards in proportion to *stake*, or existing cryptocurrency allocated and locked or *staked* for some time period. One advantage of a 'proof of stake' over a 'proof of work' system, is the high energy consumption demanded by the latter. As an example, Bitcoin mining (2018) is estimated to consume non-renewable energy sources at an amount similar

to the entire nations of Czech Republic or Jordan.[29]

Some cryptocurrencies, such as Ripple, use a system of validating nodes to validate the ledger. This system used by Ripple, called Ripple Protocol Consensus Algorithm (RPCA), works in rounds: Step 1: every server compiles a list of valid candidate transactions; Step 2: each server amalgamates all candidates coming from its Unique Nodes List (UNL) and votes on their veracity; Step 3: transactions passing the minimum threshold are passed to the next round; Step 4: the final round requires 80% agreement[30]

Other participation rules used in permissionless consensus protocols to impose barriers to entry and resist sybil attacks include proof of authority, proof of space, proof of burn, or proof of elapsed time.

Contrasting with the above permissionless participation rules, all of which reward participants in proportion to amount of investment in some action or resource, proof of personhood protocols aim to give each real human participant exactly one unit of voting power in permissionless consensus, regardless of economic investment.[31][32] Proposed approaches to achieving one-per-person distribution of consensus power for proof of personhood include physical pseudonym parties,[33] social networks,[34] pseudonymized government-issued identities,[35] and biometrics.[36]

Consensus number

To solve the consensus problem in a shared-memory system, concurrent objects must be introduced. A concurrent object, or shared object, is a data structure which helps concurrent processes communicate to reach an agreement. Traditional implementations using critical sections face the risk of crashing if some process dies inside the critical section or sleeps for an intolerably long time. Researchers defined wait-freedom as the guarantee that the algorithm completes in a finite number of steps.

The **consensus number** of a concurrent object is defined to be the maximum number of processes in the system which can reach consensus by the given object in a wait-free implementation.[37] Objects with a consensus number of n can implement any object with a consensus number of n or lower, but cannot implement any objects with a higher consensus number. The consensus numbers form what is called Herlihy's hierarchy of synchronization objects.[38]

Consensus number	Objects
1	atomic read/write registers, mutex
2	test-and-set, swap, fetch-and-add, wait-free queue or stack
...	...
$2n-2$	n-register assignment
...	...
∞	compare-and-swap, load-link/store-conditional,[39] memory-to-memory move and swap, queue with peek operation, fetch&cons, sticky byte

According to the hierarchy, read/write registers cannot solve consensus even in a 2-process system.