The not so official novella to Operating Systems Principles

CS 111

By Jahan Kuruvilla Cherian

Disclaimer: These notes were created by myself in my personal journey through the righteous path of CS111 at UCLA. The material in this text is a culmination of my own understanding, copied text and borrowed images from Professor Mark Kampe's slides and Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Please use these notes as a secondary guide to what you are already studying and not as the primary Bible for Operating Systems Principles. There may be points where I may be incorrect or may have not reached a conclusion fast enough. In such a case please email me at jcherian@ucla.edu with any concerns.

"Worthy adversaries attract interesting people, and Operating Systems are by far man's greatest adversaries." – Professor Mark Kampe (kind of)

WEEK 1

Monday:

When a computer program runs it executes somewhere near a billion instructions per second, in this it does the following:

- Processor fetches an instruction from memory
- Decodes the instruction to understand what it is
- Executes the instruction
- Moves onto next instruction.

This continues until the program finishes execution. This is the the basis of a **Von Neumann** model of computing.
Operating Systems is the body of computing that is in charge of managing memory allocation, interaction with devices and more, and all of this should be created so that its in an easy to use manner.
The technique the OS does this is through a process called **virtualization**. The OS creates this illusion of virtualization by creating **virtual machines** that is much bigger than the machine itself, in order to make things easier! Such an example occurs when you run a single program multiple times concurrently on a single processor. What happens is that the system still manages to run all these processes simultaneously through **virtualization of the CPU**. Users can take advantage of these methods through **API**'s that allow interaction with the more privileged instructions through the use of **system calls.** The OS is responsible for managing all these resources and how things are shared and communicated between all the processes running on the machine.

In order to do this there are two concepts to solidify. **Policies** (used to answer questions such as which process should run at a particular instant) and **Mechanisms** (The actual OS implementation of this scheduling).

Physical Memory, also referred to as *memory* is essentially a massive array that is indexed using **addresses**. One can read from memory and can write to memory, through the access of the addresses. Note that each process (or program running) has a unique identifier known as the **PID**. Now when running two processes that do the same thing we notice that though the address may be specified to the same point, the two processes will run completely independently! This brings the question of how does the OS do this? The answer lies in the virtualization of memory as mentioned above. The OS gives each process its own **virtual address space** so that each process is independent, and then maps those addresses to the physical memory (which is indeed a shared resource also managed by the OS).

We then face the issues of **concurrency** as in how does the OS manage to do all this juggling between resources and processes simultaneously? This issue is more intuitive when dealing with **threads** which can be thought as multiple instances of the same method within a single

process. You see, threads share certain resources such as the local variables within memory and so when multiple threads run and they are not **atomic** (all at once) strange things happen as they begin to influence each other. A very common example of such a situation can be demonstrated through a counting examples spread amongst several threads.

The final issue when talking about the OS is the topic of **persistency**. Physical memory implemented as **DRAM** (or **SRAM** for caching) is **volatile** in that the data is lost once power to the system is cut. So how do we make sure the data remains where we want it to be? We use **I/O** devices such as **hard drives** or **Solid State Drives** to store permanent data. The software that manages this is known as a **file system**. Through the file system and the disk, processes can share information! **Device Drivers** allow the OS to communicate with the multiple devices connected to the machine, and the file system of the OS acts as the bridging point between a disk and the data. The OS abstracts away the gritty details of device drivers and low level device specific interfaces through the system calls it provides users such as *write, read* and more!
To handle the problems of system crashes during writes, most file systems incorporate some kind of intricate write protocol, such as journaling or copy-on-write, carefully ordering writes to disk to ensure that if a failure occurs during the write sequence, the system can recover to reasonable state afterwards.

Design Principles:

Now that we realize what the OS does and the issues we must deal with, it is useful to follow some general good practices when designing an OS or any application. We start with an **abstraction** to hide away the unnecessary details when they are not required, **minimizing overhead** such as the extra time or space, by making useful tradeoffs when required as perfection is not always attainable. We must also be wary of **protection** between applications to abide by the security of applications, and this is generally done through a method of **isolation** as seen by independent processes. We then wish to make the OS **reliable**, because it is the center of any modern day electronic machine, its failure can lead to catastrophic results which can be no means be tolerated. This is in no way a trivial task. More important goals are **energy-efficiency, mobility etc.**

Common problems one faces in systems of many fields can be broken into four categories:

- Emergent properties
- Propagation of effects
- Incommensurate scaling
- Trade-offs

Emergent Properties

These are properties that are not evident in the individual components of a system but upon the combination of these components, such properties arise (also commonly known as

SURPRISES). Sometimes it is very difficult to be able to predict these properties through the inspection of these individual components but only evident upon the coalition of it all.

Propagation of Effects

Small local changes to a specific component results in failures in other parts of the system. This is something that should be absolutely avoided. To put it into perspective just think about the impact of changing the size of the tires a vehicle can hold affects the entire design of the vehicle. It means the wheel wells will have to be changed, enlarge extra tire storage which requires rearrangement of the trunk which will make changes to the seat layout which in turn affects the comfort and may reduce safety. Such cascading effects are a possibility in any system design and so each decision must be made with careful thought and reliability.

Incommensurate Scaling

as a system increases in size or speed, not all parts of it follow the same scaling rules, so things stop working. The mathematical description of this problem is that different parts of the system exhibit different orders of growth. A great example would be trying to scale up a mouse to the size of an elephant which would not be feasible since the skeletal structure of the mouse would fail to support its newly sized body and thus bring the whole system that is the mouse crashing down! Obviously this applies to non-biological systems, but can be avoided by accounting for scaling.
Incommensurate scaling shows up in most systems. It is usually the factor that limits the size or speed range that a single system design can handle. On the other hand, one must be cautious with scaling arguments.

Trade-Offs

The general model of a trade-off begins with the observation that there is a limited amount of some form of goodness in the universe, and the design challenge is first to maximize that goodness, second to avoid wasting it, and third to allocate it to the places where it will help the most. One common form of trade-off is sometimes called the *waterbed effect*: pushing down on a problem at one point causes another problem to pop up somewhere else. Another form of trade-off appears in *binary classification* by modelling things into two categories – present and absent, which obviously lacks a direct measure of the property for which we create *proxies* as indirect measures. However, this then misclassifies certain things and so the improvement of one sector may lead to the downfall of the other sector.

Since computer systems are so new, there is no formulaic approach to design but rather the analysis of prior examples and a mixture of thought out creativity leads to some generally decent results.

Talking about the design principles that should go behind systems begs the question as to what a system really is. A system can be defined, for our purposes as **a set of interconnected components that has an expected behavior observed at the interface with its environment.**

The system divides things into what is a part and what is not a part of the discussion. The stuff that is part of the discussion are a part of the system, but the rest is just the *environment*. The way for the system to then interact with its environment comes from the use of an *interface*.

Depending on point of view, we may choose to ignore or consolidate certain system components or interfaces (purpose). The ability to choose granularity means that a component in one context may be an entire system in another. In summary, then, to analyze a system one must establish a point of view to determine which things to consider as components, what the granularity of those components should be, where the boundary of the system lies, and which interfaces between the system and its environment are of interest.

When it comes to designing systems, one must take into account the complexity of said system. But due to the lack of a well defined measure of complexity, we instead delegate the definition of complexity to the presence of certain signs, and if enough of these signs appear, then the system is complex. Five signs of complexity can be stated as:

1.) Large number of components
2.) Large number of interconnections (This can lead to high sensitivity to disturbance which can make systems harder to rigidify)
3.) Many irregularities (Exceptions complicate understanding)
4.) A long description (The length of the shortest specification of a system {Kolmogorov complexity} or the lack thereof can make systems complex in their understanding)
5.) Team of designers, implementers or maintainers (If a system is not simple enough for one person to understand all of it, then the system is complex as it requires collaboration and coordination across a team)

Of course complexity is directly dependent on the topic of abstraction as mentioned earlier, because the deeper we dig the more complex we can get. However, for the sake of this course we stick to the definition of complexity as its persistence regardless of abstraction.

Though these are signs of complexity, where doe the sources of complexity emerge from? There are many sources of complexity, but two merit special mention. The first is in the number of requirements that the designer expects a system to meet. The second is one particular requirement: maintaining high utilization.

1.) **Cascading and interacting requirements:** As the number of requirements increases for a machine, the subjective complexity of the system increases exponentially. This is intuitive because we see that as we add on requirements the number of interactions between each component that handles each requirement also increases which in turn adds to the complexity of the system. This can be referred to as the **Principle of escalating complexity**. A common tackle to such a problem is to make a system *general* enough to satisfy all these requirements. However, we see that this generality also adds to the complexity of a system and so it comes with a trade-off. We must gauge how much this generality is actually required and move form there. Users of a system with excessive generality will adopt styles of usage that simplify and suppress generality that they do not need which thus complicates the sharing of the system. The principle to keep in mind here is **Avoid excessive generality**. As we see, artful design comes in the ability to generalize enough to minimize the number of exceptions that are to be handled as special cases. Another measure of complexity comes in battling the effects of incommensurate scaling, as the tools required to solve a larger problem become more and more complex. **Requirements change** and combined with the effects of scaling, this can provide unforeseen complications. One must update or apply *patches* to satisfy these changes, but as a system scales, the changes introduced may have negative impacts on other components of the new system and the knowledge required to maintain the system becomes larger in its base and older. A common phenomenon in older systems is that the number of bugs introduced by a bug fix release may exceed the number of bugs fixed by that release. The bottom line is that as systems age, they tend to accumulate changes that make them more complex. The lifetime of a system is usually limited by the complexity that accumulates as it evolves farther and farther from its original design.

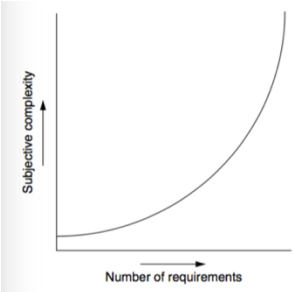


FIGURE 1.1
The principle of escalating complexity.

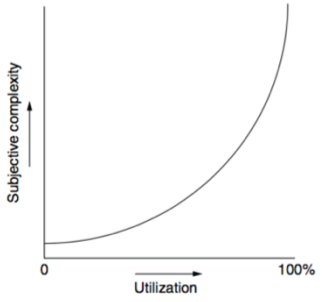


FIGURE 1.2
An example of diminishing returns: complexity grows with increasing utilization.

2.) **Maintaining High Utilization:** Whenever a scarce resource is involved, an effort arises to keep its utilization high. When striving for higher utilization we encounter the design principle referred to as **The Law of Diminishing Returns** – The more one improves some measure of goodness, the more effort the next improvement will require. The more completely one tries to use a scarce resource, the greater the complexity of the strategies for use, allocation and distribution. The more one tries to increase utilization of a limited resource, the greater the complexity (exponential).

So how do we deal with these arising complexities, now that we know their potential sources and the methods to identify them. Four general categories govern the techniques to cope with complexities:

1.) **Modularity:** The simplest, most important tool for reducing complexity is the divide-and-conquer technique: analyze or design the system as a collection of interacting subsystems, called modules. The power of this technique lies primarily in being able to consider interactions among the components within a module without simultaneously thinking about the components that are inside other modules. As an example, we may notice that debugging a non modularized system with N bugs will take N^2 time, but debugging the same program modularized by into K modules will take N^2/K time. The design principle that is of importance here is the **Unyielding foundations rule –** It is easier to change a module than to change the modularity, and thus it is of utmost importance to get the modularity correct to begin with!
2.) **Abstraction:** To reduce the drawbacks of propagation of effects between the modules, the design has to be such that the number of interactions and reliability of modules on each other is as low as possible. Such is possible through the idea of abstraction. It is the separation of interface from internals of specification from implementation. The goal of minimizing interconnections among modules may be defeated if unintentional or accidental interconnections occur as a result of implementation errors or even well-meaning design attempts to sneak past modular boundaries in order to improve performance or meet some other requirement. Now closely related to abstraction is another important design principle known as **the Robustness Principle –** be tolerant of inputs and strict on outputs. That is for a module to handle any case of an input – well defined or not – and still produce meaningful outputs. This principle forms the basis for digital logic, but at the same a tension between exists between this principle and another popular design principle known as **the Safety Margin Principle –** Keep track of the distance to the cliff, or you may fall over the edge. Though the module should recognize all forms of input it is important to realize that if an input is awry, then catching it sooner than later will stand in good stead din prevention of worse errors in the future. This is implemented in two modes:
    a. Shake out – module checks every input, and if it doesn't meet the specification then immediately throw an error to catch it before it does further damage.
    b. Production – modules accept any input that is reasonably interpreted as per the robustness principle.

    Carefully designed systems blend the two.
3.) **Layering:** By classifying sets of modules into layers, we reduce the number of interconnections between modules, and restrict them to only form meaningful connections between each other in the given layer.
4.) **Hierarchy:** The final major technique for coping with complexity also reduces interconnections among modules but in a different, specialized way. Start with a small group of modules, and assemble them into a stable, self-contained subsystem that has a well-defined interface. Next, assemble a small group of subsystems to produce a larger subsystem. This process continues until the final system has been constructed from a small number of relatively large subsystems. The result is a tree-like structure known as a hierarchy. The advantage of a hierarchy is that the module designer can

focus just on interactions with the interfaces of other members of its immediate subsystem.

We then finally wish to put all these pieces together through the concept of *naming*. That is one module will name another module it intends to use. This allows for postponing of decisions, easy replacement of one module with a better one, and sharing of modules. In a modular system, one can usually find several ways to combine modules to implement a desired feature. The designer must at some point choose a specific implementation from among many that are available. Making this choice is called *binding*. Recalling that the power of modularity comes from the ability to replace an implementation with a better one, the designer usually tries to maintain maximum flexibility by delaying binding until the last possible instant, perhaps even until the first instant that the feature is actually needed. Using a name to delay or allow changing of a binding without the implementation of the named module is referred to as *indirection*. This forms the basis of another design principle **Decouple modules with indirection –** This supports replaceability.

While these design principles and ideologies pertain to all systems, we must make the distinction that *computer systems are different in two ways:*

1.) The complexity of a computer system is not limited to physical laws.
2.) The rate of change of computer system technology is unprecedented.

These make a huge difference to the complexity!

1. No Nearby Bounds on Composition

Any analog based system has the added complexity of peripheral noise distorting the valuable signals and data, that accumulates with each added component. It provides a limit on the number of analog components that a designer can usefully compose.

However, digital systems – as are computers – are completely noise free and thus don't have this added complexity. The designers thus use a version of the **robustness principle** known as the **static discipline** - requires that the range of analog values that a device accepts as meaning the digital value ONE (or ZERO) be wider than the range of analog values that the device puts out when it means digital ONE (or ZERO). This discipline is an example of being tolerant of inputs and strict on outputs.

It is worth noting, that are the fundamental levels, digital systems are made up of *non-linear* analog components that have *gain* between the inputs and outputs for wider tolerance of values and ensuring that the outputs stay within narrow specifications. Gain and non-linearity make up the *property of digital systems* known as **level restoration** or **regeneration.** They create clean interfaces for confident interconnections between subsystems.

*Note: The static discipline and regeneration do *not ensure no mistakes.* If the input signal is far enough out of the tolerance bounds, then it could output the wrong bit! These mistakes are big ones and not small like those caused by noise in the analog counterparts.

While it may seem the lack of physical laws comes as a pro to the complexity levels of a computer system, this is not actually the case. Because there are no physical laws (other than memory size and transistor sizes), the complexity of the system can become unprecedented. For example, the number of components in a digital chip may be in the billions while that on an analog chip would be significantly lower. The contribution from software to the complexity is even worse than the static discipline, because the composition of software can go on as fast as people can create it (which is why we have millions of lines of code for some applications). Abstraction also adds some issues with digital complexity because in reality these *abstractions are leaky* as they *don't perfectly conceal the underlying implementations*. This leakiness accumulates as the number of software modules increase – similar to noise in analog systems!

As a result, systems with complexities far beyond the understanding of its designers are created, and as such leads to the misuse of the tools of modularity, abstraction, layering and hierarchy which just adds to the complexity. It is up to the designer to self-impose limits, but this is hard to do as it is hard to say no to any reasonable sounding feature -> MORE COMPLEXITY!

2. Rate of Change of Tech is Unprecedented

As in accordance with *Moore's Law* and the rapid advances made in digital components, the speed at which computer technology changes is unprecedented. Complex systems take years to build, but when it comes to computer systems, these years may lead to significant changes in the tech used in the original design which may thus lead to the need of a complete design overhaul to include these newer and better components which again adds to the complexity of the design. The general rule of *incommensurate scaling* then applies as the Incommensurate Scaling Rule – Changing any system parameter by a factor of *10* usually requires a new design.

This ever fast changing pace of computer technology means that ideas that weren't possible a couple of years ago are now mainstream, which thus requires the need for design changes. Another issue this lends itself to is that since things are moving so quickly, we don't have time to focus and iron out most of the mistakes and errors made in the previous systems upon which the new ones are built which results in the propagation of errors -> increased complexity.

Rapidly improving technology means that brute-force solutions (buy more memory, wait for a faster processor, use a simpler algorithm) are often the right approach in computer systems, whereas in other systems they may be unthinkable.

Usability – that is the human engineering aspect of computer design – becomes another issue as a result of this growth. Previous features that may have been to expensive to

implement/impossible to implement for a user can suddenly be implemented and thus must be integrated into the design which is a complex process. Also because of globalization and high networking speeds, can rapidly change the workload on existing systems overnight.

Thus, the study of computer systems involves telescoping of the usual processes of planning, examining requirements, tailoring details, and integrating with users and society. This telescoping leads to the delivery of systems that have rough edges and without the benefit of the cleverest thought.

Wednesday:

Additional coping

With these added complexities specific to computer systems, computer designers need more tools to help with these difficulties. It is often hard to generalize a perfect design principle/tool among the see of alternatives, and this is something that comes with experience. However, there are some principles (some shared by other disciplines too) that come in handy when facing the beasts of computer complexity:

1.) Iteration

Start by building a simple, working system that meets a small subset of requirements and then build from there (aka incremental build). These small steps reduce the risk of complexity as the system gets bigger and bigger. This way making adjustments for the new technology that is released is easier, and that is why a lot of computer systems, have constantly updating versions. *Successful iteration requires considerable foresight.*

a) Design for Iteration – You won't get it right the first time, so make it easy to change. This requires the process of *documentation*. Make sure to *take small steps, don't rush, plan for feedback, study failures.*

b) Keep digging – Complex systems fail for complex reasons. Sometimes new releases can sprout bugs that never existed because the case never arose. Understand why these bugs come about and then fixing them and maintaining the system for the future becomes easier. *Don't ignore unexplained behavior.*

With iteration the risk of losing conceptual integrity arises. There must always be someone on guard to make sure the overall design rationale remains clear despite the changes made during iteration. Bad-news diode can prevent the realization that changing a different part of the system is more appropriate than a quick hacky fix. This can lead to reluctance to change the modularity (*unyielding foundations rule states this is harder than changing modules*) by the original designers, as it leads one to believe that all the effort put in was a waste.

Success of small initial designs can lead to *overconfidence* and *over ambitiousness* in a later iteration -> disastrous overreaching and consequent failure – **Second-System effect**.

In summary, iteration can be thought of as applying modularity to the management of the system design and implementation process.

2.) **Keep it Simple, Stupid (KISS)**

Since a designer has to self-impose the limits on a computer system, he/she should choose abide by the ideology of *simplicity*. This is hard because the case-study of previous systems can make designers want to create grander features, more complex features that take advantage of new technology, older exceptions and complications seem easier, fear that a competitor has even more features, and arrogance, pride and overconfidence seem to over rule simplicity. This makes it hard to say **no to a requirement. Adopt sweeping simplifications** – So you can see what you are doing. They will turn out to be one of our best hopes for keeping control of complexity.

Libraries:

Libraries are chunks of code that are imported into your own code to help simplify tasks, by using pre built functionality. There are three main kinds of libraries to be wary of and their differences lie in the time and manner in which they are combined to the original source code.

1.) **Static Libraries:**

Collection of ordinary object files, ending with the .a suffix. They are linked during the compilation process and the entire code base is generally added to the final executable.
They are less used nowadays.

Advantages	Disadvantages
Saves the need for recompilation (less important nowadays since compilers are pretty fast).	The final executable is much larger.
(For Vendor) Hides the library source code.	(For User) Hides the library source code.
The static ELF Library is technically faster when linked into the executable than shared or dynamically loaded library, but negligible nowadays.	

Creation: ar rcs my_library.a file1.o file2.o
Usage: Use the –l (linker option) to specify the library. Place this after the name

of the file to be compiled.

2.) Shared Libraries (Dynamically Linked Libraries):

These are libraries loaded by the program when they start. When a shared library is installed properly, all programs that start afterwards automatically use the new shared library. In Linux, you can update libraries but still support older libraries, override specific functions/libraries and do all this while programs are running using existing libraries. There is a naming convention with shared libraries where we add the prefix *lib* the suffix *.so* followed by a version number (this is avoided in the actual linked library). The command *ldconfig* handles some of this for you. These shared libraries are stored in */lib* folder in the filesystem (*/local/lib* for startup libraries and */lib* otherwise). Because the code is linked at runtime, the system generally caches the required libraries (from the ELF headers) and fetches the code using the paths specified in this cache file. Environment variables can be used to control/override the processes involved in the linking.

Advantages	Disadvantages
Code sharing between libraries, less duplicate code.	DLL Hell – shared library dependencies can cause a nightmare. Refer to CS33 Notes for more details.
Automatic updates, avoids re-linking	Aliasing, can result of same data being shared which can cause unexpected errors.
Splitting program up into several linkage units can protect you from crackers.	

Creation: Compile with the –fPIC option to make the code position independent, and use the –shared option to make the library .so. You may also need to specify path variables and so on.

3.) Dynamically Loaded Libraries (DL)

Technically these libraries are still normal object files that are created as shared libraries, but the way they are integrated into the final executable is noticeably different. Using the *dlopen(), dlsym(), dlclose()* API we see that the libraries are only loaded into the program when the program needs it. That is the libraries are not loaded during link time or start up time, but rather while the program is running and the specific command is found that requires the library. They are generally used for *software plugins* (sometimes have *.dso* suffix). Loading a library causes memory to be allocated; the library must be deallocated in order to avoid a memory leak. Additionally, failure to unload a library can prevent filesystem operations on the file which contains the library.

X86 Calling Conventions:

Calling conventions describe the interface of called code:

- The order in which atomic (scalar) parameters, or individual parts of a complex parameter, are allocated
- How parameters are passed (pushed on the stack, placed in registers, or a mix of both)
- Which registers the callee must preserve for the caller
- How the task of preparing the stack for, and restoring after, a function call is divided between the caller and the callee

Caller Clean Up:

In cdecl (C declaration), *subroutine arguments* are passed on the *stack*. *Integer values and memory addresses are returned in the EAX register*, *floating point values in the ST0 x87 register*. Registers *EAX, ECX, and EDX are caller-saved*, and the *rest are callee-saved*. The x87 floating point registers ST0 to ST7 must be empty (popped or freed) when calling a new function, and ST1 to ST7 must be empty on exiting a function. ST0 must also be empty when not used for returning a value.
The caller cleans the stack after the function call returns.

Callee Clean Up:

In these conventions, the callee cleans up the arguments from the stack. Functions which utilize these conventions are easy to recognize in ASM code because they will unwind the stack prior to returning. The x86 *ret* instruction allows an optional *16-bit parameter that specifies the number of stack bytes to unwind before returning to the caller.*

Either Caller or Callee Clean Up (thiscall):

For the GCC compiler, thiscall is almost identical to cdecl: The caller cleans the stack, and the parameters are passed in right-to-left order. The difference is the addition of the *this* pointer, which is pushed onto the stack last, as if it were the first parameter in the function prototype.

On the Microsoft Visual C++ compiler, the *this* pointer is passed in ECX and it is the callee that cleans the stack, mirroring the stdcall convention used in C for this compiler and in Windows API functions. When functions use a variable number of arguments, it is the caller that cleans the stack (cf. cdecl).

Another part of a calling convention is which registers are guaranteed to retain their values after a subroutine call. According to the Intel ABI to which the vast majority of compilers conform, the EAX, EDX, and ECX are to be free for use within a procedure or function, and need not be preserved.

Note that another technique known as *name mangling* is used to resolve register renaming (and more). The combination of this, the calling conventions and type representations is the Application Binary Interface (ABI).

Application Binary Interface (ABI):

This is the interface between two program modules but at the level of *machine code*. This is the true difference between ABI's and *API*'s.
An ABI determines such details as how functions are called and in which binary format information should be passed from one program component to the next, or to the operating system in the case of a system call.
It is usually the job of the compiler, OS or library writers to adhere to ABI's.
ABIs cover details such as:

- the sizes, layout, and alignment of data types
- the calling convention, which controls how functions' arguments are passed and return values retrieved; for example, whether all parameters are passed on the stack or some are passed in registers, which registers are used for which function parameters, and whether the first function parameter passed on the stack is pushed first or last onto the stack
- how an application should make system calls to the operating system and, if the ABI specifies direct system calls rather than procedure calls to system call stubs, the system call numbers
- and in the case of a complete operating system ABI, the binary format of object files, program libraries and so on.

Application Programming Interface (API):

An API expresses a software component in terms of its operations, inputs, outputs, and underlying types, defining functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface. A good API makes it easier to develop a program by providing all the building blocks, which are then put together by the application programmer.
APIs often come in the form of a library that includes specifications for routines, data structures, object classes, and variables.
An API specification can take many forms, including an International Standard, such as POSIX, vendor documentation, such as the Microsoft Windows API, or the libraries of a programming language, e.g. the Standard Template Library in C++ or the Java APIs.

To design good API's the design techniques described before should be employed.

SOME NOTES FROM CLASS:

The ABI is the binding between the API and the computers ISA.

When it comes to linking we have to focus on two specific times. **Linkage time** and the **execution time**. So when a program is being compiled there comes the point where data that is imported from libraries must be linked into the overall final executable so the code is cohesive. There are multiple ways in which this happens as discussed above, with **static libraries**, **shared libraries** and the process of **dynamic linking**. Note that the terms dynamic loading and dynamic linking are often miscommunicated in the world of computer science, so the definition provided here is the perspective of Professor Mark Kampe and one that is shared by my own. With *static libraries* what happens is that before creating the object modules, the code that uses library functions is resolved into external functions that tell the compiler these functions don't exist within the source code but must be fetched from somewhere. With static libraries, during the *linkage time* all the code is resolved and all these *extern* functions are grabbed and put into the source code so that the final executable contains everything. But when resolving the externals, a lot of unnecessary code is duplicated and linked into the final executable, and as a result the program becomes heavier in size. It does however make the code more portable but more prone to errors with library updates as everything needs to be recompiled and re-shipped out! Come in the action from *shared libraries*. These bad boys basically during linkage time resolve the externals to an address in something called the **Global Offset Table**. This table contains all the functions from the library at different indices. Then during run time, as in when a function that has a stub is hit in the instructions, the program will see a *jmp* to an address within this GOT and will grab the function from there! The difference between *dynamic linking* and *dynamic loading* can be thought of as linking doing everything implicitly for you, while loading requires the use of *dlopen()* and other functions. Dynamic linking will in its linkage time find the necessary libraries you will need and call dlopen and close when necessary, thus abstracting hassle away from you. What a wonderful thing to have. Note that all the information about the program, like its text, data, .bss, and symbols and references into the GOT are store in the *load module*. Modern UNIX based systems use the **Executable and Linkable Format** to do so.

SOME NOTES FROM DISCUSSION:

Note that when a signal is received, an exception should be thrown, and this is what the system does, but you can take control of it all by defining your own *handler*. When you do this the system assumes you are going to take care of the situation. There are multiple ways you can go about doing this, and the most common is to print out a thread safe message (perror) to standard error and then exiting, but remember if you wish to do fix it/forget to exit you must increment the %rip to the very next instruction else you will remain in an infinite loop that the system will keep assuming you are fixing things in the handler. You can't just simply use *goto* statements unless you save the *context* of the stack and then go back to it when you hit the signal. This can be done through functions in the <ucontext.h> library.

WEEK 2

Monday:

OS Part I – Virtualization!

To begin our discussion about virtualization we need to first understand the concepts behind a **process**. A process is simply a program that is being run by the OS. Now you can imagine that while you are using your computer, there are several processes such as web browser, mail, music player, terminal etc. are all processes running. Now modern computers come loaded only with one or a few processors (CPU), but we could be running hundreds of processes. So how do they all run? The way this works is through virtualization of physical memory by making each process think it has its own CPU. The main method this works is through **time sharing** where the OS uses methods to determine when each process can use the CPU.
We call the low-level machinery **mechanisms**; mechanisms are low-level methods or protocols that implement a needed piece of functionality. They can be thought of as answering the *how* questions, while there is a form of intelligence the OS requires to perform this mechanisms, called **policies** that answers the *which* question.

The OS performs **context switches** between the different processes. In terms of implementing the time sharing efficiently the OS uses a **scheduling policy** to perform these context switches. Each process can be thought of as having a **machine state** or its **context** which are the different states the process can be in. Along with its states the process has an API that includes the following basic methodologies:

1.) **Creation** – Each process needs to be created, and thus needs some way of letting the OS know to create it.
2.) **Destruction** – Though a process can simply exit, there are situations wherein the process could be given user input that instantiates an exit. Thus it requires some destruction method.
3.) **Wait** – Sometimes the process needs to wait for something to be completed.
4.) **Miscellaneous Control** – Provide different controls for the process to do different things based on these controls, other than killing and waiting.
5.) **Status** – Give information about how the process is doing, how long its been running for etc.

Most programs exist as some **executable format** in the disk or SSD, when the process needs to be created all the code and data of the program is loaded into memory by the OS. Initially this used to be done **eagerly** – everything was loaded in at once – but this was not as efficient, and so the process is now done **lazily** – loading pieces of code only when needed. This uses the process of **paging** and **memory swapping** but that'll come later. After this memory is allocated on the **run-time stack** and all the code is put on here. Any data structure or data the program

needs while running is allocated through the malloc and free API on the **heap**. The OS also performs initialization tasks with I/O through **file descriptors** but this is more of a topic of **persistence**. After all this has been setup, the OS will move the **instruction pointer** to the main() part of the program.

When talking about process states, we focus on three different states:

1.) **Running –** The process is running on a processor by executing instructions.
2.) **Ready –** The process is ready to resume/start running, but is waiting on the OS scheduler to give it the go ahead.
3.) **Blocked –** The process has performed some kind of event that requires it to wait for period of time, which is when the OS will block the process so that other processes can run at this moment. An example is I/O operations that wait on the devices and user input, which is when the OS can make other processes use the resources.

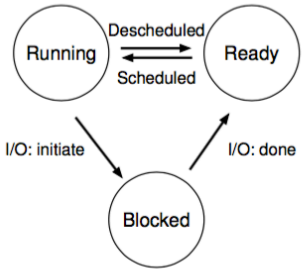


Figure 4.2: **Process: State Transitions**

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Figure 4.4: **Tracing Process State: CPU and I/O**

Figure 4.4 shows us how the processes going through different states are handled, and how their resources are shared through time-sharing. This is all done by the OS's **scheduler**.

When talking about all the states of the processes, we can imagine that the OS would need a list of all the states to go through so it can perform the correct policies. Thus one of the most basic data structures in an OS is called the **process list**. The figure on the right shows the basic idea of a process list. The registers are used to save the **context** of the process before the OS does a context switch, and then when the process resumes, the state of the process is back to how we left it.

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Figure 4.5: **The xv6 Proc Structure**

Each process has a parent process that runs it. The exit statuses of its child processes are important as they can inform the parent on how to proceed next. For example, the child process may be killed but go into a zombie state, that is, it waits until its data is cleared up by the OS.

Everything discussed until now is just a high level overview of how processes work and is a small stepping stone into delving into the nitty-gritties of virtualization.

In terms of writing actual code to spawn processes and control it. There are three main functions to be aware of – *fork(), exec(), wait()*

Fork()

This system call is used to make a new process. Each process has a unique identifier known as the **PID**. The call to fork() will spawn a *child process*, which will begin execution from the point fork was called. That is the child process is an exact copy of the parent process but it has a different PID, and the child process will start running its instructions from the line that called fork. If it is a child process, it will have the integer value **0** indicating so. Note that when you fork a process, the results as to whether the child or parent will first act is undetermined – it is **not deterministic**. Because the scheduler is complex, you can't make general assumptions about which process will execute first either. This can lead to issues involving **concurrency** in **multithreaded programs**.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <sys/wait.h>
5
6   int
7   main(int argc, char *argv[])
8   {
9       printf("hello world (pid:%d)\n", (int) getpid());
10      int rc = fork();
11      if (rc < 0) {          // fork failed; exit
12          fprintf(stderr, "fork failed\n");
13          exit(1);
14      } else if (rc == 0) { // child (new process)
15          printf("hello, I am child (pid:%d)\n", (int) getpid());
16      } else {               // parent goes down this path (main)
17          int wc = wait(NULL);
18          printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19                 rc, wc, (int) getpid());
20      }
21      return 0;
22  }
```

Figure 5.2: **Calling `fork()` And `wait()` (p2.c)**

Wait()

The way to get around this undetermined behavior is by using the *wait()* [or the *waitpid()*] system call. When you make the parent of child wait, the OS scheduler is going to immediately block the process and context switch to the process that needs not wait, and the process that waits will not begin execution until the other process is finished. This way you can make deterministic switches and programs to run even with the fork command.

Exec()

This can be useful for when you want to run a program that is different from the calling program. Fork() is only useful if you want to keep running copies of the same program. We generally use *execvp()* where you feed it the name of the program you want to run along with its arguments, and it will load the code from that executable -> overwrite the current code segment -> heap and stack and other parts of memory space are re-initialized -> OS runs this program and passing in whatever arguments from the second parameter of *execvp()* -> reverts back to the saved context. In essence what this means is that exec() isn't actually creating a new process but rather transforms the current process into the one we want.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {              // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

Figure 5.4: **All Of The Above With Redirection (p4.c)**

These commands allow us to essentially create shells that can do so much. For example, **redirection** then becomes extremely straight forward, wherein we simply change the file descriptor in our program and then when we call a new program within our main process, it will act on that file descriptor!

This is similar to how **piping** works, but rather using the **pipe()** system call that maintains an *in-kernel queue*.

There exist other parts of the Process API that are also interesting such as the *kill()* function that can use signals to manipulate other processes.

Now that we are aware of processes and their relative uses and creation, we must delve into how to efficiently virtualize the CPU. That is, we focus on *performance* – how do implement it without excessive overhead to the system? – and *control* – run the process efficiently while still retaining control over the CPU.

The basic technique behind this is to use something called **Limited Direct Execution (LDE).** The "direct execution" just defines how the OS prepares the program for running and how it exits and deals with the program when it finishes. This is shown on the right hand side.

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

However, this doesn't place any restriction on the program, which can hog up all the CPU for as long as it wants, and thus gives the OS no way of scheduling tasks and switching between them to actually implement the virtualization of the CPU! Herein we face two problems.

1.) **Restricted Operations:**

We need to come up with some system to allow for respective control, and so we have two modes of operation. The **user mode** is the mode the process actually runs in and it is limited to what it can do. Any interaction with I/O devices or hardware for example, is not permitted in user mode. The other mode is the **kernel mode** wherein; the OS has full control of the system. The operations that cannot be run in user mode but can be done so in kernel mode is referred to as **privileged instructions**. So how exactly can a program access this kernel mode of operation? The answer is **system calls** which can be thought of as a path from user mode to kernel mode. To do this though, the program issues something known as a **trap** instruction, which simultaneously jumps into the kernel and raises the privilege level. Once in this new mode, the system performs the operation specified and then the OS calls a special **return-from-trap** instruction to fall down into the user mode and switch back the privilege level. However, there has to be a way of *saving* the context of the program before moving into a higher level. It does this by saving things on a per process **kernel stack**, and the return-from–trap will pop these values off the stack and resume execution. But the other issue then is how does the calling process know where to jump to? The kernel needs to carefully control what code it executes in the trap. This is done through the initialization of a **trap table** at boot time. The machine boots into kernel mode and configures the hardware as it needs to, by telling it what code to run when certain interrupts occur. The OS informs the hardware of the location of **trap handlers** through special instructions. Once the hardware knows it remembers these locations.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of... syscall handler	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs from kernel stack	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call **trap** into OS
	save regs to kernel stack	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs from kernel stack	
	move to user mode	
	jump to PC after trap	
		...
		return from main **trap** (via `exit()`)
Free memory of process		
Remove from process list		

Figure 6.2: **Limited Direct Execution Protocol**

2.) **Switching Between Processes**

There still exists the question of how to switch between processes. The decision is made by the scheduler and some complex algorithms that we will cover a little later, but we will focus on how the switching is actually done for now. The issue arises because, when a program is running, it has access to the CPU but the OS doesn't so how does the switch actually happens? There are two approaches to solving this issue:

a.) **Cooperative Approach: Wait for System Calls**

The OS trusts the process to behave reasonably, and the process transfers control of the CPU to the OS through system calls. They also include an explicit **yield()** system call which simply transfers control. But basically any system call will transfer the control. The applications on a system that follows this approach will also transfer the control when an illegal operation is done, such as dividing by zero wherein a **trap** is generated which gives control of the CPU back to the OS. But for example, if the process gets stuck in an infinite loop or simply never makes a system call this can mean the OS never regains control of the CPU, and you would have to **reboot** the system!

b.) **Non-Cooperative Approach: The OS Takes Control**

The system has a **timer interrupt** implemented. It can be programmed to raise a trap/exception every now and then, and a pre-configured **interrupt handler** will run to handle this event.

Obviously as mentioned before this handler is loaded into the trap table and the hardware is made aware of it at the booting process. In addition to the flow from before, the timer is also started after the booting is finished. The hardware and software have an added responsibility to save the state of a process after its time is finished, so that it can switch between processes without loss of information. In the process of making a **context switch** the OS will save the general purpose registers, program counter, kernel stack pointer of current running process and then restore the registers, PC and switch to the kernel stack for the soon-to-be-executing process. This is all run through a call to **switch()** routine that carefully saves the state in software and loads the respective process as determined by the scheduler. The diagram on the right gives a better sense of this.

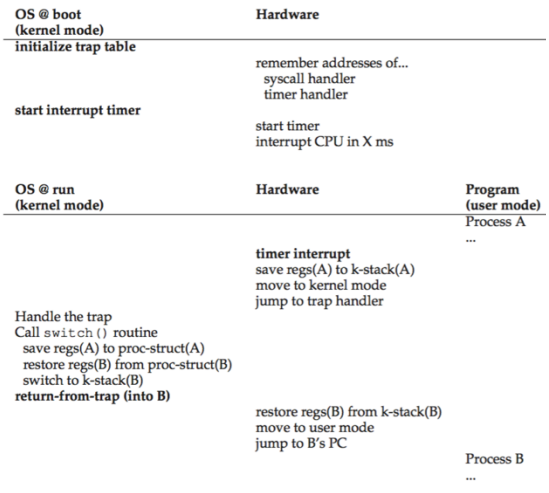


Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

This – along with the scheduler - then partially solves the issue of efficient virtualization, but still doesn't deal with problems involving **concurrency** – for example what happens when the system receives two interrupts at the same time?! One approach is to **disable interrupts** so it ensures that when one interrupt is being handled, nothing else will mess with the process. However, then the OS has to be cautious about not disabling interrupts for too long. Another method of handling concurrency involves **locking mechanisms** that protect concurrent access to internal data structures.

Thus collectively this set of techniques is referred to a s **LDE** and is essentially **baby-proofing** the CPU by setting trap handlers and starting an interrupt timer and running in restricted mode.

SOME NOTES FROM CLASS:

A process invokes or initiates a program. It is an instance of a program that can be multiple and running the same application.
When we talk about zombie processes, technically everything of the child process is cleared so it has no resources, but it still has an entry point in the process table for the parent to reference. After the parent gets the relevant information the zombie is cleared off and the process table entry is cleared. However sometimes the parent could ignore its children and so it could end up having a lot of zombie children which will hog up the resources and fork will fail.
The parent reads the zombie child's exit status using the wait command. The zombie's entry is then removed from the process table and the zombie is reaped.
Kill has no effect on zombie processes.
If the parent dies before the child, then the init process will take up all the orphaned children, and is great at handling the statuses of the children.
Zombie processes should not be confused with orphan processes: an orphan process is a process that is still executing, but whose parent has died. These do not remain as zombie processes; instead, (like all orphaned processes) they are adopted by init (process ID 1), which waits on its children. The result is that a process that is both a zombie and an orphan will be reaped automatically.
Is signal enabled or disabled. What is its handler?
In exec, one of the first things the new program does, is it registers its signal handlers.
Kill is the default action, that is all signals get defaulted to KILL and child gets set to Ignore.
Exec gets rid of all the handlers that you defined, and defaults to kill.

Resident state – is your process running or not. All this information is stored on the Resident Process Table. Based on the stuff in this data structure you can do a lot of things like sending signals and getting exit statuses back.
Non-resident state is in memory if you are ready to run or running, but swap you out to disk when you aren't running.

The scheduler will run through the resident process table and select the process -> finds its non resident state -> swaps it out through the switch code.

Creating a new process works as follows:

Allocate/initialize resident process description
Allocate/initialize non-resident description
Duplicate parent resource references
Create a virtual address space

- Allocate memory for code, data, stack
- Load/copy program code and data
- Copy/initialize a stack segment
- Set up initial registers (PC, PS, SP)

Return from supervisor mode into new process

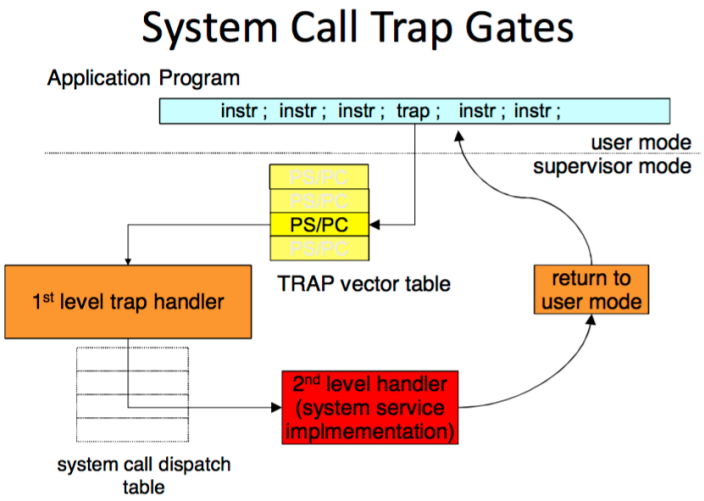
PS is the process status word that contains information about privilege, registers and address space, condition codes, interrupt enables (what interrupts are enabled).

Just before the trap, the current PC and stack pointer is pushed onto the stack of the new mode, so that when we want to return-from-trap we simply pop that block off to get back to where we left off from the asynchronous trap.

The 1st level handler gets a bunch of values from the user mode. It saves the user mode registers.

Decide who should handle this trap.

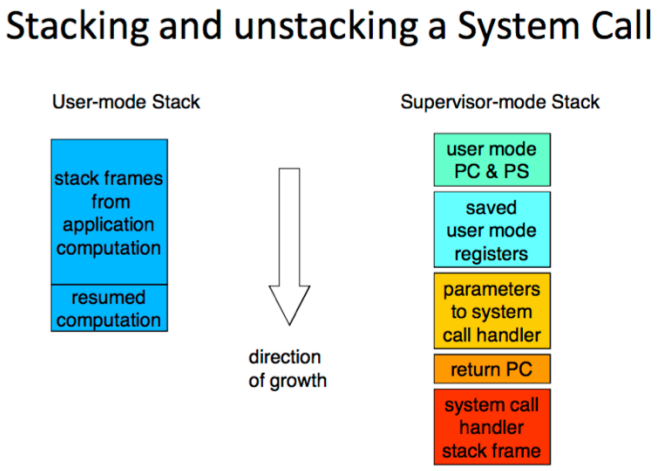
1st level handlers are written in Assembly while the 2nd level can be written in something like C.



Note that the 2nd level handler once returning goes back to the 1st level trap handler which then restores back to user mode.

Asynchronous Exception – I did something bad that caused the trap.

Asynchronous Event – Something that happened that I want to handle.



Interrupts wait in a buffer, traps don't.

Completion call-backs – CPU waits until it is ready to deal with the interrupt.

So interrupts are handled almost exactly the same as traps (but they have to wait). Traps are caused by the **CPU** while interrupts are generated by the **I/O Bus.**

Now we have something similar in the software world as well. These are called **signals**.

Processes can control the handlers for these signals, but it is delivered to the user process mode. But you go through the operating system to create the signal and to go through the signal handler.

Now there are two types of kill signals. Catchable – you can register a signal handler for this, and this is always sent to all processes to begin with. Uncatchable will just kill you.

Traps are raised from an exception, interrupts are asserted.

Exception is raised, corresponding trap from trap table is raised. If the user has not specified a signal handler, the hardware will call the corresponding trap handler, else the handling of the exception is given to the user, and the signal handler defined by user is supposed to correctly handle the exception.

Non-resident state can be thought of encompassing the other states. NR In contains, *running, ready, blocked in* while NR Out contains *blocked out, runnable* states.

Wednesday:

Basics of The Scheduler:

After understanding the mechanisms of processes, we need to discuss the policies behind them that makes them run. More specifically, we need to understand the **scheduling policies** involved in making processes run, so effectively. We can define the **workload** as the information about the processes running on the system. We start by making the following assumptions to reach to a final solution that is the **fully-operational scheduling discipline:**

1.) Each job runs for the same amount of time.
2.) All jobs arrive at the same time.
3.) Once started, each job runs to completion.
4.) All jobs only use the CPU.
5.) The run-time of each job is known (**worst assumption**).

Let us define **2 scheduling metrics** that we base our policy around. **Turnaround time = Time to completion – time of arrival**; **Response time = Time of first run – time of arrival.** Turnaround time is a *performance metric*. There exist other metrics such as *fairness*. Let us start by focusing on turnaround time:

1.) **First In – First Out (FIFO):** The first job to come into the queue of jobs, is the first to run to completion and so on and so forth. **Convoy Formation** comes about from this.

Remove ASSUMPTION 1 -> Jobs run for different times. The above policy will then seriously reduce turnaround time if the very first job is a long one (Think about standing in a grocery line behind someone who has a lot of shit. Its annoying).

2.) **Shortest Job First (SJC):** Based on the jobs that arrive, complete the shortest ones first.

This now solves the problem discussed earlier. **Remove ASSUMPTION 2 ->** Jobs arrive sporadically. Now based on this we see that the scheduler won't know which job is shorter, and so will essentially emulate the problems initially associated with with FIFO.

Remove ASSUMPTION 3 -> Jobs don't run to completion upon starting. With this we recall the **timer interrupt** and **context switching**. Now the scheduler can *preempt* a job.

Note: Preemptive means that the scheduler can stop processes at any point. It doesn't wait till the completion.

3.) **Shortest Time-to-Completion First (STCF):** That is once a job is started, if new jobs that are shorter arrive, block the longer job and prioritize the shorter jobs, and then finish the longer blocked job.

All these policies favored *turnaround time* but never took into account the *response time* which is highly important in time-sharing as users expect interactive performance.

4.) **Round Robing/Time-Slicing:** Based on a certain *time-slice/scheduling quantum* run each job for intermittently for that time slice. But deciding the time slice is important because you don't want to make so short that the time for context switching is longer! You have to *amortize* the cost of switching without making it so long that the system is no longer responsive. [Run: A->B->C->A->C->B->A->…]

Because this is a *fair* policy it will perform horribly on turnaround time since it could take a really long time for these programs to reach completion if they are constantly switched out. Thus a good policy has to come to a decent **trade off** between these two metrics.

Remove ASSUMPTION 4 -> Along with using the CPU, jobs also require I/O. This is good, because it frees up the CPU for other processes while one process waits for the results from the I/O device, other processes can take advantage of the freed up CPU time. NEVER SACRIFICE RESOURCES! The solution would be to run **STCF with overlap**.

However, all the solutions stated above haven't taken into account the undetermined time of each job. The scheduler is not an oracle, and can't see into the future!

Taking this main issue and the problem with making a suitable tradeoff between our metrics, OS God's have come up with the **Multi-level Feedback Queue (MLFQ)** scheduler. We will mention the rules of the scheduler, and identify their problems to reach a final general rule set for MLFQ.

MLFQ: The scheduler maintains distinct levels of *queues* which each assign different *priority levels*. The priorities in the queues determine which job should run and for how long. The priorities vary based on *observed behavior*. It tries to learn and predict future behavior in a fair and efficient manner.

Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
Rule 2: If Priority(A) = Priority(B), A and B run in *Round Robin* manner.

How do we change the priority levels of the jobs?

<u>Attempt 1:</u>

Recall the workload includes a set of interactive short jobs and long CPU-bound jobs.

Rule 3: When a job enters the system, it is placed at the highest priority (topmost queue).
Rule 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).

Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

This set of rules seem to work for cases involving simple long running jobs (simply cascades down to the lowest level and runs based on that), interjecting short jobs (the short job if short enough will fall through levels but still stay above the longer job and will finish before it reaches the lowest level), I/O use (if it uses I/O the CPU time will be given up and the lower priority jobs will use it, but the higher priority job will remain at the same level so that when it comes back it will continue to function as top dog).

The problems with this model include:

- **Starvation:** If there are too many short interactive jobs, they will continuously be run and the longer jobs will never be run.
- **Game the scheduler:** Before time slice is up, issue an I/O operation and thus you remain at the same priority and thus overall get the maximum CPU time. This will monopolize the CPU.
- **Changing Behavior:** Jobs change through different phases of interactivity, but they will not be given the benefit of the doubt.

To solve the first and last problem we introduce a **priority boost –** Periodically boost the priority of all the jobs in the system.

Rule 5: After some time period *S*, move all the jobs in the system to the topmost queue. This way all processes are guaranteed not to starve as they will move up at some point (in a RR fashion). If a process has become interactive then regardless its priority will be boosted, and thus could remain at the new higher level or continue to move down based on the transition. However, the **new problem** is: What should *S* be? This is referred to as **voo-doo constants** as they seem to require some form of black magic to set them correctly. **Ousterhout's Law –** Avoid these weird constants as much as possible! Do this by possibly creating a configuration file filled with default values that could be changed if something isn't quite working correctly.

To solve the gaming issue, we introduce the measure of **accounting** the CPU time at each level of the MLFQ. Keep a track of how much time slice a process used at a given level. Once it has used the allotment, demote it.

Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

This now guarantees protection from gaming!

Issues with *parameterizing* the MLFQ still remain – how many levels of queues? How much time slice per queue? These come through fine tuning the MLFQ through experience and practice and the system you're working on. [Solaris MLFQ uses a set of tables to determine how the

processes priority changes throughout its lifetime, length of time slices, how often to boost jobs etc. Other MLFQ's use mathematical formulas to sort this out [**decay-usage algorithms**].

MLFQ uses history (feedback) as its aid to become a better scheduler based on Rules 1-5 (ignoring 4a and 4b). It utilizes best of STCF/SJF and is fair!

Everything mentioned above, about the scheduler, is assuming a *single core processor* – only one CPU! Issues arise when dealing with multiple CPU's. Lets first understand some of the intricacies of multiple CPU's and then move the discussion to scheduling.

With the advent of multicore processors, tasks are often *parallelized* using *threads*. These can create concurrency and synchronization issues galore!

When dealing with a single CPU, we assume one set of caches dedicated to that CPU to make operations faster after the initial fetch from disk. The principles that govern the fetch to caches involve **temporal locality –** When a piece of data is accessed, it is likely to be accessed again; and **spatial locality –** If a program accesses data at address *x* it is likely to access data near *x*.

With multiple processors, each CPU has its own set of caches. If one CPU is working on a program and caching the relevant data in its cache, the OS may decide to switch the program to another CPU in which case it would not have all the relevant data in its cache. This issue describes **cache coherence**. General high level overview of fixes includes **bus snooping –** each cache pays attention to memory updates by observing the bus that connect them to main memory. If there is an update it will notice the change and either *invalidate* or *update* its copy. [Still doesn't fix write-back caches but the general scheme still works!]

As mentioned before the issues of multiple CPU's mean that programs have to worry about sharing data. **Mutual exclusion primitives** (locks) should be used to guarantee correctness. But these don't come without their respective overheads.

Cache Affinity – Processes build up states in caches on the CPU they are running on. Thus the next time it is run, it is useful to run it on the same CPU since some of its state is still there. Even with cache coherence fixes, processes that run on a CPU like are better off sticking to that CPU in the near future.

Solutions:

1.) **Single-Queue Multiprocessor Scheduling (SQMS):** Take an existing policy and extend it to run on more than one CPU. It has the following problems:
    a. **Lack of scalability:** Locking mechanisms are put into the code, but as the number of CPU's increase then the overhead of locking also increases -> reducing performance.
    b. **Cache Affinity issues:** Because we are using a single queue, the processes have to switch between caches and so no one process sticks to a cache. There could

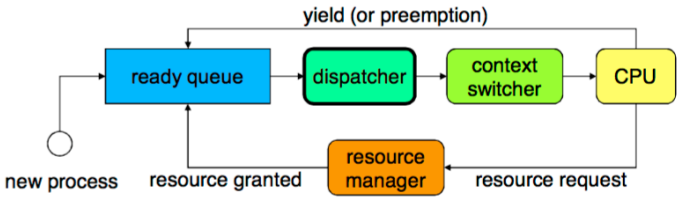
be affinity mechanisms to prevent this issue (along with **migrating** some processes to balance load).

2.) **Multi-Queue Multiprocessor Scheduling (MQMS):** We have multiple queues that scale with the number of CPU's in the system. Then the jobs are essentially independent which avoids problems involving information sharing and synchronization. Then within the queue we just use RR. Lock and cache contention are not problems and provide cache affinity since the jobs remain on the same CPU. The issue comes with:
   a. **Load Imbalance:** Once one task on a queue dedicated to a CPU finishes, then there are less jobs on that CPU and so there is an imbalance in comparison to the other CPU's. (A CPU can even be left idle! While another CPU is juggling multiple jobs in its queue). **Continuous migration** is the solution (keep switching certain jobs between the queues). The technique to do so is known as **work stealing –** Source queue (low on jobs) peeks at Target queue, and will steal one or more jobs. This does incur overhead and trouble with scaling. Thus again a *threshold* must be reached between the stealing and not stealing!

Examples of common multiprocessor schedulers in Linux – **O(1) [MQMS – priority based], CFS [MQMS – proportional share like Stride scheduling], BF Scheduler [SQMS – proportional share on complicated scheme called Earliest Eligible Virtual Deadline First (EEDVF)].**

SOME NOTES FROM CLASS:

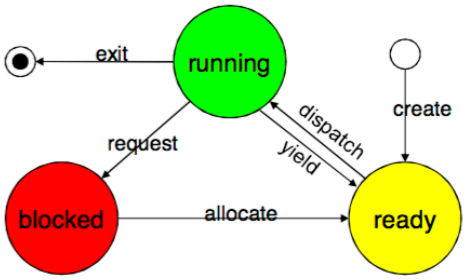
Turnaround Time – Time to completion.
Latency – Time between stimulation and response.



Throughput – Operations per second for the same problem.
Response Time – time to complete a service request. Some delays are not the schedulers fault.
Fairness – Equality of distribution per user, per process. Based on the standard deviation.

The metrics we use for scheduling are:

- Mean time to completion for a particular job mix.
- Throughput
- Mean response time (time spent on the ready queue). Can't always be based on the scheduler, maybe the I/O devices are reducing the response time, but they're not always up to scheduler.
- Overall "goodness" – requires a customer specific weighting function. Often stated in Service Level Agreements.



Quality of Service – How well can you guarantee a service level agreement.

The figure on the left shows the basic scheduling model. A process may be blocked for **I/O, suddenly getting a resource you asked for, external events.**

Real time applications are generally *non-preemptive*. Such tasks focus on throughput as our metric.
Sometimes you don't need a scheduler in **real time applications –** Everything follows a certain order, just fucking follow it.

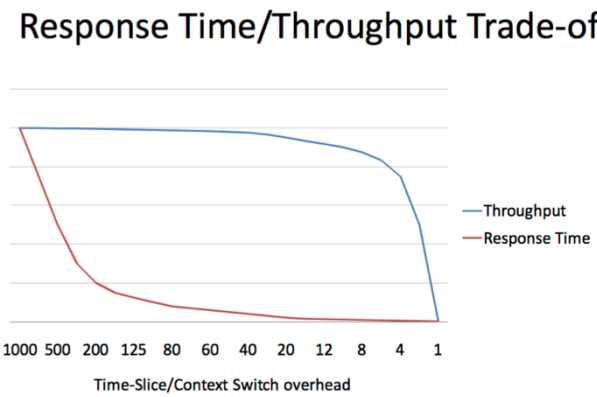
SJC may face unbounded wait times – Is this fair? Is this "correct" scheduling?
Real-time applications don't give a shit about fairness. Starvation is completely acceptable, because you're facing hard deadlines that could result in catastrophe, and so at that moment what matters is what can solve the situation – not fairness.

Fair share comes with preemptive scheduling. Over the course of some time splice everyone will get equal access to a resource such as the CPU.

To force processes to yield, we can use system calls or clock interrupt (based on the LDE); but before we return to the process make sure we consult the scheduler to make sure their priority/state is accordingly adjusted.

Note: Timer interrupts have a pretty significant overhead. These numbers for the timer interrupts come from *tuned parameters* from where we get trade-offs. If you make them too long, you make not yield at correct points, but if its too small then you're going to spend more time in the overhead. It's a fine art to hit the "sweet spot".

Round robin has some context switch overhead that adds to the overall time spent. But it greatly improves response time.
Mechanism/Policy Separation – separate the idea of how to allocate resource, versus which method to use to do so.



WEEK 3

Monday:

Virtualization!

Method 1:

Early systems didn't really provide much of an abstraction. The OS code was loaded at the start of memory and the current program was loaded ~64KB to max size. With the birth of multiprogramming (running multiple programs at once) and time sharing (many programmers using different programs) this method was no longer valid. With these constraints *utilization* and *interactivity* become key factors.

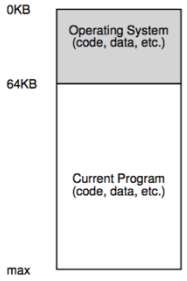


Figure 13.1: **Operating Systems: The Early Days**

Method 2:

The notion of having one process running for a short while then switching out – giving each process access to all of memory – was also a horrible approach as it is extremely slow to save to disk each time doing a switch!

Method 3:

Now each process has its own little chunk of memory allocated to it in main memory, with free space for alignment and other issues around. Each process goes through virtualization, that is thinking it has the entire addressable space available to it.

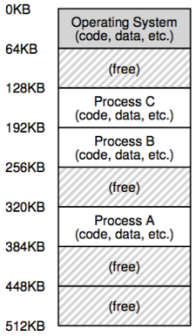


Figure 13.2: **Three Processes: Sharing Memory**

Thus came about this system of mapping virtual addresses to physical addresses to assign good amounts of space to each process. The goals of virtualization include:

- Transparency – Abstract yourself, from each processes POV and think you have the entire disk space.
- Efficiency – solve the constant read and writes to disk/memory using hardware parts such as the TLB and use software based algorithms for the right flushing algorithms.
- Protection – Isolate each process from each other, so they cannot conflict on memory or snoop on other application's information!

Note that all the addresses you will see as a programmer are actually virtual addresses! Only the OS knows the locations in physical memory.

Memory API:

There are two parts associated with a processes memory in its addressable space – **stack** (typically grows upwards from bottom) and **heap** (typically grows downwards after the text and stuff). The stack isn't as exciting – the compiler will allocate all static and initialized local variables in here when compiling, it will deallocate this space when it sees that you are done with it – such as when you exit a function, the function frame will be popped off the stack.

The heap on the other hand is dynamic allocation and for older languages like C, put the responsibility of allocating and freeing the space up to the programmer. Most new languages like Java have **automatic memory management**, such as a **garbage collector** that will run and figure out what memory locations you no longer have reference to and free it. The two most common memory management functions in the C standard library are:
Malloc – Gives you back a void* pointer to the newly allocated space (or NULL in case of failure). Generally we use the **sizeof** command to calculate the size of memory we want to allocate – be careful though, trying to calculate the size of dynamic types is going to give you issues. Also when it comes to strings, sizeof can be silly based on which version of malloc you are using. Stick to always doing str.len + 1.
Free – Corresponding deallocation mechanism to free the memory you just allocated. All you have to pass is the pointer that points to the newly-allocated region.

Common issues:

- **Forgetting to allocate memory –** Segfaults!
- **Not allocating enough memory –** Undefined behavior
- **Forgetting to initialize allocated memory –** Uninitialized reads are undefined.
- **Memory leak –** Could lead to crashing your system and requiring a boot. Don't do this.
- **Dangling pointer –** Don't free memory before you're done with it. Dereferencing a dangling pointer is as bad as that with NULL pointers.
- **Double free –** Undefined behavior.
- **Invalid frees**

Note that the above two methods are not actually system calls but library calls that system calls that are effectively used in buffered methods (such as *brk* and *sbrk*).
Mmap can be used to allocate *swap space* (can be treated as a heap).
Calloc – zeroes the memory you just allocated
Realloc – copy the old memory and allocate more memory
Base-Bound Relocation:

Interposition – OS will interject at critical times to ensure it maintains control over hardware.

Hardware-based address translation – hardware transforms each memory access (ILP – fetch, execute etc), changing the **virtual address** provided by the instruction to a **physical** address where the information is actually located.

The OS must set up the hardware to **manage memory** keeping track of the free spots and information about what was at a specific spot.

Dynamic Relocation/Base and Bounds – every program starts at 0 (virtually) and the *base register* has its value set to the initial value of the physical address. Then every successive transparent relocation translation will convert the virtual address to physical through: *virt = phys + base*. This process is known as **address translation** and because it is done at runtime based on the initial PC value, it is called *dynamic relocation*. The **bounds** register on the other hand will contain a value to determine its limit. Thus every time it does a translation it will check its bounds register to make sure it doesn't go beyond its limit. This ensures protection!

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

Changing values of these registers are **privileged instructions** and thus can only be done in kernel mode.

Memory Management Unit (MMU) – the part of the processor that helps in the translation. These base and bounds registers exist here amongst other circuitry. They are *specific to each CPU*.

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via* ***free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise; likely action is to terminate offending process*

Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**

Two important data structures the OS has to keep a track of are:

Free-List – A list containing the physical addresses of free space that is scattered throughout memory due to random load points. This lists all the places a process could be relocated to or newly loaded into.

Process-Structure – A list of all the processes that need to be run in resident state.

Based on any errors that come about during the virtualization of the address space, exceptions are thrown and the OS arranges a corresponding exception handler (from the trap or interrupt table) to handle this.

The OS must also be able to kill a process, and then save its information until the zombie process is reaped and thus must manage all this information in the process structure of **Process Control Block.** This is also useful for storing states of processes during context switches. When

a process is no longer running (blocked) its location in memory is changed by first being *descheduled, copying the new process location, and then updating saved base registers*. This provides a level of transparency as far as the process is concerned!

All of this execution still follows **LDE**.

Though this is an efficient virtualization mechanism, because the stack and heap of a process are not too big, we get a lot of space wasted in the process itself. This waste is called **internal fragmentation –** If not all the allocated space inside a unit is not used up.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of... system call handler timer handler illegal mem-access handler illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table initialize free list		
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
To start process A: allocate entry in process table allocate memory for process set base/bounds registers return-from-trap (into A)		
	restore registers of A move to user mode jump to A's (initial) PC	
		Process A runs Fetch instruction
	Translate virtual address and perform fetch	
		Execute instruction
	If explicit load/store: Ensure address is in-bounds; Translate virtual address and perform load/store	
		...
	Timer interrupt move to kernel mode Jump to interrupt handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) (including base/bounds) restore regs(B) from proc-struct(B) (including base/bounds) return-from-trap (into B)		
	restore registers of B move to user mode jump to B's PC	
		Process B runs Execute bad load
	Load is out-of-bounds; move to kernel mode jump to trap handler	
Handle the trap Decide to terminate process B de-allocate B's memory free B's entry in process table		

Figure 15.5: **Limited Direct Execution Protocol (Dynamic Relocation)**

Segmentation:

The idea behind segmentation is simple. Take the address space of a process and break it up into chunks or *segments* and then apply the dynamic reloading mechanism using base and bounds registers for each segment. This then prevents any internal fragmentation as all addressable space for each process will be scattered and thus will not have a chance to waste memory "in-between" the stack and the heap.

The OS needs to know what offset to apply to which segment as they will all be different, and so support from hardware is required.

There tends to be three segments that a process is broken into – **code, heap and stack** – and thus 2 bits are required to index the segments. Of course you could have a finer granularity and have more segments for which you would need more bits!

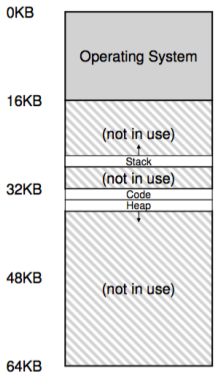


Figure 16.2: **Placing Segments In Physical Memory**

Segmentation fault comes from this mechanism of segmentation, wherein you access a part of memory that is not part of the processes segment!

Explicit Approach – This is the method used to reference the segment as stated before. The **virtual address** is broken up into top 2 bits for the segment and the rest for the offset.

Implicit Approach – The hardware determines the segment by noticing how the address was formed. If from PC then the address is within the code segment, if by stack/base pointer then stack segment, else heap segment.

However, we notice the stack effectively grows backwards. We could take our *offset* and then subtract it from our *max offset* and then add this result to the *base* of the stack relocation to get our correct physical address for growing backwards. Thus more support from the hardware comes in the form of an extra bit to determine if the segment grows "positive".

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Figure 16.5: **Segment Register Values (with Protection)**

With segmentation another benefit that arises is the ability of **code sharing –** Using the same code segment amongst different processes. In order to do so while ensuring protection requires **protection bits** that determine if a segment is read, write or executable! This way isolation is maintained.

Fine-grained Segmentation – The address space is chopped up into even more segments than just the three mentioned above. This requires a **segment table**. The thinking behind this was that through finer segmentation, the OS will be able to better learn which segments are in use and thus manage memory more efficiently.

Coarse-grained Segmentation – The method of breaking up the address space into larger segments as described above with just 3 segments.

OS Support problems with Segmentation:

1.) In context switching the OS must now additionally save the segment registers and must respectively restore them, before the process is run.
2.) Now that there are multiple segments per process, the amount of space located in between these segments in physical memory could get wasted – **External Fragmentation**. We could **compact** the memory, by regularly collecting the segments and putting them contiguously but this is **very expensive**. **Best-fit, worst-fit, first-fit and buddy algorithm** are just a few algorithms that aim to manage the *free-list* so as to keep as much of this "external" memory freed up and so to effectively place the segments so as to not waste any memory.

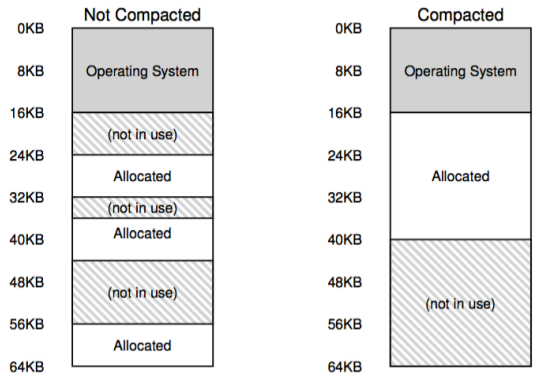


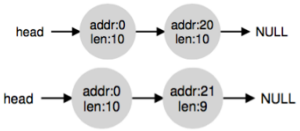
Figure 16.6: **Non-compacted and Compacted Memory**

Segmentation is still not flexible enough. If our model of how the address space is being used doesn't exactly match how the underlying segmentation has been designed to support it, segmentation doesn't work very well.

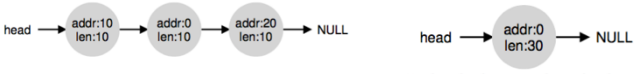
Free-Space Management

How does the OS manage all the free space that comes about from *internal and external fragmentations*? We care more about **external fragmentation** that is more interesting. To recall this comes about from a result of segmentation. Example: You have 30KB of space, and we have segments from 10->20KB, and though we have 20KB free they are not contiguous and thus fragmented.
Internal fragmentation happens if the allocator hands out too much memory, and all of it is unused and thus you have wasted space.
Compaction only helps with segmentation, because once malloc is called, that chunk of memory is owned by the process and cannot be moved until free is called.
Free space is denoted as chunks in a free list. The allocator performs two key actions with this free list:

- **Splitting –** If the space requested is less than or equal to a chunk of free space in the free list, the allocator will split this chunk to satisfy the requirement while the rest will just continue to exist in the free list.



- **Coalescing –** If the allocator notices chunks of free space that are right next to each other, it will merge these chunks together for more efficient free space allocation!



But how does this memory allocator actually track all this given out space that makes the interface for malloc and free so simple? The answer relies on a **header** that is allocated to each chunk of memory allocated. This header contains information such as **size** and a **magic number** (used as a unique identifier essentially for extra integrity checking). There could be more information in the header to make certain operations faster, but this is the basic case. We also don't want to inflate our header too much as this would waste memory. Thus when we malloc we actually allocate a little more space – to account for the header as well (everything with the memory management with malloc actually has to do with N + sizeof(header)).

The free list can for simplicity be thought of as a linked list. Thus the process remains the same in memory. Once a specific chunk of memory is requested, the chunk **splits** into a free chunk and the chunk being used.

We can essentially create the heap using the *mmap* system call to gather space from the swap space on disk! There are definitely many other ways of creating a heap.

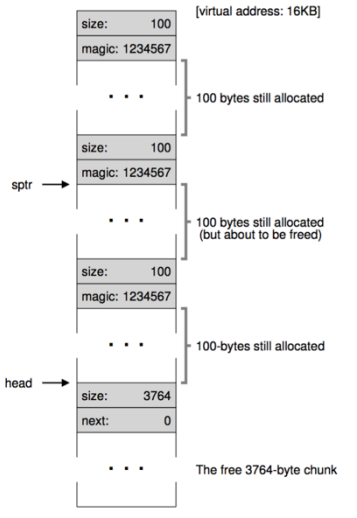


Figure 17.5: **Free Space With Three Chunks Allocated**

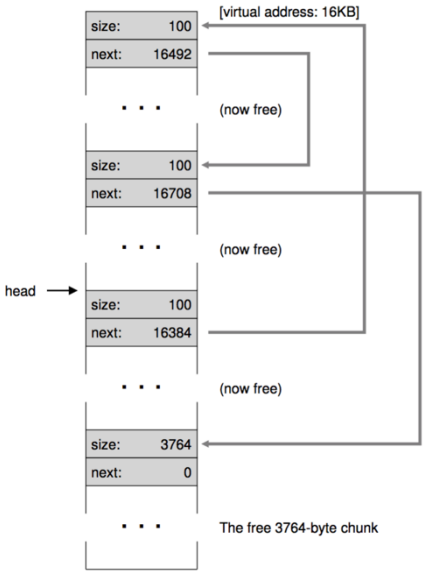


Figure 17.7: **A Non-Coalesced Free List**

If we want to free the space then the pointer we feed into free(void* ptr) will point to the start of the free chunk and then clear the space based on the size in the header, and the next pointer will point to the rest of the free space. **We coalesce all the free chunks when we have two or more contiguous chunks of free space.**

Most traditional allocators start with a small-sized heap and then re- quest more memory from the OS when they run out. Typically, this means they make some kind of system call (e.g., sbrk in most UNIX systems) to grow the heap, and then allocate the new chunks from there. To service the sbrk request, the OS finds free physical pages, maps them into the address space of the requesting process, and then returns the value of the end of the new heap; at that point, a larger heap is available, and the request can be successfully serviced.

Basic Allocation Policies:

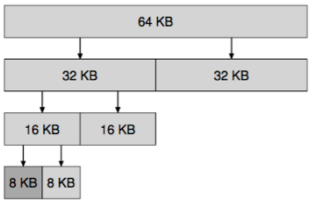
1.) **Best/Smallest Fit –** Loop through the free list until you find the smallest chunk of memory that meets the requested free space, and keep the remaining of that chunk in the same position in the free list. Bad because it requires an exhaustive search of the entire list.
2.) **Worst/Biggest Fit –** Loop through the free list until you find the largest chunk of memory, and then allocate the requested amount and keep the remaining of that chunk in the same position. Bad because it requires an exhaustive search of the list. One would assume this would be better than best fit because it fragments the space into larger (more useable) chunks but studies show that it sucks.
3.) **First Fit –** Find the first block big enough to service the request. This has the advantage of speed but pollutes the beginning of the list with small objects! We could use **address-based ordering –** keep lists ordered by the address of the free space, coalescing becomes easier, and fragmentation tends to be reduced.
4.) **Next Fit –** Store the location of the last freed space and perform first fit from that point onwards. Fast and spread the search more uniformly thus avoiding the splintering at the front.

More Complex Policies:

1.) **Segregated lists –** If one application has one or a few popular sized requests it makes, keep a separate list to manage objects of that size, the rest go to a more general memory allocator. Essentially we create a specialized allocator for common requests by an application.
    a. **Slab allocator –** The kernel allocates a number of **object caches** for frequently used things such as locks and inodes. These can each be thought of separate segregated lists. When a object cache is running low on free space it will **request**

a slab from the more general memory allocator that is a multiple of the page size. When the reference count in the cache is 0, then the general memory allocator can absorb this cache thus coalescing memory! The state of the objects in the slab allocator caches are in a pre-initialized state when free. This avoids unnecessary overhead in initialization and destruction of the objects.

2.) **Binary Buddy Algorithm –** Recursively divide the large free space (of size 2^N) recursively until you reach a size that matches most closely to the requested free space – can lead to issues with wasted internal space = **internal fragmentation**. However once done, the general allocator simply looks for the "buddy" memory chunk that this binary divide created and simply coalesces them! This is an efficient coalescing algorithm. It is effective because each buddy is only 1 bit off from the other which is determined by the level they reside in the tree.



These approaches have issues with scaling. Because the free list was assumed to be a linked list. Of course modern day memory allocators maintain more complex data structures that ensure faster access, trading simplicity for performance (like balanced binary trees, splay trees or partially ordered trees). We must also remember that memory allocators for multiprocessing systems have more complex thought behind them!

Wednesday:

Paging:

Paging is the idea of breaking up the processes address space into a number of **fixed-size** units – **pages**, and physical memory is seen as an array of fixed sized slots called **page frames**. Each of these frames contain a single virtual memory page.

This method has the following advantages:

- *Flexibility* – the system can support the abstraction of an address space effectively regardless of the use of the process address space. Direction of stack and heap don't matter.
- *Simplicity* – Avoids the fragmentation caused by segmentation, and the OS simply has to use the page numbers and act like its indexing an array (simple case).

The OS may keep a **free-list** of all the free pages for easier access. Note that the free-list spoken about earlier was in reference to segmentation method, but the allocation mechanisms discussed still apply.

Page Table – A *per process* data structure that stores the location of the physical page numbers in physical memory. The information in here is accessed using the **virtual page number (VPN)**. As a simple data structure we can think of the page table as being **linear** like an array, but more complicated data structures exist. **The role of the Page Table is to store address translations.**

An exception of per process page tables, are **inverted page table**.

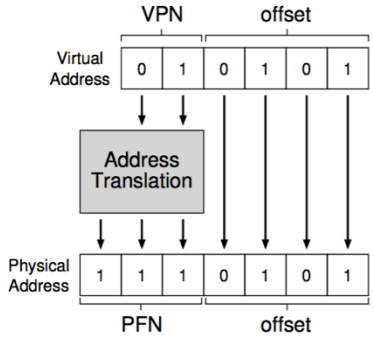


Figure 18.3: **The Address Translation Process**

Virtual Addresses – Each virtual address (the addresses that processes actually use) is broken up into two key parts – **Virtual Page Number** and **Page Offset**. The VPN is used to index into the page table and the page offset is used to index into the actual page that exists in physical memory.

Physical Frame/Page Number – This is the value of the actual address in physical memory. Loaded in when the process starts, the PPN refers to the location of that page in memory, and along with the offset (which is the same in the virtual address and the ppn) loads into the specific byte into the page in the page frame to get the instruction.

The problem with Page tables is that they themselves have to be stored somewhere, and based on how many instructions exist in the process, given each page is a **fixed size** then the page table can get pretty fucking large.

The page table for each process is stored somewhere in memory. We'll get back to this a little later.

Contents of a page table entry:

1.) **PPN** – The location of the information in physical memory that is accessed by the vpn.
2.) **Valid bit** – Is the translation valid or not? All the space in between the initial code, heap and stack will be empty and so is invalid.
3.) **Protection bit** – to mark whether the data is read, write or execute.
4.) **Present bit** – Is the page in physical memory? If not, it has to be **swapped out.**
5.) **Dirty bit** – Indicates whether the page has been modified since it was brought into memory.
6.) **Reference bit** – tracks whether a page has been accessed (determines which pages are popular and thus should be kept in memory). Important in page **replacement.**

The actual **page table entry** must also be stored somewhere, and for now we assume it lies in the **page-table base register**.

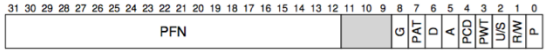


Figure 18.5: **An x86 Page Table Entry (PTE)**

```
1  // Extract the VPN from the virtual address
2  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4  // Form the address of the page-table entry (PTE)
5  PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7  // Fetch the PTE
8  PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access is OK: form physical address and fetch it
17     offset   = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

Figure 18.6: **Accessing Memory With Paging**

Problems with paging:

- Without proper care, paging will be slower because of all the translations and all the fetches from memory.
- Memory waste because of the size of page tables and where they must be stored.

TLB:

Because paging can be pretty slow with common access to memory all the time (and access to disk for swapping) we use the age old trick of bringer important data closer to the memory hierarchy by employing a *cache* known as the the **TLB.**

Translation Lookaside Buffer – hardware cache of popular virtual-to-physical address translations.

TLB hit – When trying to perform the translation to get to the page in memory, the hardware first looks in the TLB and in this case finds the relevant translation => Success. This means extremely quick access.

TLB miss – When trying to look in the TLB for the translation and notice its valid bit is not set (i.e. the translation doesn't exist) it has to go to memory and grab the translation, update the buffer and then repeat the look, now to find it in the TLB.
This is extremely slow as it performs multiple fetches. This occurs mostly on first time access – **cold-start miss (compulsory miss)**. [**capacity miss –** cache became to full and had to evict item; **conflict miss –** which lines can things be put on such as in direct mapped caching or set associative caches].

The premise behind the TLB is *locality* that is with an initial TLB miss, the next few translations will be much faster due to either *spatial locality* – the translation I just made now will also apply to the next 3 pieces of data or so (like in an array) – or *temporal locality* – once I have made the translation, I am likely to make it again (such as reusing variables).

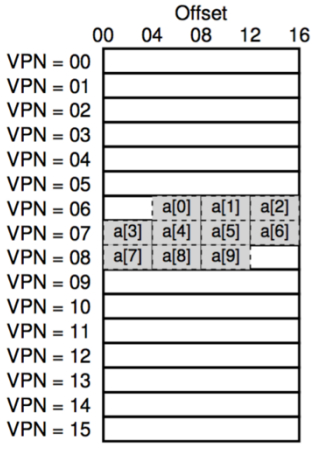


Figure 19.2: **Example: An Array In A Tiny Address Space**

Our **hit rate** can be calculated by noticing the number of hits and misses during the execution of a piece of a program. Try to make this as close to 100% as possible and notice that upon first access the hit rate might be a little low, but upon frequent access, the hit rate will be closer to 100%!

Two ways of handling TLB Miss:

1.) **Hardware-managed TLB (CISC):** Seen in older architecture, the hardware would know where the page tables are located in memory through the **page table base register (PTBR)** as well as their *exact format*. Upon a miss, the hardware would walk through this table and find the correct PTE and extract the desired translation, update the TLB and retry the instruction. This can be seen in Intel x86.
2.) **Software-managed TLB (RISC):** On a TLB miss, the hardware will *raise an exception* and pause the execution stream and raises the privilege level to *kernel mode* and jumps to a **trap handler**. The difference between this trap handler and that run by system calls is that the *return-from-trap* instruction for the TLB must *resume execution at the instruction that caused the trap,* while the system call would just jump to the next instruction. The hardware must *save a different PC* in order for normal execution upon return.
Trouble: Make sure not to run an infinite chain of TLB misses. Solution: Keep TLB handlers in physical memory (**unmapped**), or reserve entries in the TLB for permanently-valid translations and use some of those slots for the handler code – always a hit.
Advantage: Flexible and simple!

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else if (CanAccess(PTE.ProtectBits) == False)
16          RaiseException(PROTECTION_FAULT)
17      else
18          TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19          RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

What's in the TLB?
The TLB is a **fully-associative cache** and so it performs an exhaustive search of the TLB (in parallel by the hardware) to find the hit line! [**Directly mapped cache –** Maps to exactly one line in the cache and so if multiple translations map to the same line this could raise problems with maintaining cache hits; **Set Associative cache –** Contains multiple lines (sets) for cache hits!]

The TLB generally contains the VPN, PFN, valid bit, protection bit, **address-space identifier (ASID)** and a dirty bit.
Note that the the TLB valid bit talks about whether that translation has been moved from physical memory into the TLB, while the valid bit in the page table talks about whether the mapping to that specified location in physical memory is actually valid or not. If TLB bit is invalid, it will be a TLB miss and thus grab the data from memory, if PTE bit is invalid, it will generate a trap and kill the process.

Now because the Page Tables and their entries are process specific, we can imagine that there could be issues with **context switches** for the TLB! Thus we have to make sure that the translation for two processes don't exist in the TLB at the same time as we could get ambiguity in matching and this could result in bad errors!

Solution 1: Flush out the cache every time we do a context switch (set all valid bits to 0), and thus the translations are always independent, but this could become costly if the OS is running a lot of processes and performing regular context switches!
Solution 2: Use the hardware **ASID** as an identifier for which address space (almost like a PID but fewer bits) to do the translation into! If we have 8 bits for this, then we can 256 processes running at a time. If the machine exceeds this, then maybe go to flush the least recently used ASID translations?

Having multiple VPN's map to the same PFN is acceptable however, as this denotes **shared code** between processes which is always beneficial when its required such as **shared libraries!**

So how do we decide what data to flush out when we need to? There are two common approaches to this:

1.) **LRU (Least Recently Used) –** Take advantage of locality in the stream. If certain translations haven't been used in a while => not popular => flush these to make room for potentially more popular translations.
2.) **Random Policy –** This is much easier to implement and basically just randomly selects translations to flush. Can sometimes be better than LRU (such as the case for looping over *n+1* pages with TLB size of *n*).

We mentioned above the important things to have in a TLB entry. Note that sometimes the VPN will probably take fewer bits than expected (such as 19 bits in a 32-bit address space which would mean it should take up 20 bits) because half of the memory space is reserved for the kernel and so we have no use in including that in our translation.

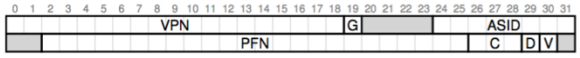


Figure 19.4: **A MIPS TLB Entry**

We also have a **Global bit (G)** that refers to data that is shared amongst almost all processes and should always be in the TLB for constant hits. The **dirty bit (D)** is used to mark when a page has been written to (to update the value in memory). There is also a **page mask** field to support multiple page sizes. A few of the bits in a TLB entry are reserved for the OS (as mentioned before with the trap handler code) which sets a *wired register* for the OS – used for critical mappings where a TLB miss could be catastrophic.
MIPS TLB is software managed and so it has special instructions to manage the TLB. These instructions are **privileged** as manipulating them as a user could wreak havoc with the processes!

Culler's Law – RAM isn't always random (i.e. not all access is quick as another) because randomly accessing something – if the number of pages accessed > TLB coverage – can lead to performance penalties.

Two problems exist with TLB's as we know them now:

1.) **TLB coverage –** Number of pages a program accesses in short time is more than the number of pages a TLB can hold => slow run. Solution: increase the size of a page so as

to store more information in a single TLB! That is if we have a bigger page, then the address that we translate to has more content and thus the hit rate increases. **Database Management Systems (DBMS)** exploit this feature.

2.) **Bottleneck in Physically-indexed cache –** Address translation has to happen *before* the cache is accessed which can be a problem with TLB. Solution: **Virtually-indexed cache**.

Going back to the idea behind the locations of these pages being in memory, one can conclude that this may become unrealistic as memory can run out. Where do all the pages go then?

Swap space – Designated space on **disk** where the OS can *swap* out pages from memory into disk and out of disk into memory! Because this requires disk access, it is a much slower operation but gives the flexibility of running a lot of programs especially for multiprogramming and ease of use (method used before was **memory overlays** where the programmer was in charge of finding space for his/her allocation!).

Note that swap space is not the only region in memory the OS goes to. All programs (especially nowadays) are stored as binary on disk, and then when they are run, they are unpacked and moved into memory from disk by the OS.

To do this the OS remembers the **disk address.** The general process of execution is as follows:

- Extract VPN => Check if in TLB => TLB Hit = YAY
- Extract VPN => Check if in TLB => TLB Miss => Use PBTR to find page table => Look up PTE from VPN => If page in memory, extract PFN from PTE => install into TLB => retry instruction => TLB Hit = *EH*
- Extract VPN => Check if in TLB => TLB Miss => Use PBTR to find page table => Look up PTE from VPN => Page not in memory (determined by **present bit**) => Page Fault => Page Fault Handler swaps the page into memory from disk => update page table as present, PFN field of PTE => TLB miss … => TLB hit = Fuck that was long

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                  // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else
16         if (CanAccess(PTE.ProtectBits) == False)
17             RaiseException(PROTECTION_FAULT)
18         else if (PTE.Present == True)
19             // assuming hardware-managed TLB
20             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21             RetryInstruction()
22         else if (PTE.Present == False)
23             RaiseException(PAGE_FAULT)
```

Figure 21.2: **Page-Fault Control Flow Algorithm (Hardware)**

```
1  PFN = FindFreePhysicalPage()
2  if (PFN == -1)              // no free page found
3      PFN = EvictPage()       // run replacement algorithm
4  DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5  PTE.present = True          // update page table with present
6  PTE.PFN     = PFN           // bit and translation (PFN)
7  RetryInstruction()          // retry instruction
```

Figure 21.3: **Page-Fault Control Flow Algorithm (Software)**

Page Fault – Essentially a page miss, that is the page is not present in memory and thus some mechanism has to fetch it from the swap space on disk.

Page Fault Handler – An OS run handler uses the bits in the PTE for a disk address and issues an I/O request to grab the data. While this happens the process is **blocked** and so other programs can run at this time. Yay for multiprogramming and time-sharing!

Page in – Bring page into memory from swap space.
Page out – Push a page currently in memory to swap space to make room for other pages (replace that shit).

Note that in the processes described above we could have also hit an invalid page access which would raise an exception and a corresponding trap to probably kill the process.

When do replacements occur?

OS keeps a **low watermark** and **high watermark** that defines the limits of space in memory before which we perform replacements. If the current free space in memory is lower than *LW* then page out pages until we get *HW* page space available. This is done in a *background thread* or **swap/page daemon.**

Performing multiple replacements at once, we can **cluster/group** a number of pages and operate on them as one cohesive unit which can improve speed.

Page-Replacement Policies:

Because of *memory pressures* the OS has to decide what pages to evict and thus comes in the replacement policy. If we think of our physical memory as a cache to disk, we want to increase our *cache hits* just as we wanted to with the TLB and memory. We wish to calculate the **average memory access time (AMAT).** Using this we will notice that access to disk is very slow and so we wish to minimize page faults as much as we can and so an effective policy is definitely required.

$$AMAT = (P_{Hit} \cdot T_M) + (P_{Miss} \cdot T_D) \quad (22.1)$$

where T_M represents the cost of accessing memory, T_D the cost of accessing disk, P_{Hit} the probability of finding the data item in the cache (a hit), and P_{Miss} the probability of not finding the data in the cache (a miss). P_{Hit} and P_{Miss} each vary from 0.0 to 1.0, and $P_{Miss} + P_{Hit} = 1.0$.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: **Tracing The Optimal Policy**

Belady's optimal policy – This policy quixotically states that the page we must replace is that which will be accessed *furthest in the future* as it does not need to be used now, it can be swapped out for something more important at this time.

Unfortunately, neither us nor the OS can see into the future and so this simple but effective policy is near to impossible to implement. Similar to scheduling policies, we look at some basic page-replacement policies starting from the most basic approaches:

1.) **FIFO –** Use a queue to place pages in and the page that was first in (tail of the queue) is the first to be evicted. This is simple to implement. But this places no weighting on the importance of pages and thus is a horrible implementation with generally more misses. Suffers from **Belady's Anomaly –** As cache size increases, the hit rate actually decreases!

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Figure 22.2: **Tracing The FIFO Policy**

2.) **Random –** Picks a random page to replace under memory pressure. Also simple to implement but has no intelligence in its picking of blocks to evict. Because random is in its essence random, it could do better or worse than a lot of algorithms, and on average actually does pretty well, but once again this up to the luck of the draw which is not something engineers/scientists should work on.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Figure 22.3: **Tracing The Random Policy**

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: **Tracing The LRU Policy**

3.) **LRU –** Evict the least *recently* used pages. This uses history of the page involvement to make a decision about what to evict. This is smarter than both the policies discussed above and has the added benefit of having a **stack property –** A cache of size N + 1 naturally includes the contents of size N => increase size = increased hit rate.

4.) **LFU –** Evict the least *frequently* used pages. Similar to LRU but uses frequency as the metric for eviction.

Workload examples – We use *no-locality, 80-20 (80% of references made to 20% of pages), Looping (loop over a set of pages).*

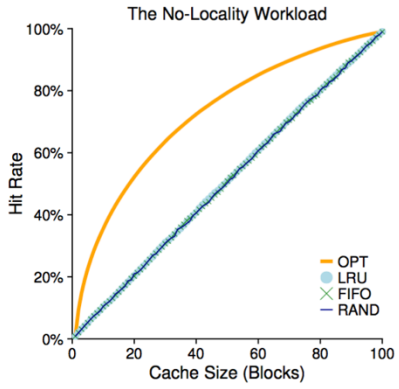


Figure 22.6: **The No-Locality Workload**

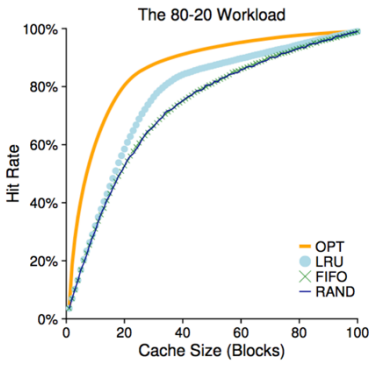


Figure 22.7: **The 80-20 Workload**

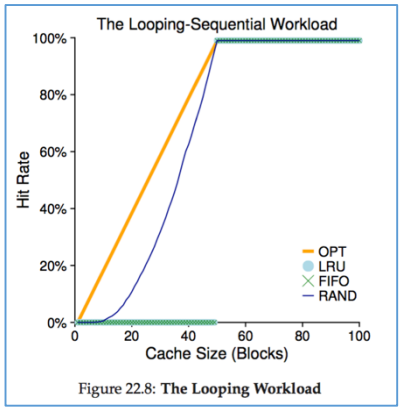


Figure 22.8: **The Looping Workload**

With historical algorithms we generally need an additional data structure to hold all the **memory references** made, and if care is not taken to implement this, it's overhead could

outweigh the performance benefits. We could use some hardware to store some reference to this information, but this suffers at the hand of scaling as it becomes memory intensive! So instead of holding every reference, we could get by approximating instead!

We use the **use/reference bit** mentioned earlier to keep a track of the most recently used page. Algorithms that then periodically go through the list of pages checking for 0 used bits for eviction are ideal in approximating LRU. **Corbato's Clock Algorithm** – place a clock hand on any arbitrary page and check its use bit. If it is 1, change it to zero and move forward repeating the process until you get to a 0 use bit and evict that, else evict it because it is 0.

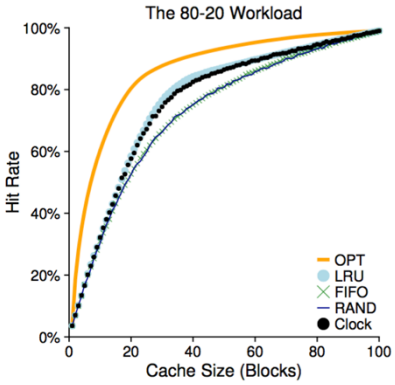


Figure 22.9: **The 80-20 Workload With Clock**

Note that when we talk about eviction we are not necessarily talking about writing back to disk, but rather just clearing it in memory, and if we need it again we fetch it from disk. However, if we notice that data in the page in memory has been **modified** (that is, the page is now **dirty**) then we must perform a write to disk before the eviction which can be expensive. This is why we have a dirty bit! The clock algorithm then works to first find an unused and clean page, else unused and dirty, else…

Page replacement is only one of the many VM policies implemented. The OS also has to decide when to page into memory – **page selection**. A common approach is to go off of **demand paging –** when the page is needed it is "demanded" out of disk into memory. Another approach is **prefetching –** predict if a page is necessary before actually calling to it; can be based in terms of spatial locality i.e. if page P is being demanded, the likelihood of P+1 being needed is also high.
Another policy is the **write-out** policy – how the OS writes to disk. The common approach is **clustering** to gather a list of pages that need to be written to disk and then do it in one go.

Thrashing – The memory demands of running processes exceeds the available physical memory, which results in constant paging.

Old systems approach – System could decide not to run a set of processes in the hope of reducing the working sets – **admission control –** sometimes better to do less work well than try to do everything at once poorly.

Modern systems approach – When we are running out of memory, a daemon (process running in the background) will kill the most memory intensive program – **out-of-memory killer**. This could potentially close an important application!

While these algorithms are somewhat important, modern systems compromise by simply buying more memory or improving the access times between memory and disk (as with flash SSD's!).

Note that while LRU is great, it is only great on a per process basis. Imagine combining the scheduling technique of Round-Robin and Global LRU. Well because we are switching processes out so quickly, the least recently used pages won't matter as they will have to be replaced for the new process.

So we use another metric to help guide us to a better mechanism. This is the **working set** that can be defined from the thrashing. If we get constant amounts of thrashing, then we notice that the working set (the number of pages this process requires) is very large and so we look for processes who are using more memory than they need for their working sets and allocate based on that. This is similar to exploring the interactivity of a process in scheduling and using

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 probably have too many processes to fit in the available memory, and we need to start swapping some of them out.

this as a metric to schedule some processes over another.

Now we notice that this **page stealing** algorithm works brilliantly because it is a **dynamic equilibrium mechanism –** maintains a level of balance by stealing and gaining!

SOME NOTES FROM CLASS:

Running in the background can mean either things I'm not directly interacting with, or processes that we don't directly instantiate but runs to save state such as email updates. **Daemon process –** A process that the OS (or somebody else) started, but are waiting to serve and perform some function. Something that I started but am not looking at or something I didn't even start.

The heap is being managed as a free list next fit allocation by malloc. Done by variable partition allocation.

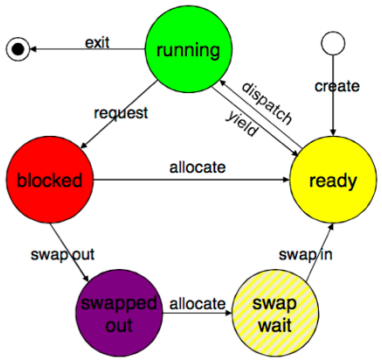
There is a part of memory that is not divided into pages (analogous to heap) that can be used for different things such as networking and device drivers. The OS uses this space for its things. There is a **third type of memory** as well. That is not this paged memory nor the malloc heap memory the OS uses, but rather **memory aside** that can be used by mmap (that doesn't map files from disk, but rather steals memory from this non paged memory) by a process to reserve space for whatever the process requires (such as a super write back cache for IPC), and upon booting, this application will tell the OS that this space is reserved and so will be divided into this third type of memory.

Note that segments are no longer units of relocation because of paging.
Read or write is a one time copy of the contents of a file into the system, mmap creates a permanent persistent copy of the file.

A segment is implemented as a set of virtual pages. Thus our **internal fragmentation –** averages only 0.5 page of last one; **external fragmentation –** completely non-existent (we never carve up pages).

Note that we have two types of schedulers – process schedulers which is what we spoke about in week 2, and memory scheduler which is what we are talking about with the working set clock algorithm. These two schedulers work together to make things run smoothly with virtualization.

scheduling states with swapping



Adaptive replacement cache (ARC) – Don't use LRU to replace, but use it to specify a candidate for replacement and then use LFU for replacement. Period of time over which you're averaging so your time period for the ARC has to cover this time frame.
Pre-loading – Use previous information to pre-load pages in order to reduce page faults.

WEEK 4

Monday:

Inter-Process Communication

When talking about IPC we start by focusing on data exchange between the different processes. These interactions can be classified into uni-directional and bi-directional.

1.) Uni-Directional Byte Streams – Pipes

Can be opened by a parent and inherited by a child process for communication.

The parent produces a byte stream output that is a well defined input for the child process through the pipe. The programs in the 'pipe-line' operate independently unaware of the other programs. The suitability of the data transmitted is up to the agent that created the pipe. The data for the most part is unstructured unless an external parser does so.

We can think of pipes as temporary local files that are special in that they:

- Remain open even if all the data has been exhausted
- If the writer gets too far ahead of the reader, the system will Block until they are caught up – **flow control**.
- Writing to a closed pipe will cause a SIGPIPE.
- The file is automatically deleted after use.

There is no encryption or authentication with pipes since they are a **closed system.** They exist in the kernel and not in the file system.

2.) Named Pipes

These are persistent pipes that can be used for communication amongst multiple different processes. However, there is no way of knowing who is writing and reading at what time, all the members must be on the same node and the data may become interspersed because of concurrent reads and writes. These named pipes are persistent and so remains in the file system for later use even after its done. The *mknod()* system call can be used to create such a named pipe in C. If the FIFO (as it is called) is open for reading, the process will block until it is opened for writing (and vice-versa). You can send this to the background if you wish. Again a SIGPIPE also applies to this scenario if you try to write to a pipe that has no reader.

3.) Mailboxes

This is a more general IPC. The data is not bytestream but rather each write is stored and delivered at once like a mail. Thus each write has an authentication stamp from its author. Unprocessed mails remain in the inbox even after the death of a reader for use by the next reader. However, this still exists on a single node!

4.) General Network Connections

API involving socket and binding, listening, accepting and connecting all use the basic principles of networking for IPC. Using either byte stream data communication as is with TCP or data gram models such as UDP. This is also the principle behind which higher level services are built on. They allow for worldwide communication but with this there comes added complexity such as security issues for data communication over unknown public networks, discovery of addresses of servers, detecting and recovering from node failures and ensuring for interoperability. The issue that comes with network based IPC is that it is slow as network throughput is limited and latency is high.

5.) Shared Memory

Processes that wish to communicate with each other on a local system can do it efficiently and quickly through the use of shared memory. A file is created that is mapped into a shared segment amongst these processes and is locked down so that its not paged out. Now anything written in this shared segment (that is mapped to each processes virtual address space), once something edits this file the change is

propagated. This could cause issues however, as a single bug could break multiple processes and it can only be between processes on the same memory bus. There is also a lack of authentication with this method.

Note that data sent over network is done so through FIFO and so priority of data is not the method in which data is transferred. To fix this and make the data transmission based on importance, for local processes we could use **out-of-band –** Send a signal to invoke a registered handler and flush out buffered data to make priority room for more important data. This works because the signal travels over different channels than the data.
The same effect can be gained in networks by opening multiple communication channels. 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.

When it comes to using *socket()* we treat it just like any other regular file descriptor but with the ability to communicate through the network (or UNIX filesystem). We can use *write()* in order to send data or use the *send(), sendto()* calls as well which just adds the presence of flags to the data being sent. *Sendmsg()* does the same but with data as a message. So there is a message header struct that should be referenced and dealt with. Similarly there is a *read()* and corresponding *recv(), recvfrom()* and *recvmsg()* calls that only differentiate in the presence of flags. In both cases if there are messages that are too long to be sent over or no messages being received, they will block and wait until the right thing is done – unless set to non blocking mode.

Mmap() can be used to create a new mapping in the processes virtual address space, by specifying the start address, offset (multiple of the page size that can be retrieved using *sysconf(_SC_PAGE_SIZE)*), referred to by a file descriptor. The prot argument specifies protection of this space and the flags can be used to specify whether this memory mapping is private to the calling process or can be shared (using *msync()*).

OS Part II - Concurrency!

Each process has the ability of running programs in parallel and does so through the use of **threads**. Such a program is referred to as **multi-threaded**. Similar to processes, each thread is separate from each other but they all live in the **same address space** and so they can access the **same data**. They are given **separate registers** and so **context switches** must take place between threads and their register states are saved in a **Thread Control Block (TCB).** Each thread gets its **own stack** and thus the processes address space is broken into multiple stacks – which isn't a problem since stacks aren't generally large unless the program is a large recursive one.

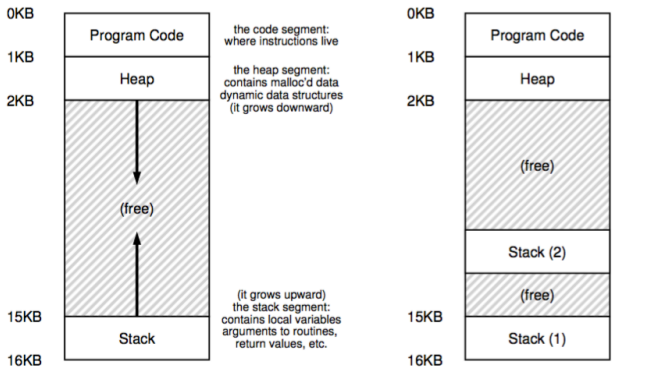


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

Stack allocated variables, parameters and return values that are generally stored on a stack can be referred to as **thread-local storage.**

When running threads, we still have a scheduler that schedules which thread to run and at what moment, similar to that of a process scheduler. The problem with schedulers running multiple things at once (such as two threads concurrently) is that their results are **non-deterministic** and thus cannot be predicted and can seem random and undefined.

On top of that because **threads share data** the results can be modified in seemingly unknown ways to produce non-deterministic results. Think of a situation wherein two threads are incrementing a variable. Essentially thread 1 will be running the code to increment the value of the variable, and before the value in memory can be updated a **timer interrupt** switches the context to the other thread and does the same thing, so that when both threads end up storing the result in memory, the value is the same – that is one increment overall instead of two. This is referred to as a **race condition** and is the result of **uncontrolled scheduling**.
Race Condition – When two threads act upon a section of code at the same time to produce **indeterminate results** due to concurrent access.
Critical Section – The piece/section of code that contains the possibility of running into a **race condition.**
Mutual Exclusion – The property that guarantees blocking off of a critical section to one thread at a time.
Atomicity – Principle of running "nothing or all" that is there is no interrupt anywhere in between this section. Grouping of multiple actions into a single atomic action is called a **transaction.**

Synchronization Primitives – Primers from the hardware in tandem with the OS that allows us to safely execute multi-threaded programs without race conditions.

More issues persist with a thread waiting on another thread to complete an action – such as I/O access – which requires this *sleeping/waking cycle*.

To create a POSIX thread we use *pthread_create()* and to wait for a thread to complete and grab its return value we use *pthread_join()*. Be careful when using variables in the functions that the threads use. Make sure the data used as a *return value is allocated on the heap and not the stack!*
Some programs such as web servers don't need to join threads, but in programs that accomplish a specific well defined task in parallel it is useful to join threads to make sure the following flow of execution is well defined.

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  typedef struct __myarg_t {
7      int a;
8      int b;
9  } myarg_t;
10
11 typedef struct __myret_t {
12     int x;
13     int y;
14 } myret_t;
15
16 void *mythread(void *arg) {
17     myarg_t *m = (myarg_t *) arg;
18     printf("%d %d\n", m->a, m->b);
19     myret_t *r = Malloc(sizeof(myret_t));
20     r->x = 1;
21     r->y = 2;
22     return (void *) r;
23 }
24
25 int
26 main(int argc, char *argv[]) {
27     int rc;
28     pthread_t p;
29     myret_t *m;
30
31     myarg_t args;
32     args.a = 10;
33     args.b = 20;
34     Pthread_create(&p, NULL, mythread, &args);
35     Pthread_join(p, (void **) &m);
36     printf("returned %d %d\n", m->x, m->y);
37     return 0;
38 }
```

Figure 27.2: **Waiting for Thread Completion**

When dealing with *critical sections* we wish to maintain mutual exclusion and this can be done through the use of *locks* and *condition variables*. Locks can be thought of as controlling mechanisms. Once a thread enters a locked section it gains the "key to the lock" and can run that critical section while other threads have to wait until the thread gives up the key allowing for another thread to unlock the critical section and continue. Condition variables are used if one thread in a locked area needs another thread to do something before continuing i.e. there is some thread dependency.

Locks come in the form of *pthread_mutex_t* and they must always be initialized to ensure proper behavior. It is best practice to initialize them dynamically using the *pthread_mutex_init(&lock, NULL)* method (returns 0 on success). Also destroy the mutex when done completely (*pthread_mutex_destroy()*). There are timed locks and tried locks too but are advised against in cases of *deadlock*.

Condition variables can be declared through *pthread_cond_wait(&cond, &lock)* or *pthread_cond_signal(&cond)*.

Research has shown using while loops in waiting for threads is most effective as it is safer in thread waking. Also don't use flags to synchronize between threads and us condition variables instead.

Now the reasons threads came about was for parallelization, in-memory data communication (before memory sharing) in a lighter weight context. We can think of the following abstraction:

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

Earlier threads used to be run in user mode libraries in the following manner:
Allocate memory for fixed size thread private stack in heap => create thread descriptor that holds information => add new thread to ready queue => save general thread registers upon *yield()* or *sleep()* and select next thread => dispatch and restore thread registers from return from yield => upon *sleep()* remove from ready queue (add back in reawakening) => upon exit free stack and thread descriptor.

With *preemptive scheduling* the Linux system brought about SIGALARM to interrupt long running threads.

Problems with threads in user-mode library:

1.) System call blocks – upon I/O access the process blocks and so do all the other threads in the process, and since OS doesn't know about them it cannot know about the runnable state of threads.
2.) Exploiting multi-processors – If OS is not aware of the threads then it cannot parallelize them amongst multiple CPU's.

Non-preemptive scheduling – Faster with user mode yield/sleep method than OS based threads.
Preemptive scheduling - 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.
Multi-core – threads can run in parallel and so 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.

SOME NOTES FROM CLASS:

Ack-Window – Acknowledgement window, this is the amount of data you can get ahead of in your IPC sending and receiving. After getting ahead of this window, we have to let the flow control take over.

Wednesday:

Lock – A variable that holds the state of the lock at any given time. Either **available** or **acquired**, by exactly one thread and generally in a critical section.
If no thread currently holds the lock, then the thread that gets to the lock first shall become the *owner* of it. Other threads will continue waiting on this lock until it is given up.

Coarse-grained – Using one large lock for a critical section.
Fine-grained – Using multiple locks around for different critical sections.

How do we build locks? A lot of support comes from hardware and the OS and is typically specific to that ISA, but the concepts remain the same. Some of the goals we try to achieve when creating locks are as follows:

- Correctness/Efficiency – Does the lock do the right thing while maintaining correct behavior of the critical section.
- Fairness – Does each thread get a fair chance at acquiring the lock once freed. (Do the threads *starve* or not).
- Progress – Does the locking mechanism create multiple dependencies, convoy formation or deadlock.
- Performance – How does the mechanism hold with multi-processors, convoy formation, uni-processors.

Method 1 – Disable Interrupts

Pros:

This approach is simple. Simply disable interrupts for the critical section when locking and re-enabling these interrupts when unlocked. This would require some special hardware instruction, but is simple.

Cons:

Requires calling thread to perform *privileged instruction* and thus place a sense of trust. If the locking mechanism is user mode created and is trusted to turn interrupts on and off, then one could game this mechanism and keep calling lock to ruin fairness. This method could also slow down progress by getting stuck in a loop within the lock.

This does not work on multi-processors since the interrupts are processor specific in that it will block for that specific processor, and so threads from other CPU's could still come.

Interrupts can get lost which can mess up OS scheduling.

Masking and unmasking interrupts (i.e. turning them on and off) is an expensive operation and can be very slow.

Method 2 – Spin Lock

Test and Set (Atomic Exchange):

With the support of hardware there is an *atomic instruction* known as *test-and-set instruction*. The basic idea is to hold a universal flag that a thread entering a critical section will first *test* to see if the flag is free (0) and if it is, it will *set* the flag to busy (1) and that thread will now hold this flag that is the log. The test and set atomic instruction returns the old value of the flag. This flag will remain held (keep returning 1) until a thread finishes and changes it to 0.

Spin-wait – Another thread tries to enter the critical section but realizes that its currently held, and so will sit there and spin (wasting CPU time and resource) until its freed up, or on a multi-core processor, it will try to find another CPU to run on.

Issues:

This testing and setting could be interleaved and be interrupted while changing the flag! This can then set the flag to 1 by two threads and thus breaks the principle of *mutual exclusion*.

The performance issue comes from the spin waiting, which will waste resources.

This *spin-waiting* allows us to build something known as a **Spin Lock**. The fact that this instruction is atomic means that only one threads can execute it at time making it a mutual exclusion primitive. On a uniprocessor system, there must exist a preemptive scheduler to relinquish threads to prevent infinite spinning.

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available, 1 that it is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

Pros:

This spin lock is easy to build with test and set. It is correct in that it accurately provides mutual exclusion.

Cons:

This is not fair as we can starve different threads by continuously making them spin. Performance wise they are pretty horrible too for long operations because they will waste CPU cycles in the continuous spinning. Note that *spin-locks* work *reasonably well* on *multi-processor systems*, and for *short critical sections*.

Compare-And-Swap:

This is another hardware primitive with the atomic instruction known as **compare-and-swap/exchange**. Essentially we test whether the value specified by some pointer is equal to an *expected* value. If not, then we do nothing, else we update the location pointer to by the pointer to the third argument of this function which is the *new* value. In both cases we return the value at the memory location which allows for this comparison.

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int actual = *ptr;
3      if (actual == expected)
4          *ptr = new;
5      return actual;
6  }
```

Figure 28.4: **Compare-and-swap**

While this instruction **can make spin-locks** they are **more powerful** because they can make use of **wait-free synchronization.**

Load-Linked and Store-Conditional:

```
1  int LoadLinked(int *ptr) {
2      return *ptr;
3  }
4
5  int StoreConditional(int *ptr, int value) {
6      if (no one has updated *ptr since the LoadLinked to this address) {
7          *ptr = value;
8          return 1; // success!
9      } else {
10         return 0; // failed to update
11     }
12 }
```

Figure 28.5: **Load-linked And Store-conditional**

then the store-conditional will only succeed and *update the value* when there is no *intervening store* has taken place. The conditional will return 1 and update the location to *value*.

Yet another hardware primitive that works on the following basis. We load the value from the specified memory location into a register,

```
1  void lock(lock_t *lock) {
2      while (1) {
3          while (LoadLinked(&lock->flag) == 1)
4              ; // spin until it's zero
5          if (StoreConditional(&lock->flag, 1) == 1)
6              return; // if set-it-to-1 was a success: all done
7                      // otherwise: try it all over again
8      }
9  }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

Figure 28.6: **Using LL/SC To Build A Lock**

Cons:

While executing the load-linking one thread is returning 0 as the lock is not held, but before store-conditional it is interrupted and another thread load-links and returns 0 too and so we get a deadlock at the point of the *store-condition*. This avoids the *mutual exclusion* correctness criteria.

Method 2.1 – Ticket Lock

Fetch-And-Add:

The final hardware primitive mentioned in the class is the *fetch-and-add* instruction. Instead of using a singular flag value what the **ticket lock** will do is first do an atomic fetch-and-add on a global *turn* variable which will determine which thread's turn it is to enter the critical section. Note that this is still effectively a **spin-lock** as threads spin until their turn, but there is no way of a thread running forever unlike the test-and-set method.

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15  }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```

Figure 28.7: **Ticket Locks**

Pros:

This is a very fair system and avoids deadlock as it gives a fair chance to every thread to run.

The issues mentioned with spinning above mainly focus on wasting CPU time and resource which is extremely precious. To fix this we need more than just *hardware-primitives*, we need OS support.

Method 3 – Yield

Because spin-locking has performance drawbacks and can result in **starvation** (other than in ticket lock) we need a way to solve this issue. The OS allows for a thread to **yield** itself from the **running** state to the **ready** state, while promoting another thread. It allows for the thread to **deschedule** itself. However, while this is **better than regular spin-locking** because it **doesn't waste a majority of the time slice** in the scheduling of the process, it still incurs *large overheads* due to the context switches. Also this could still result in *starvation* through **infinite yield** where threads simply enter and exit the critical section.

```
1  void init() {
2      flag = 0;
3  }
4
5  void lock() {
6      while (TestAndSet(&flag, 1) == 1)
7          yield(); // give up the CPU
8  }
9
10 void unlock() {
11     flag = 0;
12 }
```

Figure 28.8: **Lock With Test-and-set And Yield**

Method 4 – Queues and Sleeping

In the previous examples we let it up to the **scheduler to make the decision** of what thread to run, but this could result in bad choices being made. Instead we keep a **queue of waiting threads** that are put to **sleep** until woken up during the lock release. *Park()* and *unpark()* instructions put a thread to sleep and wakeup in Solaris systems. In this method we have a guard variable that allows for thread control flow (allowing threads to enter sequentially) while a flag holds the lock status. If locked, the next thread is added to a queue and then put to sleep until the lock is freed. Spin-waiting may still occur if an interruption happens while *acquiring* or *releasing* the locks, but is a lot shorter.

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

Wakeup/waiting race – A switch takes place before a thread is put to sleep and another thread releases a lock, which could result in an *infinite sleep*.

Solution 1: Use *setpark()* which indicates a thread is about to sleep. If it happens to be interrupted and another thread calls unpark before park, the subsequent park returns immediately instead of sleeping. Just setpark before setting guard.

Solution 2: Pass the guard to the *kernel* which will take precautions to atomically release the lock and dequeue the running thread.

The above idea is similar to the concept of a **futex** in the Linux system. The difference lies in that the futex provides more in-kernel functionality. Each futex has an associated physical memory location and an in-kernel queue per futex. Callers can use futex calls to sleep and wake (futex_wait and futex_wake)

Two-Phase Locks – First phase the lock will spin waiting to be acquired. If not successful it will enter phase two where it will be put to sleep and woken up when the lock becomes free later.

That is use spinning only once (or a small fixed number of times) before using futex to sleep. This is a **hybrid** approach.

WEEK 5

Monday:

When talking about concurrency, we wish to see how it is applied to data structures to create **thread-safe** data structures. The aim is also to make these concurrent data structure **scalable**.

1.) **Counters**

Counters are the simplest data structures in that all they do is add or decrement the value and return this value. One could make it *concurrent* by adding locks around every operation and unlocking it after every operation. This will ensure correctness but it will severely reduce the performance. Thus we have to be able to come up with a correct and effective solution.

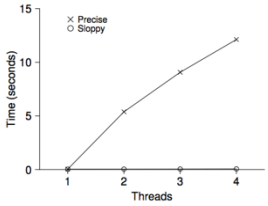


Figure 29.3: **Performance of Traditional vs. Sloppy Counters**

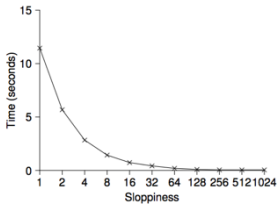


Figure 29.6: **Scaling Sloppy Counters**

Sloppy Counters – Each CPU will have its own 'local' counter, and there will exist one 'global' counter. Similarly, each local counter will have its own lock and the global counter will have its own lock too. We update each local counter *safely* when needed until we hit a threshold – **sloppiness**. The higher **sloppiness** is the *faster the counter* but the *less accurate* it is. This allows us to parallelize the addition which is what makes it fast, and once a local counter hits this maximum, we transfer it to the sum of the global counter and reset the local counter back to 0.

```c
typedef struct __counter_t {
    int             global;          // global count
    pthread_mutex_t glock;           // global lock
    int             local[NUMCPUS];  // local count (per cpu)
    pthread_mutex_t llock[NUMCPUS];  // ... and locks
    int             threshold;       // update frequency
} counter_t;

// init: record threshold, init locks, init values
//       of all local counts and global count
void init(counter_t *c, int threshold) {
    c->threshold = threshold;
    c->global = 0;
    pthread_mutex_init(&c->glock, NULL);
    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;
        pthread_mutex_init(&c->llock[i], NULL);
    }
}

// update: usually, just grab local lock and update local amount
//         once local count has risen by 'threshold', grab global
//         lock and transfer local values to it
void update(counter_t *c, int threadID, int amt) {
    int cpu = threadID % NUMCPUS;
    pthread_mutex_lock(&c->llock[cpu]);
    c->local[cpu] += amt;                // assumes amt > 0
    if (c->local[cpu] >= c->threshold) {  // transfer to global
        pthread_mutex_lock(&c->glock);
        c->global += c->local[cpu];
        pthread_mutex_unlock(&c->glock);
        c->local[cpu] = 0;
    }
    pthread_mutex_unlock(&c->llock[cpu]);
}

// get: just return global amount (which may not be perfect)
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val; // only approximate!
}
```

Figure 29.5: **Sloppy Counter Implementation**

2.) **Concurrent Linked Lists**

```c
void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

void List_Insert(list_t *L, int key) {
    // synchronization not needed
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return;
    }
    new->key = key;

    // just lock critical section
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
}

int List_Lookup(list_t *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv; // now both success and failure
}
```

Figure 29.8: **Concurrent Linked List: Rewritten**

The issue that comes with locking operations with linked lists is the failure of malloc wherein the code must release the lock before failing the insert. This can be a pretty big issue on systems. Thus it is important to place the lock only around the critical sections and assume *malloc()* is thread safe. As for scalability, there is a specific linked-list that aims to see if the

speedup with concurrent scalability is something to be concerned about.

Hand-over-hand Locking (lock coupling) – Add a lock per node of the list. Upon traversal, grab the next code's lock, and release the current node's lock.

It is found however, that there is significant overhead incurred on such a mechanism, and the speeds between this method and a single lock around the entire list is not really different. Perhaps a hybrid approach could be the best solution.

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;

typedef struct __queue_t {
    node_t             *head;
    node_t             *tail;
    pthread_mutex_t     headLock;
    pthread_mutex_t     tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}

void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
```

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++) {
        List_Init(&H->lists[i]);
    }
}

int Hash_Insert(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Lookup(&H->lists[bucket], key);
}
```

Figure 29.10: **A Concurrent Hash Table**

3.) **Queues**

Adding one big lock around the queue could prove to be slow. Thus we take *Michael and Scott's queue*. This involves adding a lock only to the head and tail of the queue for dequeue and enqueue operations respectively. There is the use of a dummy node for the separation between the head and tail of the queue. This queue isn't the perfect queue for full use in multi-threaded applications. There is more to come on the topic of queues.

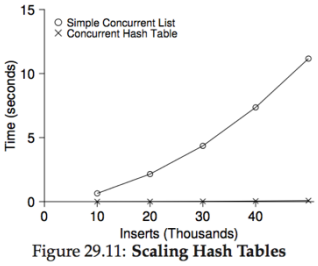


Figure 29.11: **Scaling Hash Tables**

4.) **Concurrent Hash Tables**

To make a hash table concurrent we just use *concurrent linked lists* as mentioned before, where each bucket of the hash table is a concurrent linked list. This way instead of having one massive lock around the entire data structure, there is a lock around each bucket which makes it more scalable.

Knuth's Law – Premature optimization (trying to optimize before even building a working model) is the root of all evil. That is to say, we start by wrapping our data structure in one big lock to ensure correctness, and then from there we optimize if need be. As an example is the **Big kernel Lock** that applied one big lock around the kernel in early Linux and Sun OS. This worked brilliantly until multi-processors, which is when we had to transition into further optimization.

Non-blocking data structures – Another technique to achieve concurrency without the use of traditional locks.

While locks are great in ensuring concurrency, we must consider the idea of finishing a thread and knowing how to signal to the creator that said thread is done. Such is an example of the

need for threads to check for a **condition.** A naïve solution would be to use a **shared (global) variable** to indicate a certain status, but this involves making the 'checker' thread spin which wastes CPU cycles.

Condition Variable – Explicit queue that threads can put themselves on when some state of execution (condition) is not as desired (waiting on said condition). When another thread changes said state, it can then wake the waiting thread and allow them to continue (through signaling).

In POSIX we have *pthread_cond_t* with two key functions.

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

Wait will put the checker thread to sleep (thus not wasting any cycles) and the signal will wake the sleeping thread upon completion of the condition. Wait requires a *mutex* because it assumes this is locked when wait is called and it is the responsibility of wait to release the lock and then put the thread to sleep. When woken up, the thread must reacquire the lock before returning to the caller. These instructions are done atomically, but it is always important to remember the following rule as to avoid permanent sleeping or waiting due to race conditions: **Hold the lock when calling the signal or waiting.** We thus consider a few popular examples of condition variables:

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

1.) **Producer/Consumer (Bounded Buffer) Problem**

We have producers that generate some data and place them into a buffer and we have consumers that grab said items form this buffer and do something with them. This is a real world problem with HTTP requests in working queues, and even in shell pipelining with kernel bounded buffers. Solving such a situation has two common pitfalls:

a.) Using *if* statements as our measure of checking for the use of condition variables. The problem is that another consumer can come and sneak and grab the data which is when the first consumer will try grab some data, finds that there is none and will thus produce wrong results. The **solution** is to **replace if statement with while loops**, so that consumers and producers are always checking for the state after being awoken and not just doing a one of check. Always use while loops with condition variables.

Mesa semantics – Signal *hints* to a change of state but does not *guarantee* the state remains changed upon your waking up.

Hoare semantics – Harder to build but provides a stronger guarantee that the woken thread will immediately run upon being woken (i.e. guaranteeing the state remains changed when you wake up).

b.) Consumer can wake another consumer or a producer can wake another producer which can result in everyone falling asleep at some point in the program which leads to a stall. Example: Producer puts value in buffer -> first consumer wakes and grabs value and by some off-chance wakes up the second consumer instead of the producer -> second consumer sees buffer as emptied and goes back to sleep -> producer remains asleep and first consumer is asleep. Everything is asleep! The **solution** is to have **two separate condition variables – 1 for consumers and 1 for producers**.

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Figure 30.11: The Final Put And Get Routines

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        while (count == MAX)                   // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                // p4
        Pthread_cond_signal(&fill);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            Pthread_cond_wait(&fill, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&empty);           // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Figure 30.12: The Final Working Solution

Spurious Wakeups – Two threads get woken up by a single signal. To prevent any issues with such an incident, always use while loops around conditional checks.

2.) **Covering Conditions**

When dealing with an example such as building a memory/resource allocator, we can imagine the situation wherein we have a couple of threads that require a certain amount of memory. These threads both go to sleep until memory/resource is freed. A third thread frees some resource but instead wakes up a thread that requires more resource than offered, while the second thread that could have benefited from such a free would remain sleeping. This way we have a waste of resource allocation. The **solution** was to use a *pthread_cond_broadcast()* that will wake up all threads (this could be expensive, so use it wisely) such that the relevant thread will acquire what it needs to while the other threads will go back to sleep. Possibility of race conditions with such a broadcast?

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t  c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...; // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c); // whom to signal??
    Pthread_mutex_unlock(&m);
}
```

Figure 30.13: Covering Conditions: An Example

Now that we know of both thread locks and condition variable, along with the hardware primitives discussed earlier, we may think we have a complete understanding of concurrency in modern day systems, but rather we need to discuss one final primitive – **Semaphore** which is somewhat the combination of locks and condition variables.

Semaphore – Single primitive for all things synchronization. It is an object with an integer value that can be manipulated with *wait* and *post*. They can be initialized to work amongst several processes (second argument) or focus on the threads within a single process. They must be initialized with some integer value.

Waiting will decrement the value the semaphore by 1 and will wait if it is negative. Post will increment by 1 and wake a thread if it is waiting. **When the semaphore is negative its value is equal to the number of waiting threads.**

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit sect: begin)	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem<0)→sleep	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	(crit sect: end)	Running		Sleeping
-1	call sem_post()	Running		Sleeping
0	increment sem	Running		Sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	sem_wait() returns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

Binary Semaphores

A binary semaphore is essentially a lock around the critical section. That is put a sem_wait() before and sem_post after the critical section. We can look at an example of a double thread trace using such an instantiation of a semaphore. It is called as such because of the lock only having two states – held and not held. It may be simpler to stick to a generic lock if this is the only reason you're using a semaphore. **Initial value of semaphore should be 1.**

```
1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

Figure 31.6: **A Parent Waiting For Its Child**

Semaphores as Condition Variables

We can take the parent-child example into consideration in such a situation. We see that we just call a post within the child thread and wait in the parent thread right after the creation. **Initial value of semaphore should be 0.**

Value	Parent	State	Child	State
0	create(Child)	Running	*(Child exists; is runnable)*	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem<0)→sleep	Sleeping		Ready
-1	*Switch→Child*	Sleeping	child runs	Running
-1		Sleeping	call sem_post()	Running
0		Sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	sem_post() returns	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	sem_wait() returns	Running		Ready

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

Value	Parent	State	Child	State
0	create(Child)	Running	*(Child exists; is runnable)*	Ready
0	*Interrupt; Switch→Child*	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	*Interrupt; Switch→Parent*	Ready
1	call sem_wait()	Running		Ready
0	decrement sem	Running		Ready
0	(sem≥0)→awake	Running		Ready
0	sem_wait() returns	Running		Ready

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)**

As is with condition variables, we can approach the bounded buffer problem with semaphores too. Because this is a concurrency problem we must recognize the potential of race conditions and must thus use **3** semaphores – **mutex, full (set to 0) and empty (set to MAX buffer size).** The reason empty is set to the max buffer size while full is set to 0, is the way in which they are called. We see that the consumer first calls wait on full, which should then hand control over to the producer who will start by waiting on the empty semaphore, but since this is MAX, it will continue and post in full which will then wake up the consumer to do its business. The mutex **must** be added after the call to wait on the empty and full semaphores before the locking begins so as to avoid a **deadlock –** Threads are left waiting on each other to finish resulting in an infinite loop.

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&empty);          // line p1
9          sem_wait(&mutex);          // line p1.5 (MOVED MUTEX HERE...)
10         put(i);                    // line p2
11         sem_post(&mutex);          // line p2.5 (... AND HERE)
12         sem_post(&full);           // line p3
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full);           // line c1
20         sem_wait(&mutex);          // line c1.5 (MOVED MUTEX HERE...)
21         int tmp = get();           // line c2
22         sem_post(&mutex);          // line c2.5 (... AND HERE)
23         sem_post(&empty);          // line c3
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33     // ...
34 }
```

Figure 31.12: **Adding Mutual Exclusion (Correctly)**

Reader-Writer Locks

This is a more flexible locking primitive that focuses on allowing *many readers* but only *one writer*. If we want to insert we acquire the write lock and release the write lock when done. When acquiring the read lock however, the reader first needs to acquire the generic lock and increment a reader variable to know how many readers exist and *then* allow for the writing. Essentially what this allows for is multiple readers to acquire the read lock, but only one to acquire the write lock, while the other will have to wait.

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // used to allow ONE writer or MANY readers
  int   readers;   // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
    sem_wait(&rw->writelock); // first reader acquires writelock
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
    sem_post(&rw->writelock); // last reader releases writelock
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Figure 31.13: **A Simple Reader-Writer Lock**

The problem with such a primitive is that it is **unfair** since it could potentially starve writers. They must also be used with caution because they incur serious overhead and thus don't necessarily speedup the performance.

Dining Philosophers

This is a famous problem that involves **n** philosophers sitting next to each other on a round table with **n** forks in total, one on the left and one on the right of each philosopher. Each philosopher will *think, getfork, eat, putfork*. The aim is to effectively synchronize this process by focusing on how to get and put fork. A naïve solution would be to grab a lock when *getfork* and release it on *putlock* but if we have the condition where all philosophers try to getforks on their left before they getforks on their right, we achieve *deadlock* which can result in starvation of the poor old philosophers.

The solution is to break the dependency caused by the circular nature of the problem. That is when the *n-1*th philosopher tries to grab a fork, just make him grab the fork (i.e. grab the lock) by getting left then right *instead of right then left*.

```
void getforks() {
  if (p == 4) {
    sem_wait(forks[right(p)]);
    sem_wait(forks[left(p)]);
  } else {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
  }
}
```

Examples of similar problems – **cigarette smoker's problem, sleeping barber problem.**

Semaphores can be built out of the locking and condition variable primitives. They do not hold the invariant that negative value of semaphores represents the waiting threads, but rather modern Linux based implementation never even reach negative semaphore values. The converse of building locks and condition variables out of semaphores is very difficult.

```
1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }
```

Figure 31.16: **Implementing Zemaphores With Locks And CVs**

WEEK 6

Monday:

Concurrency bugs come in two flavors – **non-deadlock bugs** and **deadlock bugs**. For the most part, the former makes up more of the bugs than the latter.

Non-Deadlock Bugs

These bugs come about in two variations as well which can be described as:

1.) **Atomicity-Violation Bugs:**
The desired serializability among multiple memory accesses is violated in the sense that we assume an operation is atomic but it isn't and as such we don't enforce the atomicity during the execution. An example is checking to see if a pointer is not null and then execute an instruction but then get a context switch wherein the pointer is made null and so now we try to *dereference a NULL pointer.* Naïve but obvious solution would be to take note of this and make sure it doesn't happen ever again, by preventing this preemption, by maybe using a mutex for a lock.

2.) **Order-Violation Bugs:**
The desired order between two groups of memory access is *flipped* (that is *A* should be run before *B* but rather the order is not enforced during the execution. An example would be setting a thread equal to another before actually knowing what hat thread is. A simple *solution* is to *enforce ordering* by using **condition variables**,

Deadlock Bugs

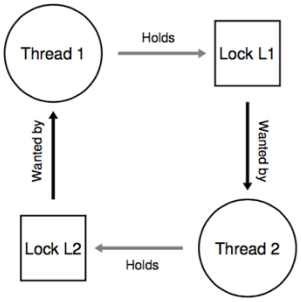


Figure 32.2: **The Deadlock Dependency Graph**

This occurs when one thread holds a lock waiting for another, but the other thread is holding another lock in anticipation of the first lock being freed. As such the execution enters this infinite loop of dependency that prevents furthering of the program. We have achieved *deadlock* in this manner. *Encapsulation* and *modularity* can be a huge contributing factor to such errors. Deadlock must involve one thread being blocked and waiting for the other or both threads being blocked. This is what differentiates it from other dependency based concurrency bugs.

There are essentially **4 fundamental** ways of reaching a deadlock:

1.) **Mutual Exclusion:** Threads claim exclusive control of resources that they requires such as grabbing a lock.
2.) **Hold-and-wait:** Threads hold resources allocated to them such as locks that are already acquired, while waiting for additional resources such as locks they wish to acquire.
3.) **No preemption -** Resources cannot be forcibly removed from threads that are holding them.
4.) **Circular wait –** There exists a circular chain of threads such that each thread holds one or more resources that are being requested by the next thread in the chain.

If any of these four are not met, then *we don't have deadlock.*

Prevention:

1.) **Circular Wait:**

Apply either a *total ordering* (for small number of threads/locks) or a *partial ordering* to enforce an ordering to the locks for example which in turn prevents this wait cycle caused by the continuous circular dependencies.

2.) **Hold-and- wait:**

Try and acquire all locks at once in a single atomic instruction. We can use a global lock within which we lock/unlock all other locks. The *problem* with this is that it requires us to know about which locks exist and need to be acquired before we even run, and by doing this locking so early on we essentially decrease the concurrency.

3.) **Preemption**

```
top:
  lock(L1);
  if (trylock(L2) == -1) {
    unlock(L1);
    goto top;
  }
```

A lot of languages add support for different kinds of lock acquiring mechanisms such as *trylock* which will try to grab the lock if its available, else return -1. This would essentially solve deadlock problems, but give arise to a new problem (though quite rarely) – **livelock.**

Livelock – Two threads attempt to trylock at the same time and both fail to acquire a lock. Thus the threads run through this sequence of code again and again.

4.) Mutual Exclusion

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  do {
    n->next = head;
  } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

We wish to avoid the need for mutual exclusion which is in general pretty difficult because of the presence of critical sections. Use **wait-free data structures** using *hardware primitives* so that the instructions themselves are atomic already. Such as using compare and swap for incrementing values or inserting to the head of a list. This is obviously much more difficult to do and could still fail. In the example of inserting a node into the Linked List, we could have failure if another thread swapped in a new head which would make the current thread try again with the new head. They also lack generality.

Deadlock avoidance using Scheduling

Instead of preventing deadlock, it may be more beneficial to *avoid* them completely. This requires knowledge about the types of locks that will be grabbed by different threads and then schedules these threads ins a way to guarantee no deadlock. Essentially through scheduling we let threads run in such a manner such that two threads that contend for a resource are never run at the same time. This can lead to a significant *cost to performance* but can prevent deadlock. However, this is not a widely-used general purpose solution.

Detect and Recover

Another horrible strategy to handle deadlocks is to actually *allow them to happen* and then try to *take some action once they've been detected*. A deadlock detector runs periodically, building a graph of resources and trying to find any **cycles**. In such an event, the detector simply recommends shutting down. More complex detectors may have human help to help inform the user/program on what to do. But this relies on the fact that the *deadlock occurs very rarely* which may not be the case for your specific application.

Hopefully this shows how pesky locks can be. Some companies/programs have taken to implementing completely new concurrent programming models such as *MapReduce* by *Google*.

We now wish to analyze deadlock avoidance in a little more depth with it pertaining to *exhaustion of a critical resource.* 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.

Reservations:

While declining requests works in preventing resource exhaustion deadlocks, handling failure of allocation mid operation maybe more difficult to handle gracefully. As such it is common to ask a process to *reserve their resources before they actually need them*. Such as the *sbrk* call which just asks the OS to expand the data segment in the virtual address space for the process, and only does the actual memory assignment when needed. This allows us to decide how to handle the case where we can no longer allocate resources, whereas if we didn't do this, sbrk would just kill the process when it required more memory allocation during the operation.

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:

It is relatively safe to grant more reservations than we actually have resources to fulfill. This way we can get more work done with the same resources, but this then puts us in danger of not being able to gracefully handle a demand. Think of airlines and how they manage bookings. Some OS's the notion of killing processes is just terrifying to the extent that there may be *under-booking*.

A process has many available options on how to handle rejection. It could just throw an error message and exit; it may keep trying the request until it succeeds; it could throw errors and continue trying the request; or it could try to reduce its resource consumption. The key is that through this *deadlock avoidance* we give the process the chance to decide on how to exit, rather than kill it by default.

As we mentioned before a possible way of dealing with deadlocks, is to allow it to occur and then basically talk about doing a system restart. This was a fucking horrible effort at solving deadlock, and so we look to a better alternative, that tends to be more general and a lot more informative.

Health Monitoring and Managed Recovery

Firstly, how do we determine if a system is in a deadlocked state?

- Identify all 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 graph contains any cycles

However, for the most part it is difficult to do the aforementioned things, and even if we could, how do we deal with this deadlock? 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 having the service send periodic heart-beat messages to a 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.
- 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.

Now that we know how to tell if a service/process has entered deadlock/hung, how do we deal with it? Well our services should be designed for effective restart and recovery.

- The software should be designed such that upon a system shutdown, kill or restart, the process can get back to the state it left of in with minimal disruption.
- Designed to handle multiple levels of restart:
    - Warm-start – Restore the last saved state and resume service from where we left off.
    - Cold-start – Ignore saved state as it may be corrupted and restart new operations from the ground up.

- If process A is killed due to a failure by process B, upon A's restart it will try connecting with B again and the problem will persist unless we design our software for progressively escalating scopes of restarts:
    - Restart only a single process and expect it to resync with other processes when it comes back up.
    - Restart all of the software on a single node.
    - Restart a group of nodes or the entire system.

The problem with these notifications and restarts is that sometimes we don't know what actually failed. Was it the node or the process or a delay in the heart-beat due to a network error? Thus it is not always wise to immediately declare a failure.

- the best option would be for a failing process 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.
- in some cases, we might want to wait for multiple other processes/nodes to complain.

Once again there is a tradeoff here between how much time to wait – too long and we prolong the service outage; too short and we may suffer unnecessary service disruptions form forcing fail-overs from healthy servers.

While considering failure and restart, it is wise to take note of two other interesting cases:

1.) Non-disruptive rolling upgrades – If a system can operate without some nodes, we can essentially take these nodes down, upgrade them and reintegrate them one at a time. We must make sure this new version is upwards compatible so nodes can still communicate and that there exist fallbacks in case new nodes fail.
2.) Prophylactic Reboots – Software becomes slower the longer it runs (usually due to memory leaks). If we don't fix the bug, we can instead schedule automatic restarts at regular intervals of the entire system.

If any of the text above talks about anything, it is that synchronization is difficult. It is hard to find the critical sections, choose an effective and valid mutual exclusion method and implementing a valid strategy. We effectively need some sort of "magic bullet" that satisfies all of this. As such multiple mechanisms were created to try and achieve this.

Monitors – Thread-safe class or objects that use mutual exclusion to safely access member variables or methods safely by more than one thread, by allowing only one thread at a time. Just as mentioned before, monitors are just another locking mechanism that aim at solving the issues with deadlock. They are good because they are fair, using semaphores to implement waiting queues while assuring mutual exclusion. They don't completely solve deadlock however, because we could have *inter-class dependencies* which could lead to deadlock, and they are *coarse grained* which is not the most effective.

Java Synchronized Methods – This is different to monitors in the sense that it provides mutual exclusion to a specific method that gets instantiated only for that object block. Thus the

locking mechanism is finer grained and therefore performs a lot better, but at the same time does well with mutual exclusion. To invoke this in java we define a function with *synchronized* keyword. When the single thread has executed this synchronized method, it will update all other threads on this new change, so that everything remains consistent. This is great if the developer can correctly identify the critical methods. This is safe form single thread deadlocks for sure, but the downside is that because it uses priority thread scheduling, it can potentially starve.

Encapsulated Locking – Similar to Java Synchronization *Statements* - Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock - in that each object has internal intrinsic locks that synchronization statements can use to opaquely encapsulate implementation details. It prevents the interaction with external resources, and makes the life of the client using it a lot easier.

Wednesday:

When dealing with OS performance, we wish to measure how well our system or application or software is doing. Hardware improves with every year and the software must do so to, in the same extent to achieve the best possible results for the user. But when getting onto the topic of performance and measuring it, we must first come to grips with the goals of our software. What is it trying to achieve, where should it be at its fastest, what *metric* would be good for measuring this and do I have the ability to measure said metric? All these questions aid in analyzing the performance of software.

When dealing with measuring performance it must be noted that your measurement tool could get in the way of accurately measuring the metrics. If the experimental framework interferes with the processes one is trying to measure, then one could end up with a false reading for the metric.

After understanding what metric is of importance and how to measure its performance with the given tools, we must come to understand that large systems unbelievably complex. Take the OS for example, we have software for caching, hardware for caching, scheduling policies, synchronizations, interrupts and more. This means that repeated experiments are going to yield slightly varying values. Thus we take repeated readings and use mechanisms from the field of statistics to help quantify this data into valuable readings.

Mean is good for getting the cumulative reading for the data, the median is good for seeing where things are centered, mode is good for finding the most probable results, ranges are necessary to set bounds, standard deviation is helpful for describing the most commonly occurring range of values around the mean, and confidence intervals describe the probability that a particular measurement is within a certain range.

Effectively gathering this data requires a mastery of the experimental frameworks used for measuring the metrics.

Sometimes we can have alternatives in our analysis that we call factors. Varying these factors in different ways is what gives us a large range of results, from which we compare and decide. However, as the number of available factors increases, so does the number of experiments that would have to be run as a combinatorial explosion. It is important to strike the fine balance between which variation of factors is important and which ones are unnecessary to measure. Another element to consider is how many runs of each alternative are run. The more runs the better your statistical results but the longer you spend measuring. When measuring alternatives, we must try to be fair not for moral reasons but because fairness plays an important role in caching data which can thus affect the performance. Running the experiments over and over again while randomly interspersing which process to run will generally wash this caching out. When measuring the performance of alternatives, it is also important to consider system reboots and isolation from various external factors, to improve the fairness.

When you know that there exists a performance problem, it important to ask yourself, why is the performance bad, so that you can gear your experiments to explicitly test for the bottleneck. Take the best knowledge you can easily obtain about your code and the observed performance problem and generate a hypothesis about why it's happening. Run experiments to prove or disprove said hypothesis, and then iterate on the results from that. This entire process is very similar to finding bugs in a program but just over a wider range.

Workloads are important factors to consider when measuring performance. Sometimes you tailor your workload to test the performance of a very specific case, or in a more general measurement you may have to generate realistic workloads. The types of workloads are listed below:

1.) Traces – Gather a detailed trace of the workload of the system in its ordinary activities. Depends on nature of what you're testing. Save it in a form for recreation. This type provides realism and reproducibility. It is *not* easily reconfigurable. Scaling traces are generally hard. Good traces are hard to come by, most traces not produced by yourself will most likely be very old. May be hard to gather information needed to generate the trace. A single trace may or may not be an accurate representation of the activity. Systems are complex and can could have just had unusual activity during the trace. Privacy implications could affect saving traces.
2.) Live workloads – Perform measurements on a working system while it goes about its normal activities. This is obviously very realistic. If you can do this for a long time, you can get a wide range of real system behavior. However, this has *lack of control* as we can't reproduce the behavior seen, and we can't scale. Experimental framework will have some small impact on the performance and functionality because it is live -> can't run the measurement for too long on a working system as it could get in its way of doing its actual job. Possible privacy implications as well.

3.) **Standard benchmarks –** Sets of programs or data that are used as to drive the experiments. The benchmarks may have been derived from traces or models. Easy integration by any developer, and easy comparison to other systems as they are standardized. Generally gives a holistic set of data about your system. Widely used benchmarks are guaranteed to be robust as they have been the subject of a lot of study. Some are even built to be inherently scalable, and we have **no privacy implications**. However, there are a *limited number of them*. They may not contain information about your specific workload space, and so it may not fit to your specifications. Because they are intended for long lasting behaviors, they may represent *old workloads*.
4.) **Simulated workloads –** Build models of the loads you are interested in. Usually parameterized for flexibility with scaling and parameter variation. Easily customizable to many different scenarios and possibilities. If there is no randomization involved then they are infinitely repeatable. Also no privacy implications. However, validity of results is only as good as the quality of the models. Not easy to produce good models for complex systems. Parameters may prove unrealistic in certain cases. Hard to conjecture what the parameters should be to mimic realism. If the parameters are incorrectly set, then we get a false picture.

There are some common mistakes to watch out for when running experiments. It is up to you as the experimenter to clearly understand your goals, metrics, factors and workloads and write out a good design for performance measurement. Some specific mistakes are as follows:

1.) Measuring *latency without utilization*. One must emulate a realistic situation. Trying to measure latency on an operation when nothing else is running is pointless. Latency should be measured with some characteristic background load or when we have heavy load.
2.) Not reporting the *variability of measurements.* Tyr to always list out all the statistical data you have gathered, how this impacted your analysis of the system etc. Be more descriptive.
3.) *Ignoring important special cases.* You may either ignore a special case's ability to affect the measurement, or you fail to consider the edge case. The most common of the first is ignoring *startup effects* – caching is ubiquitous and you should be wary that the start will incur more performance penalties because of cold misses. Take this into account when comparing alternatives where one alternative has had the benefit of a warm cache. The opposite is also possible where we get performance benefits to begin with but as time increases it gets worse (think hash tables). Remember to consider all cases, and don't tailor make your application for just a specific problem and expect its performance to be great in all aspects.
4.) *Ignoring cost of measurement program*. This has been mentioned earlier with respect to how a measurement tool takes up similar resources as what you are trying to measure. For example if you are measuring the performance of a file system, then you have to remember that you tool is also an entire process competing for the CPU and

memory with the processes you are trying to measure. This is impossible to get rid of, but can be minimized! Write out you measured data to RAM instead of disk, keep the code small and cheap, trying bundling it into the process you are measuring, avoid live data analysis, startup the expensive shit before measuring.

5.) *Losing your data*. Don't throw away your data, you never know when it could come in handy.
6.) *Valuing numbers over wisdom*. Performance measurement is to understand the characteristics of your system – not for numbers. Don't bother getting numbers that are not going to add to the wisdom.

Now that we know the steps and precautions to take when performing performance analysis, we can look into a specific type of testing.

Load and Stress Testing

Most testing will try to assert positive or negative (error handling) behavior by stating if situation *x* the program will *y*. This type of testing if a form of *functional validation* and basically verifies if a program works. Load and stress testing is different.

- Number of test cases is relatively small
- Run tests in pseudo-random order for unspecified periods of time
- No expectation of repeatability.
- No definitive passing of a test, but rather how long the test ran for.
- No need to define a complete set of assertions to verify correctness of a specific operation.

The aim of **load testing** is to measure the ability of a system to provide a service at a specified load level. We basically send requests to the specified component at a specified rate and collect data such as:

- Response time for each request
- Mean throughput
- CPU time and utilization
- Disk I/O operations and utilization
- Network packets and utilization

This can then be used to either **measure the system's speed and capacity** or to **analyze the bottlenecks to enable improvements**.

Load generators – A load generator is a system that can generate test traffic, corresponding to a specified profile, at a calibrated rate. This traffic will be used to drive the system that is being measured.

In all cases, the test load is characterized in terms of:

- **Request rate –** The number of operations per second.
- **Request mix –** Different types of clients use a system in different ways. The load generator should be emulate all of the various types of clients.
- **Sequence fidelity –** It may be critical to simulate particular access patterns, or generate the right mix of read and write operations or simulate realistic scenarios against pre-determined or random objects.

Load generators should be able to be tunable in the above factors.

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

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

In the first two usages, the load generator is a measurement tool. In the other usages, it provides a calibrated activity mix, to exercise the system.

Load generators can be used to create realistic traffic to simulate normal traffic for long periods of time. Alternatively, they can be cranked up to much higher rates in order to simulate **accelerated aging**. This allows for testing of memory leaks and other problems that accumulate due to time.

Using load generators to simulate scenarios far worse than anything that is ever expected to really happen, we enter the realm of **stress testing.**

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

Such testing is extremely demanding and can make us sure of the robustness of a program or service. This is more common in mission critical or highly available products.

In mature products, the difference between a good product and a great one is often found it the seriousness of the ongoing performance and robustness programs. **Performance and availability** don't just happen. They must be **earned**.

WEEK 7

Monday:

Threads are not the only ways of concurrent programming. Another method is known as **event-based concurrency.** It is used in GUI-based applications and some internet servers. It aims of solving two problems:

1.) Multi-threading brings about a lot of challenges with missed locks, wakeup races, deadlocks and so on and so forth.
2.) Multi-threading gives the programmer little control over the scheduling since everything is handled by the OS.

The basic premise of such concurrency is – Wait for an event to occur, when it does, check its type and do the small amount of work required for it. This is based around the simple **even loop** construct where we continuously wait for events and process them. **Event handler –** code that processes the event.

The question then arises; how does one know if an event is taking place? The simple answer is an API – *select* or *poll* system calls. Select basically goes through the first *nfds* file descriptors in each set of read, write and error file descriptors, and then modifies those sets with a subset of descriptors that are ready for the requested operation. It also uses a *timeout* argument to set how often select blocks. Setting it to NULL as is most often the case will make select block indefinitely until some descriptor is ready.

```
int select(int nfds,
           fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    // open and set up a bunch of sockets (not shown)
    // main loop
    while (1) {
        // initialize the fd_set to all zero
        fd_set readFDs;
        FD_ZERO(&readFDs);

        // now set the bits for the descriptors
        // this server is interested in
        // (for simplicity, all of them from min to max)
        int fd;
        for (fd = minFD; fd < maxFD; fd++)
            FD_SET(fd, &readFDs);

        // do the select
        int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);

        // check which actually have data using FD_ISSET()
        int fd;
        for (fd = minFD; fd < maxFD; fd++)
            if (FD_ISSET(fd, &readFDs))
                processFD(fd);
    }
}
```

Figure 33.1: **Simple Code Using `select()`**

So how is this simpler? Well on a single CPU case we only have one event being handled at a time so there is no need for continual locking and releasing and the event is **non-preemptive** since its decidedly single threaded.

There is a *problem with the current event-based programming*. If an event requires that you issue a system call that might block. It will take a long time since there is **no overlap** that gives the opportunity to parallelize. So far we aren't seeing events take advantage of the CPU when a certain event is blocked waiting for some operation to be completed. Thus this is a huge potential waste of resources. We could have an iron clad rule that no blocking calls are allowed within events.

Solution: Asynchronous I/O – Enable an application to issue an I/O request and return control immediately to the caller before the I/O has completed; additional interfaces enable an application to determine whether various I/O's have been completed. Most of the API's involved in this are referred to as **AIO control Block**. The structure is first filled an *asynchronous instruction* is called. One API is needed in conjunction with this to determine if the I/O has completed. Mac OSX refers to this as *aio_error()* which checks whether the request referred to by the control block has been completed. If so it returns a success else returns a EINPROGRESS for which the application can periodically *poll* to determine if the I/O has completed or not.

```
struct aiocb {
    int             aio_fildes;         /* File descriptor */
    off_t           aio_offset;         /* File offset */
    volatile void   *aio_buf;           /* Location of buffer */
    size_t          aio_nbytes;         /* Length of transfer */
};
```

More recent systems however choose to inform the completion of an I/O operation through **UNIX signals**. This informs the application of I/O completion that can then be handled through the respective signal handler defined in the application.

Systems **without asynchronous I/O cannot** implement the pure event-based approach. There are clever hybrids however such as events to process network packets and thread pools to manage outstanding I/Os.

Another problem that persists with events is that it is actually harder to implement because process states must be saved manually – **manual stack management**. With threads this was a no brainer since the OS automatically managed the *thread stacks* for each thread.

Solution: Continuation – Record the needed information needed to finish processing the event in some data structure, and when the event happens, look up the needed information and process the event.

More problems with events:

1.) **Multi-core CPUs make events harder –** With more than one CPU we have to run multiple event handlers in parallel and thus the original problems of synchronization with critical sections and locking cannot be avoided.
2.) **Page-faulting is implicit blocking –** Though events forbid blocking operations, something like page faults are hard to catch and thus can really waste CPU cycles and be inefficient.
3.) **Can be hard to maintain over time –** Because blocking is so dangerous for event based programmers, the developers must always be on the lookout to make sure they can accommodate for any changes.

The take away from everything on concurrency is that there is no perfect solution. Event based programming is great for obtaining the power of scheduling but has its draw backs, thread based programming is difficult because it has so many dangerous pitfalls. Choose what's best for you and your situation based on experience and current technologies.

OS Part III – Persistency!

To really understand persistency, we must first understand how I/O devices work on our system. We must dive into some of the old CS 33 stuff – System architecture!

The CPU is connected to the main memory using a **memory bus**, devices are generally connected through the **I/O bus** (in modern systems this is generally **PCI** or some derivative of it), graphics and other high performance I/O devices are found here. Lower than this is the **peripheral bus** such as **SCSI, SATA, USB**. They are used to connect the slowest devices such as *disk, mice* to the system. For reasons of physics and costs, the higher performance devices are generally closer to the CPU while the rest is further away.

Devices have two important components that are key to look at:

1.) **Hardware interface –** An interface that allows the system software to control the hardware's operation. All devices have some specified interface and protocol for typical interaction.
2.) **Internal structure –** This is generally implementation specific and it is what provides the abstraction the device presents to the system. They can be as simple as a few hardware chips or their own CPUs. **Firmware –** Software within a hardware device.

The simplest protocol referred to as the canonical protocol has a **status –** read for the current state of the device; **command –** tells the device to perform a certain task; **data –** pass data to device or from device; registers.

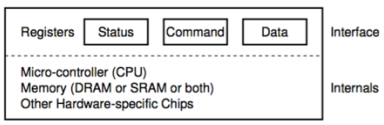


Figure 36.2: **A Canonical Device**

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

The most basic protocol involves **polling** the device continuously by the CPU. When the CPU is involved in the data movement it is referred to as **Programmed I/O (PIO)**. The writes through the CPU are generally placed in a transfer disk block to the device. The problem with this protocol is that it is *inefficient as it wastes CPU cycles through continual polling*.

As we have been learning the more common approach is through **interrupts**. When the CPU issues a request to I/O it simply puts the calling process to sleep and performs a context switch to run a different process. When the device has finished the task it will raise a hardware interrupt causing the CPU to jump into the OS at a pre-determined **interrupt service routine (ISR) [Interrupt handler].** This will finish the request and wake the process. This method allows for overlap of computation.

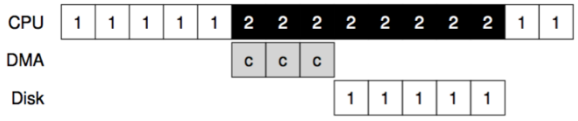
However, note that interrupts are not the best approach in all situations. If the I/O is a **short one** then performing the context switch so rapidly may result in an overhead that is larger than the I/O operation itself. In such as situation polling might be the better suited. But because **time is not always known** it is best to use a **hybrid two-phase** approach where we poll for a little while and then switch to the interrupt method.

Interrupts with networks can also lead to **livelocks** where the OS is only processing interrupts due to a high load of incoming packets and thus the user process never gets to run.

Another *benefit* with interrupts is **coalescing**. If we have multiple interrupts at the same time, then we can coalesce them into one interrupt delivery. There must be a tradeoff made in determining how long to wait to coalesce the interrupts such that it is optimal.

PIO however is not the most optimal way of solving data transfer to devices. The CPU is involved in a trivial task that wastes time and effort that could be better spent running other processes.

Direct Memory Access (DMA) – A DMA engine is a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention. The OS programs this engine telling it where the data lives in memory, how much data to copy and which device to send it to. Thus the *DMA controller* handles the data transfer and raises an interrupt to tell the OS that it is done with the transferring. This allows the CPU to focus on running different processes in this time!



But how exactly does the OS communicate with these devices?!

1.) In older times there used to be explicit **I/O instructions**. They specified a way for the OS to send data to specific device registers. This meant that the instructions were **privileged** and it made it such that only the OS could communicate with the devices.
2.) **Memory Mapped I/O**. This allowed for sections of memory specifically allocated for communication with devices. Thus the OS could load/read from a specific address which would the same as reading from the device registers. This is nice because no new instructions have to be created.

How can we keep most of the OS device-neutral, hiding the details of device interactions from major OS subsystems?

We solve this issue through abstraction. At the lowest level the OS has to know the detail about each device, which it does through a piece of software called a **device driver**. This software is at the bottom most layer of multi-layered approach. This is great because it makes interaction with devices so easy but is not the best because it takes the specific strengths from each device away because of the generalization it has to provide. Device driver code is ~70% of Linux code and is generally the most of your kernel.

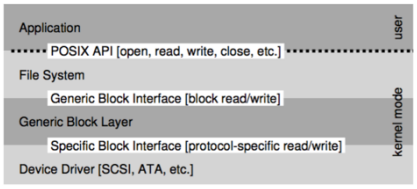
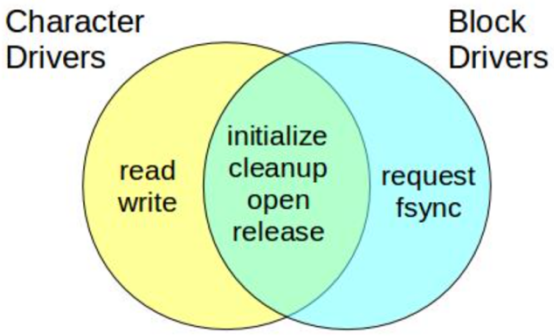


Figure 36.3: **The File System Stack**

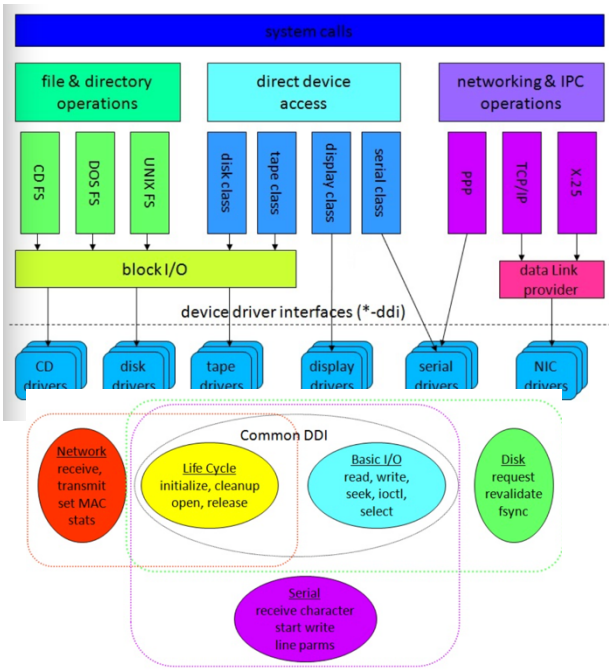
For the most part devices are broken up into two fundamental types:

1.) **Block devices –** Random access devices with fixed size addressing. Their drivers implement a request method to enqueue asynchronous DMA requests. This request includes information about desired operation, completion information, condition variable used for awaiting completion. 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. 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.
2.) **Character devices –** Sequential access or byte-addressable. 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. 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 classes of drivers are not mutually exclusive. A single driver could export both types. 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.



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 output 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

Each of these sub-class specific interfaces is referred to as a Device Driver Interface (DDI).

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.

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 arise for other classes of dynamically loadable kernel modules (such as network protocols and file systems).

Hard Disk Drives

These are the main form of persistent data storage in computer systems and much of the development of file system technology is based on their behavior. The basic interface for modern drives consists of large number of 512 byte blocks known as **sectors** which can be read from or written to. They are essentially a large array numbered from 0 to n-1, which is the *address space of the drive.*

Most modern drives support multi-sector operations, that is operating on a chunk of sectors (~4KB) at a time, but it is not guaranteed to be safe as only 512 bytes are **atomic**. Thus you could have data loss during power failure or something of the sort – **torn write**. We can for the most part assume that operating on sectors that are close to each other and in contiguous chunks is faster than the alternatives.

Structure of the disk:

Platter – Circular hard surface on which data is stored persistently through inducing magnetic charges. Generally made from hard substance such as aluminium and then thin magnetic layer.

Surfaces – Each side of the platter (max of 2 sides per platter).

Spindle – The central point around which each platter is bound. This is connected to a motor than spins the platters around at a constant rate. This is calculated through **Rotations Per Minute (RPM)**.

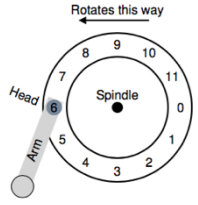


Figure 37.2: **A Single Track Plus A Head**

Track – Data is encoded in concentric circles of sectors, and one concentric circle is called a track. Note that a single surface contains many thousands of tracks tightly packed together, with hundreds of tracks fitting into the width of a human hair.

Disk head – This is the hardware that reads and writes to each sector. There is one per surface, and is attached to a single **disk arm** which moves across the surface to position the head over the desired track.

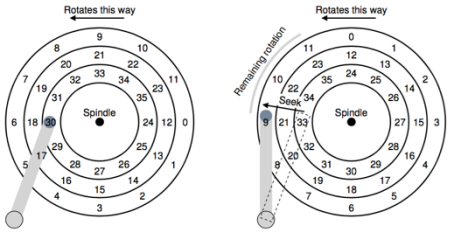


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

Single-track Latency: Rotation Delay – The time it takes for the specific sector to rotate to under the head.

Multiple tracks: Seek Time – Time taken to move the disk arm to the correct track. This along with rotation delay is the costliest disk operation. There is *acceleration*, *coasting*, *decelerating*, *settling* phase in the seek time, and it is the **settling time** that incurs the most time.

Once the head is over the correct sector we have **transfer** which is the final step in the I/O to disk. Seek -> Rotational Delay -> Transfer.

Track skew – Sometimes we skew the sectors for continuous seek and read. Without said skew we would seek the head to a new track and by that time the relevant sectors may have just passed under us and we would have to reposition the head to get to the right spot by incurring rotational delay.

Zones – Disks due to their circular nature can store more sectors at the outer edges. Thus we have **multi-zoned** disk drives where outer zones have more sectors than inner zones. It is a set consecutive tracks.

Track buffer (cache) – The disk can store about 8-16MB of read or write information into its memory for quicker future access to the same tracks.

Write back – Acknowledges the write as completed when it has put the data in memory. This can lead to issues involving a specific ordering of information.

Write through – Acknowledge the write after it has actually been written to disk.

Random workloads – Small reads to random locations on the disk. Important in Database management systems.

Sequential Workloads – Operate on a large number of sectors consecutively without jumping around.

We notice that generally there is a huge gap in drive performance between random and sequential workloads. Also most drives are broken into *performance* and *capacity* divisions. Generally high performance is more expensive than higher capacity.

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer} \qquad R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$$

Due to the high costs of I/O, the OS has a dedicated **disk scheduler** which

decides which I/O operations to perform first. Unlike process scheduling, disk scheduling generally involves knowing how long a job will take by estimating the seek and rotational delay of a request.

SSTF: Shortest Seek Time First – Pick requests on the nearest track to complete first. The OS doesn't really know the geometry of the drive and so it implements this policy through Nearest-block first (NBF) as it just sees the disk space as blocks. This method could lead to starvation if we had a steady stream of requests to the inner track and so any sectors requested on the outer most tracks would almost completely be ignored.

Elevator (aka SCAN or C-SCAN) – The general scan algorithm goes through the entire set of tracks in a *sweep* servicing all the sectors in the tracks in the sweep. If a sector is missed it will be queued until the sweep in the other direction. This has many variants:

F-SCAN - Freeze the queue to be serviced during the sweep. This replaces requests that come in during the sweep into a queue to be serviced later. This avoids starvation of far-away requests by delaying the servicing of late-arriving requests.

C-SCAN – Circular scan, that is don't perform a sweep in opposing directions one after the other. That is this variant only sweeps from outer-to-inner and then resets the outer track to begin again. This is fairer to the outer and inner tracks.

Neither SCAN or SSTF adhere to the SJF (Shortest Job First) principle as is ideal with disk scheduling. In fact, they ignore the idea of rotation completely.

SP/ATF: Shortest Positioning/Access Time First – Note that sometimes seek times are better than rotational delays so something like SSTF may not pick the fastest. Thus use the actual physical positioning of the sector and access the closer positioned sector first. This is hard to do in the OS since it has no information about the geometry and so is done in the drive itself (through the disk controller).

The OS used to be the one to disk schedule back in the good old days but now we see the drive do most of the work. The OS scheduler picks what it thinks are the best few requests are and then issues them to the disk, which then uses its internal knowledge of head position and detailed track layout information to do the magic. I/O merging – Merge requests that are consecutive into one request as it reduces the number of requests sent to disk and thus lowers the overhead performed by the OS.

Work conserving – Issue I/O to disk immediately to continually keep the disk busy. However, research on anticipatory disk scheduling has shown that non-work-conserving approaches are better, since a more efficient request may come through or we can find that requests can be merged. Again the trade-off between how long to wait and issue the I/O request is something only experience will grant.

Redundant Arrays of Inexpensive Disks (RAIDs)

This is a technique to use multiple disks in concert to build a faster, bugger and more reliable disk system. Externally RAID looks just like a disk with blocks to read from or write to. RAID internally however is much more complex and is like a computer system in itself used for managing a group of disks using memory and one or more processors to manage the system.

It offers *performance* since it uses multiple disks in parallel. It also offers larger *capacity* since we can have pretty large disks. *Reliability* through spreading of data across multiple disks with forms of **redundancy** such that these systems can perform normally even in the event of loss of a disk.

All this is offered **transparently** as RAID just appears as a big disk to the host system. This allows us to replace a disk with RAID without ever changing a single line of software, and everything seems to operate completely normally. This thus greatly improves the **deployability** of RAID.

RAID presents itself as a linear array of blocks. When a *logical I/O* is serviced the RAID must internally calculate which disk to access in order to complete the request, and then issue one or more *physical I/Os* to do so. A simple RAID system would mean have two copies of each block on separate disks, and in such a **mirrored RAID** we perform two physical I/O for every one logical I/O.

RAID is generally built separately as a hardware box with standard connection to a host. However, it internally contains a microcontroller that runs the firmware to direct the operation of the RAID, volatile memory such as DRAM to buffer data blocks and in some cases non-volatile memory to buffer writes safely and perhaps specialized logic to perform parity calculations.

Fault Models – The different models of errors that a disk can encounter. The simple fault model is **fail-stop** wherein a disk can be either working or failed. In working state, it can have normal operations done on it, but in failed state we assume everything is permanently lost. We can thus assume such a case is easily detectable with RAID.

When evaluating a RAID we measure it along three axis that were already mentioned above:

1.) **Capacity –** For a set of *N* disks with *B* blocks, without redundancy we have *NB*, with mirroring we have *NB/2* and with parity we have something in between.
2.) **Reliability –** How many disk faults can the design handle, this is a topic of further discussion in data integrity.
3.) **Performance –** Depends on the workload presented to the disk array.

We consider three important RAID designs: Level 0 (stripping); Level 1 (mirroring) and Levels 4/5 (parity based redundancy).

1.) **RAID Level 0 (Stripping) –** It serves as an excellent upper bound on performance and capacity. Spread the blocks of the array across the disks in a round robin fashion. This extracts the most parallelism from the array when requests are made for contiguous chunks of the array. The blocks in the same row is known as a **stripe**. The **chunk size** is the number of blocks we place in a disk until we move onto the next disk. Small chunks size means higher distribution and thus more parallelism of reads and writes to a single file, but positioning time increases across multiple disks. Big chunk sizes reduces intra-file parallelism, relying on multiple concurrent requests to achieve high throughput, but reduces positioning time. General chunk sizes are 64KB.

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

Figure 38.2: **Striping with a Bigger Chunk Size**

   a. **Capacity –** Great! We get NB blocks.
   b. **Reliability –** Not good as failure will lead to complete data loss!
   c. **Performance –** Great! We use all disks and parallelize. From the *latency perspective* the latency of a single-block request is about the of a single disk. From the *steady-state throughput* perspective we'd expect to get the full bandwidth of the system. NS MB/s for sequential and NR MB/s for random workload.

2.) **RAID Level 1 (Mirroring) –** Simply make more than one copy of each block in the system on separate disks so we can tolerate disk failures. In fact, we can use this in conjunction with RAID 0 and is called **RAID-10 (1+0) or RAID-01 (0 + 1)**. We copy over the data into a different disk and stripe them. In reading this RAID level has a choice, but in writing it must update both copies to preserve reliability. The writes can happen in parallel.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: **Simple RAID-1: Mirroring**

   a. **Capacity –** Expensive! If the mirroring level is M then we get NB/M as useful capacity.
   b. **Reliability –** Does well. It can tolerate disk failure as it maintains copies. N/M failures depending on which disks fail. Generally however mirroring is good for handling a single failure.
   c. **Performance –** From *latency perspective* in the read case is the same as that on a single disk. A write has to be done to two disks but is parallel so is almost that of a single write but the logical write must wait for both writes to complete which makes it suffer from worst-case seek and rotational delay of the two requests and thus on average will be a little higher than a write to a single disk. Maximum bandwidth is NS/2 MB/s for mirroring of 2 with both reading and writing. The random read bandwidth on the other hand is pretty good as it is distributed amongst multiple disks -> NR MB/s, while the random write bandwidth is NR/2 MB/s

3.) **RAID Level 4 (Saving Space with Parity) -** Parity-based approaches attempt to use less capacity and thus overcome the huge space penalty paid by mirrored systems. That is one disk will store all the redundant information from that row of blocks. The number of 1's in any row must be an even number. Such is the **invariant of the RAID parity** since it uses XOR to calculate the parity (1 if odd and 0 if even). We can find the lost information through a **reconstruction**. If we have 0,0,1,0 then an XOR will result in a 1 and so the missing data must be a 1. For multiple bytes we do a bitwise XOR across each bit of the data blocks and put the result in the corresponding bit slot in the parity block.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.4: **RAID-4 with Parity**

   a. **Capacity –** Pretty good! We only use one disk for parity information and thus we have useful capacity of (N-1)B.
   b. **Reliability –** Tolerates one disk failure and no more, which isn't too bad.
   c. **Performance –** In steady state throughput – Sequential reads use all but the parity disk thus (N-1)S MB/s. With Sequential writes we have RAID 4 do a **full-stripe write –** perform an XOR parity amongst all the blocks in the same row and then perform a parallel write to all disks. This evaluates to (N-1)S MB/s. Random reads will be spread across all but the parity disk and so we have (N-1)R MB/s. In random writes we must take care to update the parity block as well in two ways:
      i. **Additive parity –** Read in all the data from the blocks in the stripe in parallel and XOR them with the new block and write the data and the new parity in parallel to their respective disks. It scales with the number of disks and so will require more reads with more disks.
      ii. **Subtractive parity –** Read in old data from the block we wish to overwrite, read old parity, compare old and new data and if the same the parity bit is the same, else we flip the old parity bit.

   The parity disk generally becomes the bottleneck. This is called the **small-write problem -** even though the data disks could be accessed in parallel, the parity disk prevents any parallelism from materializing; all writes to the system will be serialized because of the parity disk.

   Thus we get random writes at R/2 MB/s since we have two I/Os per logical I/O. With single latency we see the read is the same as in the case of a single disk request. The latency of a write requires two reads and then two writes and though they can be done in parallel we see that the latency is equal to that of twice of a single disk.

4.) **RAID Level 5 (Rotating Parity) –** Rotate the parity blocks across drives which will solve the small-write problem partially. Because it is almost the same as RAID 4 it is most commonly used over RAID 4 but is harder to build.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figure 38.7: **RAID-5 With Rotated Parity**

   a. **Capacity -** Pretty good! We only use one disk for parity information and thus we have useful capacity of (N-1)B.
   b. **Reliability –** Tolerates one disk failure and no more, which isn't too bad.

c. **Performance –** Random read performance is a little better since we utilize all disks. Random write performance improves by a lot since we can actually have parallelism -> NR/4 MB/s.

There also exist other RAID Levels such as 2 and 3 and even 6 for multiple disk faults. There is also what the RAID does when a disk fails; some- times it has a hot spare sitting around to fill in for the failed disk. Finally, you can even build RAID as a software layer: such soft- ware RAID systems are cheaper but have other problems, including the consistent-update problem.

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

Figure 38.8: **RAID Capacity, Reliability, and Performance**

Dynamically Loadable Kernel Module

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.

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).

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.

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.

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 heterogeneous 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.

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.

All of this his 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 compatibly 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 version some incompatibilities.

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 compatibility with old device 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.

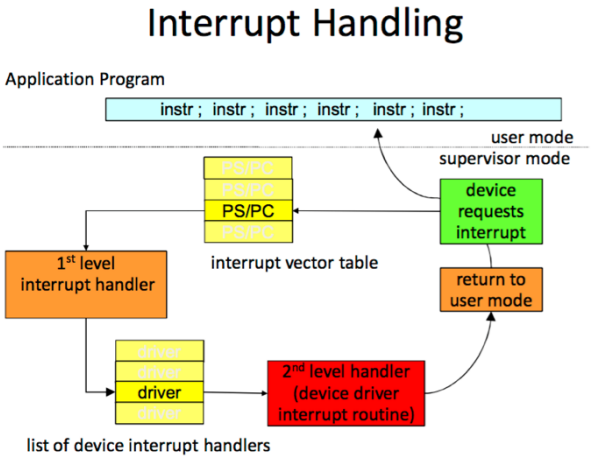
Hot-pluggable busses often have multiple power levels, and a newly inserted device may receive only enough power to enable it to be queried and configured. When the driver is ready to start using the device, it can instruct the bus to fully power the device. Some hot-pluggable busses also have mechanical 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.

SOME NOTES FROM CLASS:

For fast reads we can read in cylinders which are essentially 10 cascading tracks along the same head position. Flash (SSD's) is a lot faster than hard disks are becoming obsolete. They don't have any rotational or seek latency and has much higher transfer rates.

We used to have storage paradigms local on our machines with disks, paging, swapping, file systems etc, but now we have NAS and distributed network based systems – "the cloud".

A device can tell the interrupt controller that it wants to issue an interrupt. When there are no higher priority interrupts in the queue, this controller will send an interrupt through the interrupt BUS. This is very similar to the trap handler process.



Chain completion works as follows:

Keeping Key Devices Busy

- allow multiple requests pending at a time
  - queue them, just like processes in the ready queue
  - requesters block to await eventual completions
- use DMA to perform the actual data transfers
  - data transferred, with no delay, at device speed
  - minimal overhead imposed on CPU
- when the currently active request completes
  - device controller generates a completion interrupt
  - interrupt handler posts completion to requester
  - interrupt handler selects and initiates next transfer

When a trap happens you can't disable other traps. But with interrupts can do that. I'm doing an operation; another operation of the same type can't happen.

Big transfers are always guaranteed to give you higher throughput because the setup time is a lot lower for a lot more transfer. This is why we buffer I/O. **Read-ahead** – Use a cache of recently used disk blocks and then read more than you want at the time of servicing the buffers because you're most likely to get correct data. It's a gamble for every sequential I/O. This **deep request** reduces mean seek distance/rotational delay and may be possible to combine adjacent requests.

(double buffered input)

- have multiple reads queued up, ready to go
  - read completion interrupt starts read into next buffer
- filled buffers wait until application asks for them
  - application doesn't have to wait for data to be read
- when can we do chain-scheduled reads?
  - each app will probably block until its read completes
    - so we won't get multiple reads from one application
  - we can queue reads from multiple processes
  - we can do predictive read-ahead

Solicited input – Input someone asked for. Can you please go read this block of disk?

Unsolicited input – Comes from a network interface. I'm sure someone

(double-buffered output)

- multiple buffers queued up, ready to write
  - each write completion interrupt starts next write
- application and device I/O proceed in parallel
  - application queues successive writes
    - don't bother waiting for previous operation to finish
  - device picks up next buffer as soon as it is ready
- if we're CPU-bound (more CPU than output)
  - application speeds up because it doesn't wait for I/O
- if we're I/O-bound (more output than CPU)
  - device is kept busy, which improves throughput
  - but eventually we may have to block the process

Scatter/Gather I/O

- many controllers support DMA transfers
  - entire transfer must be contiguous in physical memory
- user buffers are in paged virtual memory
  - user buffer may be spread all over physical memory
  - *scatter*: read from device to multiple pages
  - *gather*: writing from multiple pages to device
- three basic approaches apply
  - copy all user data into contiguous physical buffer
  - split logical req into chain-scheduled page requests
  - I/O MMU may automatically handle scatter/gather

needs an operation but I have to look at the network packet to determine who. This needs read-ahead.
Pinned pages mean they can't be paged out of the current spot in memory.

If you want I/O through MMU then you need to store the memory management data for the I/O specific processes so we don't lose the state and memory locations of the I/O transfers.

We scatter reads into paged memory and gather writes from paged memory.

SRIOV – Single Root I/O Virtualization – Create a bunch of queues that can be polled continuously. Basically create something that can make it seem like being multiple devices by breaking the queues to virtualization of devices.

Wednesday:

When dealing with persistency we must take into account two key abstractions in the virtualization of storage:

1.) **File** – Linear array of bytes each of which you can read or write. The **low-level name** of it is referred to as an **inode number –** the inode itself is a data structure that holds pertinent information for persistent data storage. The type of file is unknown to the OS, but just a storage for data.
2.) **Directory** – Also has a low-level name that is an inode number but is more specific. It contains a list of pairs. Each entry in a directory refers to either files or other directories. We can create **directory trees** through putting subdirectories.

Absolute pathname – The path to the file/directory from the *root* directory.

The filesystem just provides a convenient way to *name* all the files by adding extensions/types to the end of file names.

Open() or *creat()* can both open files for reading or writing or truncating if the file already exists and then operate on it.
Read() can be used to read in blocks of characters at a time from a file descriptor into a local buffer.

File descriptor – An integer that is *private per process* and is used to access files. It is a capability.

Note: when you use cat, depending on how many processes you are using, the file descriptor returned to it by open would be **n** if you were using n processes, since each process creates 3 file descriptors automatically.

Lseek() allows us to update the current offset into a file for different patterns of read and write. The second argument is our offset, and the third is a flag that determines the kind of offsetting we are doing.
Sometimes with writes using the write system call, we face the danger of losing data if a failure occurs before the write buffer fills up, and we may lose data. Thus we use *fsync()* which forces all **dirty** data to disk immediately. Sometimes however, we may need to call fsync() on the entire directory because it has a few failure points.

Renaming allows us to **atomically** change the name of a file, but also allows you to copy over files into a new temp file that is fsynced to disk! This is what an editor like vim or emacs does.

Stat() allows us to grab information about a file or directory into a struct such as inode number, name, size, user id of file etc.

Mkdir() can be used to create directories. If you create an empty one it will contain two things inside it, a reference to itself and one to its outer directory.

Similar to files, directories have *opendir(), readdir()* and *closedir()*. These are the calls a command like *ls* actually makes!

Rmdir() only deletes empty directories because it forces the user to make sure that what they're deleting may contain large amounts of critical data, and call such is dangerous unless you have no files in the directory.

The *rm* command on UNIX calls a system call *unlink()* that simply takes the name of a file and returns 0 upon success! What does it actually do though? To understand this we need to understand the *link()* command. When talking about linking we focus on two types of linking in the UNIX file system:

1.) **Hard links –** creates another name in the directory to the file you just linked. That is we now have two names point to the same inode number. We are linking a *human readable name to a file*. Unlinking such a link works because it **decreases the reference count** (link count) within the inode number, which allows the system to track how many different files have been linked to this particular inode. Only when the link count reaches 0 does the file system free the inode and data blocks that were associated with said file.
2.) **Symbolic links –** Hard links are limited in that you can't create one on a directory since you could end up creating a directory tree cycle, and you can't hard link to other files on other disk partitions. The symbolic link is a *file in itself* that is of a third type in the UNIX file system (other than files and directories).

Dangling reference – Deleting the original file leaves the symbolic link pointing to an invalid pathname.

Mkfs() creates a file system onto a specified disk location, with a specified file system type. But after creation we must *mount*. It takes an existing directory as a target **mount point** and pastes a new file system onto the directory tree at that point.

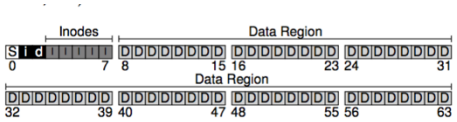
A file format is a standard way that information is encoded for storage in a computer file. It specifies how bits are used to encode information in a digital storage medium.

1.) One popular method used by many operating systems, including Windows, Mac OS X, CP/M, DOS, VMS, and VM/CMS, is to determine the format of a file based on the end of its name—the letters following the final period.
2.) A second way to identify a file format is to use information regarding the format stored inside the file itself, either information meant for this purpose or binary strings that happen to always be in specific locations in files of some formats. Since the easiest place to locate them is at the beginning, such area is usually called a file header when it is greater than a few bytes, or a magic number if it is just a few bytes long.
3.) A final way of storing the format of a file is to explicitly store information about the format in the file system, rather than within the file itself. This approach keeps the metadata separate from both the main data and the name, but is also less portable than either file extensions or "magic numbers", since the format has to be converted from filesystem to filesystem.

File systems are purely software based and there are many versions out there. When it comes to thinking about file systems it is important to focus on two concepts:

1.) **Data structures –** On-disk structures utilized by the file system to organize data and metadata. Some early file systems used arrays or lists, while newer more sophisticated ones use trees.
2.) **Access Methods –** Mapping the calls made by a process to the respective file.

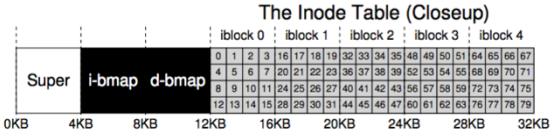
Looking at the **vsfs (Very Simple File System)** we see that data is arranged in blocks (generally of only one size). The disk can thus be thought of as partitioned. In this set of partitioned blocks, we reserve some blocks for *user data*.



Metadata – Information about the file itself that the system has to keep a track of for reference.

We spoke about inodes earlier and how they are used to refer to files' metadata, and so a small chunk of the partitioned space is used for the **inode table**, which simply holds an array of on-disk inodes.

We must also keep track of whether things are **free** or **allocated**. Thus we have an **allocation structure** as well. This is essentially what a **free-list** is, as we mentioned a while ago! For the sake of vsfs we choose something even simpler – **bitmap –** A matrix of bits that refer to the data bitmap and inode bitmap.



Finally, there exists a **Superblock** which contains information about the particular file system, such as the number of inodes and data blocks and more.

```
blk    = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

More about inodes – They stand for *index nodes* because they used to be originally arranged in an array. They are referred to through their *inumbers* and they may or may not start at the beginning. For the most part in memory we have the super blocks come first. After creating a simple bitmap, going to that inode number is just going to be on an offset.
One of the most important decisions in the design of the inode is how it refers to where data blocks are. One simple approach would be to have one or more **direct pointers** (disk addresses) inside the inode; each pointer refers to one disk block that belongs to the file. Such an approach is limited: for example, if you want to have a file that is really big (e.g., bigger than the size of a block multiplied by the number of direct pointers), you are out of luck.
Another way of doing so is to use **indirect pointer –** Instead of pointing to a block containing user data, just point to a block of more pointers. If a file grows large enough, an indirect block is allocated (from the data-block region of the disk), and the inode's slot for an indirect pointer is set to point to it. This is referred to as a **multi-level index.** As we keep growing the size of indirect pointers (double, triple, etc.). The reason this imbalanced tree structure works so well is because the generic truth of file systems is that most files are small and thus this imbalanced design optimizes for this.
A different approach is to use extents instead of pointers. An extent is simply a disk pointer plus a length (in blocks); thus, instead of requiring a pointer for every block of a file, all one needs is a pointer and a length to specify the on-disk location of a file. Just a single extent is limiting, as one may have trouble finding a contiguous chunk of on-disk free space when allocating a file. Thus, extent-based file systems often allow for more than one extent, thus giving more freedom to the file system during file allocation.

Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Figure 40.2: **File System Measurement Summary**

A **directory** is just a list of pairs as mentioned earlier. It stores the entry name and inode numbers. It may store some extra information as well such as the record length to find if subdirectories exist and string length etc. The directory is treated like a special file and so it has an inode in the inode table. The directory has data blocks that the inode block pointers lead to. Older systems used to have directories as a simple linked list structure but now we have them as B-trees and more complex structures.
Free space management is either handled through *free lists* as we saw with memory allocation earlier, or more commonly through the *bitmaps* we noted before. If an inode and thus the corresponding block is marked as 0 then it is free and can be used. Some systems will look for contiguous sequence of free space to make the data access faster on disk and this policy is referred to as **pre-allocation**.

When we mount the file system the superblock is put into memory and the file system is attached to the existing file system on disk.

Access Paths are the way the filesystem works in reading and writing data.
When we wish to read some data by opening a file the following happens:

1.) The root directory inode number is well known and in UNIX systems is set to 2 in the inode table. The search for the file we wish to read begins from here.
2.) Look through the data pointers of root to find the next directory in the path, and get its inode number. Do this recursively until we find the final file we wish to.
3.) Read the file's inode into memory and do a permissions check. Then allocate a file descriptor for this process in the per-process open-file table.
4.) The read system call will then read the first data block for this file by consulting the inode block pointers. It will update the access time, the in-memory open file table for the descriptor and update the offset.
5.) Upon closing the file descriptor is deallocated and no disk I/O needs to take place.

Note that the amount of I/O generated by an open is proportional to the length of the pathname. The bigger the directories the harder and longer it is to find the correct file.

When *writing* to a file, the opening process is the same but in addition we also gain the ability to allocate a block. It has to write data to the disk but also decide on which block to allocate to the file and thus update other structures of the disk accordingly. Thus each write essentially generates the following 5 I/O's:

1.) Read data bitmap
2.) Write the bitmap
3.) Read the inode
4.) Write the inode
5.) Write the actual data block

File creation also takes a long time with having to read the inode bitmap to find a free inode, writing to the inode bitmap to mark it as allocated, writing to the new inode to initialize it, writing to the data of the directory to link the high level file name and then to read and write to the directory inode to update it. So it looks like we have a lot of I/O done for such seemingly simple operations, so how does it do all this so quickly?!

As always the solution is caching. Early file systems introduced *fixed-size cache* to hold popular blocks. This would be allocated at *boot time* and would be about 10% of total memory. However, this static partitioning can be wasteful. Thus modern systems use dynamic partitioning by integrating virtual memory pages and file system pages into a unified page cache.

Thus with this caching, though we may have a compulsory miss in the beginning, subsequent reads to this file will be blindingly fast in comparison to the old approach. This however, only

solves reading issues, because with writing we still have to write to disk to make it persistent. Thus we use another age old approach that is **write buffering**. We delay writes and *batch* them into a chunk for a smaller set of I/O's. By doing this the system can *schedule* the I/O's for improved performance. This also gives the benefit of avoiding some writes such as the case of writing and then deleting the file. This method completely avoids wasting I/O operations for such a menial task!

Modern systems buffer the writes in memory for about 5~30 seconds. This obviously brings the tradeoff data integrity with system crashes as writes are not as common. Thus some systems such as databases use *fsync()* to force the writes using direct I/O or using raw disk interface to avoid caching and the filesystem entirely.

DOS FAT Volume and File Structure:

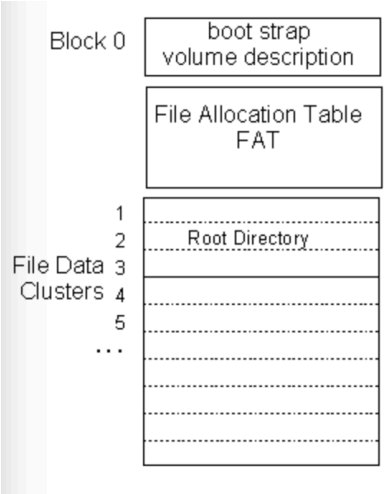
BIOS – BASIC I/O Subsystem – The BIOS ROM is what was used to provide run-time support for BASIC interpreter. This is what was used for very small disks back in the old days.

The DOS FAT file system is one of the most popular examples of *linked list space allocation* and is still used heavily around the world.

It must be noted that all computers have the following basic types of data structures:

1.) **Bootstrap –** Code that is loaded into memory and executed when the computer is powered on. The entire first track of the first cylinder is for the bootstrap.
2.) **Volume descriptors –** Information about the size, type and layout of the file system and how to find the key meta data descriptors.
3.) **File descriptors –** Information that describes a file and points where the data is actually stored on disk.
4.) **Free space descriptors –** Lists of currently unused blocks of space that can be allocated to files.
5.) **File name descriptors –** Data structures that user chosen names with each file.

The DOS FAT system divides the volume into fixed-sized blocks which are grouped into larger fixed sized logical block clusters. The first block contains the bootstrap and volume

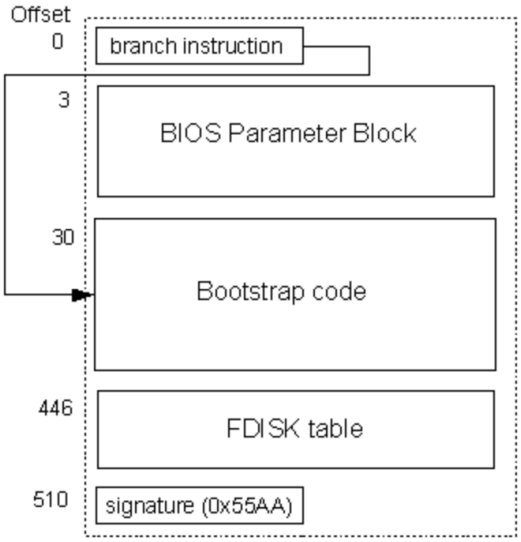


information, then is a long **File Allocation Table** which is a free list and keeps track of which blocks have been allocated to which files. The rest of the volume is data clusters for files and directories.

The boot record contains the following:

1.) Branch instruction to the start of the bootstrap.
2.) Volume description (BIOS Parameter block)
3.) Real Bootstrap code
4.) Optional disk partitioning table
5.) Signature for error checking

The BIOS Parameter Block describes the device geometry about the disk and describes how the file system is laid out on the volume. This allows the OS to interpret the remainder of the file system.



The **FDISK** table is a table containing information about multiple partitions on a disk. It is attached to the end of the Bootstrap. It includes the following:

1.) Partition type
2.) ACTIVE indication (where we boot from)
3.) Disk address where the partition starts and ends as logical block numbers
4.) Number of sectors contained within the partition

With this disk partitioning, the boot record changed. The first sector of a disk now contains a **Master Boot Record (MBR)** with the FDISK table and a bootstrap to find the active partition and reads in the first first sector – **Partition Boot Record**. Most MBR's ask the user which system they wish to boot from else boot the active default if nothing is chosen, but Bill Gates my main man chose to force Microsoft Windows as the booting option without letting users choose.

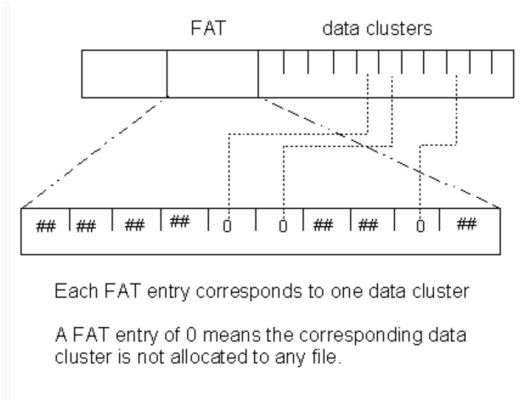
DOS combined file descriptors and file naming into a single descriptor called the **directory entry**. A directory is a special file that contains a series of 32-byte directory entries. This entry contains things like an 11-byte name, attribute bits, times and dates about creation, modification and access, pointer to the first logical block of the file and length of valid data bytes in the file.

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.

Space was allocated to files in logical multi block clusters determined at the creation of the file system. Allocating space in larger chunks improves I/O performance but leaves higher internal fragmentation because on average, half of the last cluster of each file is left unused. The maximum number of clusters depends on the width of FAT entries.

The file's directory entry contains a pointer to the first cluster of that file, and the FAT entry for that cluster tells us the cluster number next in the file. At the last cluster, the FAT entry contains a -1 to denote no other block exists for this file. This requires strict **sequential** access, through *next block pointers*. The FAT itself is relatively small and is thus loaded into memory when the file system is in use which decreases the number of I/O operations required to disk.



As far as *free space* is concerned, a 0 in the entry refers to a free cluster.

Previous versions of FAT didn't free space but rather crossed out the first byte of the file name in the directory entry which greatly reduced the amount of I/O associated with the file deletion but it meant it ran out of space regularly. Upon such an event **garbage collection** was instantiated by recursively going through all the directories and sub directories from the root, inferring that any cluster not found in a file was free and marked them as such in the FAT. This had the *benefit* of storing files even after deletion until garbage collection started.

The main complaints that came with the DOS and Windows systems were the 8+3 file names in that each file name had to be 8 bytes long at most with a 3-byte extension. The solution they came up with was to put the extended filename in additional directory entries. To maintain backwards compatibility, the extension entries had some odd attributes which would be ignored and default to the 8+3 file 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 (data file), 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.).

Because the FAT is so important to the working of the file system, any kind of corruption to it could be catastrophic, and so Microsoft added support for alternate FAT's. The sloppy solution

they came up with was to back up the primary FAT to a pre-reserved alternate FAT location, keeping all the old files you had but losing all the new ones.

The ISO 9660 file systems built for CD ROMs was based on the DOS system for compatibility and built on the mistakes of DOS.
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.

Object Storage:

Object storage (also known as object-based storage) is a storage architecture that manages data as objects, as opposed to other storage architectures like file systems which manage data as a file hierarchy and block storage which manages data as blocks within sectors and tracks. Each object typically includes the data itself, a variable amount of metadata, and a globally

unique identifier. Object storage can be implemented at multiple levels, including the device level (object storage device), the system level, and the interface level. In each case, object storage seeks to enable capabilities not addressed by other storage architectures, like interfaces that can be directly programmable by the application, a namespace that can span multiple instances of physical hardware, and data management functions like data replication and data distribution at object-level granularity.

Object storage systems allow relatively inexpensive, scalable and self-healing retention of massive amounts of unstructured data. Object storage is used for diverse purposes such as storing photos on Facebook, songs on Spotify, or files in online collaboration services, such as Dropbox.

Key-value database:

A key-value store, or key-value database, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash. Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to quickly find the data within the database.

Key-value stores work in a very different fashion from the better known relational databases (RDB). RDBs pre-define the data structure in the database as a series of tables containing fields with well defined data types. Exposing the data types to the database program allows it to apply a number of optimizations. In contrast, key-value systems treat the data as a single opaque collection which may have different fields for every record. This offers considerable flexibility and more closely follows modern concepts like object-oriented programming. Because optional values are not represented by placeholders as in most RDBs, key-value stores often use far less memory to store the same database, which can lead to large performance gains in certain workloads.

Performance, a lack of standardization and other issues limited key-value systems to niche uses for many years, but the rapid move to cloud computing after 2010 has led to a renaissance as part of the broader NoSQL movement. Some graph databases are also key-value stores internally, adding the concept of the relationships (pointers) between records as a first class data type.

File system in User Space:

Filesystem in User space (FUSE) is a software interface for 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.

WEEK 8

Monday:

Locality and Fast File System

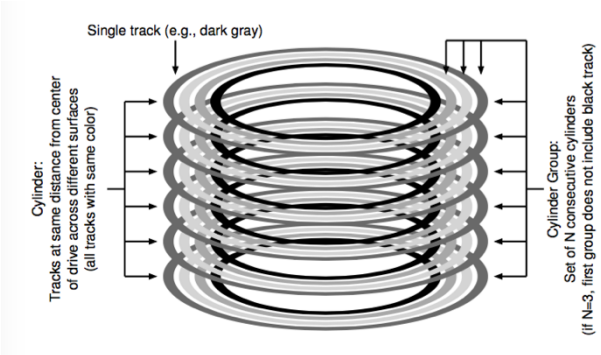
The first UNIX file system (VSFS) written by Ken Thompson was extremely straightforward and supported the basic abstractions the file system tried to deliver. The essential structure of it was we have the Superblock containing information about the entire file system, the inode contained all the inodes for the file system and most of the disk was taken up by data blocks. This was found to be pretty bad in terms of performance. It was found that this system started out as bad and got worse over time to the point where it was only using 2% of the disk bandwidth!

This was because it treated the disk like RAM as data was spread all over the place without considering how a disk actually worked with seeking and head movement. It also did not manage the free space very well and so the data was for the most part fragmented. As files got allocated they would just get random free blocks and so reading from the file would require a lot of expensive seeks. The other problem was also that the original block size was only 512 bytes – while this helped with reducing internal fragmentation – it made the overhead for transferring data huge as more I/O operations were performed more frequently.

Thus the **Fast File System** was built. It basically kept the same interface but aimed at being more dis aware and thus changed the internal implementation to be even faster.

Organizing the Structure of Disk

FFS divides the disk into a number of *cylinder groups* where a single *cylinder* is a set of tracks on different surfaces of a hard drive that are the same distance away from the spindle. Thus the entire disk can now be viewed as a collection of cylinder groups. Because the drives themselves don't actually export the geometry of location of data, the extended file systems used in Linux organize the drive as block groups which are just consecutive portions of the disk's address space. This grouping into blocks/cylinders is what FFS did to revolutionize the file system game. It allows for files to be placed in the same group meaning the seek and access times for these files would be a lot faster.



To do this FFS allocates the superblock to every group for reliability and integrity reasons, and in addition to the Inode and Data blocks, it allocates a per-group *inode bitmap* and *data bitmap* as mentioned earlier. Which basically contain the location of the respective blocks in that group, while also managing the free space very efficiently.

For directories: FFS first finds the cylinder groups with a low number of allocated directory entries (to balance directories across a group) and a high number of free inodes (for a large number of files) and put the directory data and inode in that group. For files: Make sure to allocate the data blocks in the same group as its inode (prevents long seeks) and place all files that are in the same directory in the same cylinder group as the directory. The results of taking advantage of the locality is presented in the graph of SEER Traces.

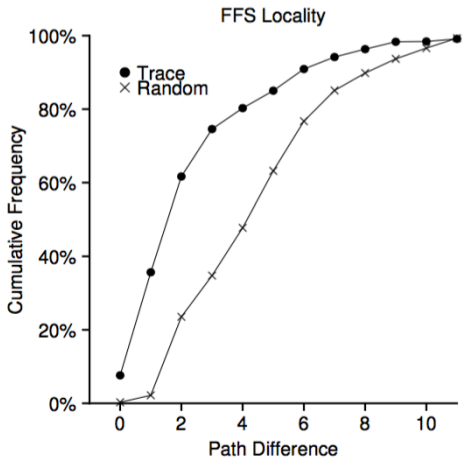


Figure 41.1: **FFS Locality For SEER Traces**

We notice that FFS performs brilliantly for path differences of one (i.e. in the same directory) but gets worse as the difference increases which makes sense. The idea of locality is still maintained due to every file having some common ancestor (such as the root being the ancestor for all directories).

Large File Exception – Large files can sometimes take up the entire group which would not allow for sequential access of files. Thus place chunks of the file amongst different groups. Now obviously this would increase the number of seeks and thus increase the time spent in seeking.
Amortization – Reduce overhead by doing more work per overhead. That is in this case we just increase the chunk size and so that way we make a tradeoff between the seek time (reduces) and transfer time (increases). FFS did this by allocating the first 12 direct blocks in the same group as the inode and each subsequent indirect block and the blocks it pointed to was placed in a different group.

FFS also introduced some other cool stuff. Small files were using blocks that were bigger than their own size and thus we had a lot of internal fragmentation that would waste about half the disk. Thus the designers of FFS introduced **sub-blocks** which were 512 byte little blocks that the file system cold use to allocate to files. As files grew it would allocate more and more sub-blocks until an entire 4KB was used at which point it would find a free 4KB block and copy over all the sub-blocks into it and free the sub-blocks.

However, this method was still inefficient as the it meant a lot of work for the file system and so FFS modified the libc library to buffer the writes and then issue them in 4KB chunks.

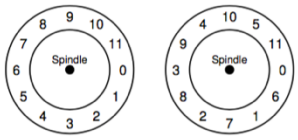


Figure 41.3: **FFS: Standard Versus Parameterized Placement**

Before modern disk drives, FFS optimized the older drives for sequential access by **parameterizing** the data blocks to figure out the specific performance parameters of the disk and use those to decide on the exact staggered layout scheme. As disks got smarter they included their own internal cache called a track buffer form which the sequential reads were done.

FFS was also the first file system that allowed for long file names and it was the first to introduce symbolic links.

Crash Consistency Through FSCK and Journaling

File systems use a lot of data structures for management, but unlike memory they must also be persistent. They must keep the data even in times of power failure or system crash. This is known as **crash-consistency problems**. If we have a crash while an on disk structure is updating, then we leave that data in an *inconsistent* state.

Imagine the situation wherein we have just issued a write to a particular file, there are about three on disk data structures that now need to be updated, that is the inode bitmap, the inode and the data block. The write is buffered into a page/buffered cache that is then issued the write every 5~30 seconds as mentioned earlier. If a crash occurs at this point, then we could have data loss and inconsistency which is never a good thing.

There are several crash scenarios that illustrate the issues we could face. We can get **file-system inconsistencies** if the inode and bitmap are not in sync and if the data block is not written to but we are informed that it has been written to through an update of the inode we could read **garbage** data. We could also get **space leaks** if the inode bitmap is updated but nothing else is, since that block would never be written to. A few possible solutions exist:

1.) **File System Checker (FSCK)**

The basic idea behind this tool is that it will let inconsistencies happen and then run a check at the end to try and fix as many of them as it can. This obviously isn't the best method of preventing all crash inconsistencies but is a step in the right direction. For example, this method wouldn't be able catch the inode pointing to garbage data. FSCK is run *before* the file system is mounted and once it is finished the on disk file system should be consistent.

- **Superblock –** Check if the superblock looks reasonable by doing sanity checks. If an error is found, then the system admin can choose to use a different copy of the super block.
- **Free blocks –** Scan the inodes, indirect blocks, double indirect blocks to understand the currently allocated blocks. From this it creates a correct version of the allocation bitmaps and inodes. The same check is performed for all the inodes to make sure they correspond to that in the inode bitmaps.
- **Inode state –** Check all the fields of the inodes to make sure they make sense and are not corrupted. If a corruption is found we clear the inode and update the bitmap accordingly.
- **Inode links –** Verify the link count of each allocated inode. This indicates the number of directories that contain a reference to a particular file. It does this by scanning through the entire directory tree and counting. If there is a mismatch between this count and the link count in the inode, we must fix the count within the inode. If an allocated inode is found but no directory refers to it, it is moved to the *lost+found* directory.
- **Duplicates –** Check if two different inodes refer to the same block. If an inode is bad then clear it, else if the pointed to block can be copied then give each inode its copy as desired.
- **Bad Blocks –** If a pointer points to something outside its valid range then just clear the pointer form the inode or indirect block.
- **Directory checks –** Perform integrity checks on the contents of each directory such as making sure '.' and '..' are the first two entries and that the inodes refer to the correct directories.

The problem with FSCK is that it is way too slow. As disks grew and RAID grew fsck just became inefficient.

2.) **Journaling (Write-Ahead Logging)**

When updating the disk, before over-writing the data structures in place, write down a little note somewhere else on disk in a well known location about what we are about to do. This section is referred to as a log for obvious reasons. Thus if a crash happens we can look at the log and go back to the point at which we were about to do something before the crash and try again based on the log. Journaling adds a bit of work during updates to greatly reduce the amount of work required during recovery.

Sometimes the journal is placed on a separate device or as a file in the file system, but is sometimes also placed within the file system image itself along with the super block and block groups. In Linux ext3 journaling is made available as a mode. The initial sequence of operations is:

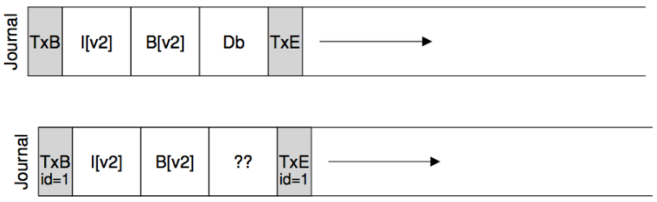
- **Journal write:** Write the transaction and save a **transaction identifier (TID)** with a transaction begin block, all pending data and metadata updates, and a transaction end block to the log.
- **Checkpoint:** Write the pending metadata and data updates to their final locations in the file system.

Note that writing of the pending data and metadata can either be physical by actually copying over the data structures and data or logical as a set of instructions (the latter is more complex).
The issue arises when we have a crash during the write to the journal itself. A simple solution would be to do the writes to the log sequentially but this would make things slow and inefficient. Also if the transaction misses only one of the updates in the log, then things look good to the arbitrary user data and then it hits a corrupt point at which point things could spiral!

The solution employed is to first write everything to the log except for the end of the transaction block, and when those writes complete to the actual data only then do we add the end block. This works for the sake of atomicity in that modern disks guarantee 512 byte blocks to be written atomically. Thus the initial sequence of operations changes to:

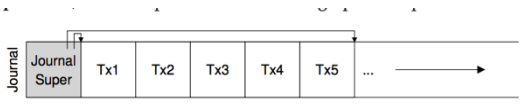
- **Journal write:** Write the contents of the transaction to the log and wait for the writes to complete.
- **Journal commit:** Write the transaction commit block to the log and wait for the write to complete. The transaction is now **committed.**
- **Checkpoint:** Write the contents of the update to their final on-disk locations.

The file system upon booting will go through the journal in order and perform those operations once again – **redo logging**. There maybe some redundant writes if the crash happened during the checkpoint stage because recovery is so rare. If the system crash before the commit, then the pending update is simply skipped.



Journaling systems also avoid excessive write traffic by **batching log updates** into a **global transaction**.

Two problems still remain and they are to do with the size of the log file. If the log file is too long then replaying the transactions can get slow, but also if we write a lot of transactions then the log gets full and we can't have any more transactions saved. Thus the solution is to use the log as a **circular log**. Once a



checkpoint is hit then the corresponding transaction is freed from the log. This can be done by maintaining a journal superblock that maintains the oldest and newest non-checkpointed transactions. The protocol now has an extra step added to it namely:

- **Free:** Mark the transaction free in the journal by updating the journal superblock.

Journaling seems expensive as we now have two writes per normal write! The journaling we've been talking about is referred to as **data journaling**, but a simpler and more common form is called **metadata (ordered) journaling** where the user data isn't written to log itself. Rather only the Inode and bitmap are written. This could create a problem that upon redoing the log the inode may point to garbage data, and so to fix this the data block should be written to disk before the metadata is written to the log. Thus the protocol is now:

1. **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
2. **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now committed.
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
5. **Free:** Later, mark the transaction free in journal superblock.

Another problem with metadata journaling is if we for example create a directory (which is just metadata) and then we delete the entire directory and then create a file in its place, the journal log only contains information about the directory (since it is metadata), upon redoing the log, we would overwrite the file with what's inside the log and thus create issues. One popular solution is to hold a **revoke record**. In the case above, deleting the directory would cause a revoke record to be written to the journal. When replaying the journal, the system first scans for such revoke records; any such revoked data is never replayed, thus avoiding the problem mentioned above.

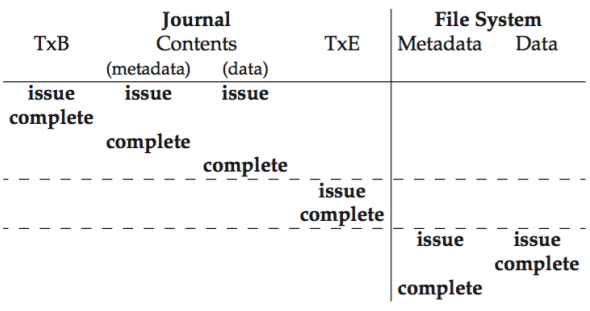


Figure 42.1: **Data Journaling Timeline**

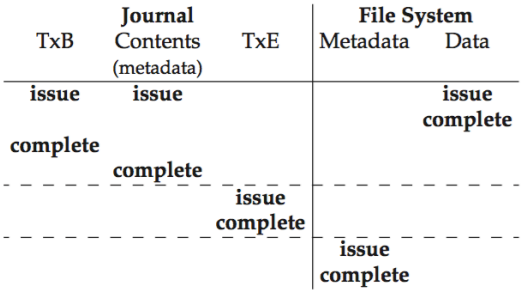


Figure 42.2: **Metadata Journaling Timeline**

3.) Other Approaches

Soft Updates:

Carefully order all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state. By writing a pointed-to data block to disk *before* the inode that points to it, we can ensure the inode never points to garbage. This however requires intricate knowledge of the exact file system structures.

Copy On Write:

Never overwrite files or directories in place but rather place new updates to previously unused locations on disk. After a number of updates are completed, just flip the root structure of the file system to include pointers to the newly updated structures.

Backpointer-based consistency:

No ordering is enforced between writes. An additional **back pointer** is added to each block. The file system can determine if the file is consistent by checking if the forward pointer points to a block that refers back to it.

Optimistic Crash Consistency:

This issues as many writes to disk as possible and uses a generalized form of the **transaction checksum** to detect inconsistencies.

Data Integrity and Protection:

When we spoke about RAID we mentioned that it faced one type of failure modes, namely **fail-stop** which was relatively straightforward as it only had to states to worry about. However, modern day disks have other types of failure modes too, of which two are key to the discussion of data integrity which fall under the **fail-partial** disk failure model:

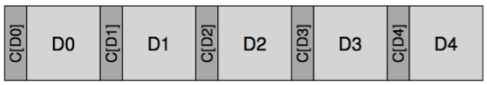
1.) **Latent Sector Errors (LSEs) –** The disk head touches the surface for some reason (**head crash**) and thus damages the data bits on the surface which will in turn return an error as to the data being inaccessible.
2.) **Block Corruption –** Buggy disk firmware may write a block to the wrong location and thus everything seems to be working fine from the disk's perspective but the user gets the incorrect information and thus this error is silent.

Error Correcting Codes (ECC) are used by the drive to determine whether on disk blocks are good and try to fix them. This is what causes the errors that are raised by LSEs.

LSE's are pretty straightforward to fix because they are easily detected. Based on the redundancy system used we just look into the copy or reconstruction to get the data back. If the RAID level 1 (mirroring) was used, then we just use the alternate copy; if the RAID level was 4/5 then we can reconstruct the damaged data from other blocks in the parity group. This LSE has influenced RAID design. When an entire disk fails the RAID tries to **reconstruct** the disk by reading through all the other disks, but if an LSE is encountered on one of these disks during this process then we can have issues. **RAID-DP** basically has two parity disks instead of one to

help with the reconstruction, but can take a long time (or use the **WAFL** file system) and can take lot more space.

Detection of corruption is the key to solving the problem since it is a silent failure. Once the error is detected then the same recovery mechanisms can be used. Modern storage uses a **checksum** – take a chunk of data and computes a function over this data and produces a small summary of the contents of the data. This can be used to compare new incoming values to previously computed checksums for the same data.

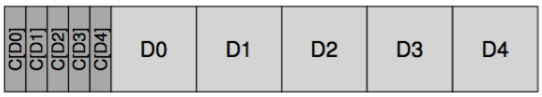


There exist multiple types of checksum functions varying in their efficiencies. The more protection we get the costlier the checksum.

1.) **XOR –** Align the data bits and xor them in groups to retrieve the checksum. This is pretty efficient and fast but has an issue that when two bits in the same position within each checksummed unit change, the checksum will not detect the corruption.
2.) **Addition –** This is fast and just requires 2's complement addition to be applied over each chunk of data ignoring overflow. This fails if the data is shifted however.
3.) **Fletcher checksum –** Computes two check bytes. If a block contains bytes *d1…dn* then $s1 = s1 + d_i \bmod 255$ and $s2 = s2 + s1 \bmod 255$. This is almost as strong as the next function and is shown to be good at detecting all single bit errors, double bit errors and a large percentage of burst errors.
4.) **Cyclic redundancy checks (CRC) –** Take the bits of a data block *D* and divide it with an agreed upon value. The remainder is the checksum. This is very efficient to compute and is pretty secure which is why it is so popular.

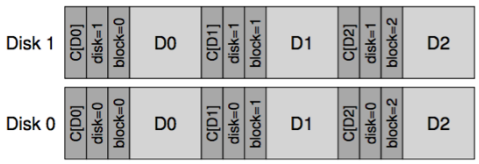
Collision – Two data blocks with non-identical contents will have the same checksum.

For the most part checksums are stored for each data block just before the block itself, with most drive manufacturers formatting the drive with 520 byte sectors with 8 bytes per sector for the checksum. Some disks that don't have this functionality store the *n* checksums as a sector followed by *n* data blocks. This can be slightly costlier with one read of the checksum and then an update of it in the sector, but it works on all disks.



But how are checksums actually used? When reading a data block, we also read the associated checksum known as the **stored checksum** and then we compute the checksum over the block and call it the **computed checksum**. If the two are equal the data has most likely not been corrupted, and if they don't match then we have a corruption.

These all work well for corrupted blocks but don't solve the problem of **misdirected writes** which write the data to disk correctly but to wrong locations. To fix this we simply add a **physical identifier (PID).** Thus each block will have an associated checksum and disk and sector number of the block. This is now helpful in finding the correct locations based on redundancy mechanisms that save this information.



The final problem we must consider is the **lost write** which occurs when the device informs the upper layer that a write ahs completed but in fact has never persisted. Checksums don't work here because the old block that was meant to be over written still has a valid checksum for that physical location. One solution is known as **write verify (read-after-write)** which immediately reads the data after a write to ensure that the information on the disk is correct and up to date. Some other solutions include adding a checksum elsewhere in the system.

Disk scrubbing – Go through every block on the system and check whether their checksums are still valid. This ensures data integrity and reduces the chances of corruption happening.

Checksums can add *space overhead* because they take up small amounts of space in the disk, and can take up memory space as well if they are stored for more accessing and checking! They also add a larger *time overhead* because the computations over the entire block for both storage and access can be costly. One approach to reducing CPU overheads, employed by many systems that use checksums (including network stacks), is to combine data copying and checksumming into one streamlined activity; because the copy is needed anyhow (e.g., to copy the data from the kernel page cache into a user buffer), combined copying/checksumming can be quite effective. They may also add I/O overheads if they are distinctly stored from the data. This can be fixed through background scrubbing during low work times.

Flash based SSDs

Hard disks are things of the past nowadays. There are faster and more reliable means of data persistence through transistor based **solid state storage**. The technology mainly used is known as **NAND-based flash**. **Flash page –** The area we write to; **flash block –** Usually the container of the pages, which must be erased before every write. Writing too often can cause it to **wear out**.

There are multiple ways the information is actually stored:
Single-Level cell (SLC) – A single bit is stored within a transistor.
Multi-Level cell (MLC) – Two bits are stored by computing low, somewhat low, somewhat high and high charges.
Triple-Level cell (TLC) – Encodes 3 bits per cell.

It is found that SLC performs the best and are generally more expensive.

Flash chips are organized into **banks** or **planes** which consist of a large number of cells. This bank/plane can be accessed in two different sized units:

1.) **Blocks (erase blocks)** – generally 128KB or 256KB
2.) **Pages** – few KB in size such as 4KB

Within each bank we have a large number of blocks and within each block we have a high number of pages. There are three low level operations that a flash chip support:

1.) **Read (a page) –** Specify the page number of the device and the rad of the page shall be done extremely quickly and uniformly across the device. This makes the device a **random access device.**
2.) **Erase (a block) –** Before writing to a page, the nature of the device requires for one to first erase the contents of the block by setting all the bit values to 1. Thus if important data exists here we must copy it to another location before the erase. This command is somewhat costlier.
3.) **Program (a page) –** This command can be used to change some of the 1's within a erased blocks page to 0's and write the desired contents of a page to the flash. This is costlier than the read but cheaper than an erase. Note that you can only read from a page that has been programmed.

A write can be thought as a combination of erase and programming, but continual writes wear out the flash chip and thus present us with a problem of data reliability.

Because flash chips don't have mechanical components as did hard drives, there aren't errors such as head crashes to worry about. One reliability problem to worry about is the *wear out*. This happens through continuous writes which accrues a bit of charge in that location. Another reliability issue to worry about is known as **disturbance –** When accessing a page it is possible to flip some of the neighboring bits which result in **read disturbs or program disturbs**.

SSD's are based on flash storage and essentially have to provide the convenience of integrating with the existing interface while using flash storage to store data. The SSD contains a number of flash chips for the storage, some volatile memory (SRAM) for caching and buffering and for mapping tables; it also contains some control logic to be able to perform the operations.

Flash Translation Layer (FTL) – This is the essential control logic within the SSD. It takes read and write requests on logical blocks and turns them into the low level read, erase and program commands on the physical blocks and pages.

The excellent performance comes about from a combination of techniques such as **Parallel** flash chip usage. We must also reduce the **write amplification –** total write traffic issued to chips by FTL/ total write traffic issued by client to SSD. The FTL must also try to spread the writes across the bocks of the flash as evenly as possible to ensure for similar wear out times. This is

known as **wear leveling**. To reduce disturbance, the FTL will commonly program pages in order from low page to high page. Thus there are multiple FTL organizations:

1.) **Direct Mapped –** Read in logical page N and map it to physical page N. For the write, we first read the physical page, erase it and then program it. This can be even slower than hard drives. Also there is no distribution of the writes and so we can wear out the blocks a lot faster and thus this is a very bad approach.

2.) **Log structured –** For writing a block N we append it to the next free spot in the *currently-being-written-to-block*; and for subsequent reads of block N we have a **mapping table** that stores the physical address of each logical bock in the system. **Logical Block Address –** the address used by the client of the SSD to remember where information is stored. This is then converted to physical page number through the *in memory mapping table*. This table is updated upon every fresh write to a new block in the SSD, and then on subsequent reads the translation happens pretty quickly. This method also allows the FTL to spread writes all across pages and less frequently through logging. However, overwrites of logical blocks can lead to **garbage** – old versions of the data remain around the drive and take up space. Thus the device has to periodically perform *garbage collection* to find these blocks and free up space for future use. We also note problems with storage of the in memory mapping table.

- **Garbage Collection –** Find a block that contains one or more *garbage pages*, read in the live pages from that block and write them out to the log and reclaim the entire block for use in writing. There must be enough information to enable the SSD to determine if a page is live or dead. We can do this by storing at some location in each block, some information about which logical blocks are stored within each page and then we use the mapping table to determine the state of the data. This can be expensive with the reading and rewriting of data. To reduce these GC costs, some SSDs add extra flash capacity and this allows for the cleaning to be pushed into the background.
- **Mapping Table Size –** This can aim to be an issue with large SSD's. There are two main methods of mapping:
    i. **Block Based Mapping –** Keep a pointer per block of the device instead of per page. This is known as **block level** FTL while the latter is known as **page level** FTL. This is akin to having bigger page sizes in virtual memory. This is however, not as efficient as one may think in a log based FTL in the case of small writes (a write less than the size of the physical block). In such a case the FTL must read a large amount of live data from and old block and copy it into a new one which increases the write amplification. It does however make reading easier since it simply extracts the chunk number by taking the top most bits out of the address, then looks up the chunk number to physical page mapping in the table and then computes the address of the desired flash page by adding the offset from the logical address to the physical address of the block.

ii. **Hybrid Mapping –** A few blocks are kept erased and the FTL directs all writes to them – **log blocks**. For these blocks there exist a per page mapping. Thus the FTL has two types of mapping – a small set of per page mappings in the log table and larger set of per block mappings in the data table. The FTL will first consult the log table and if not in there then it will consult the data table. The key to this strategy is to keep the number of log blocks small. This is done by periodically examining log blocks and switching them into blocks that can be pointed to by only a single block pointer. **Switch Merge –** blocks that have been written in the same manner as before can use old blocks as log blocks. This is the best case. **Partial Merge –** It does the same thing as the switch merge but the case is that we have to perform extra I/O to do so. **Full Merge –** Pull together pages from many other blocks to perform cleaning.

The Log based FTL must also periodically read all live data out of blocks and re-write them elsewhere for wear leveling. It increases the write amplification in the process but keeps the SSD around for longer.

Because SSD's exhibit some performance difference between sequential and random I/Os the techniques used by the hard disks for file systems still play an important role. SSD's are slowly replacing HDD's but are doing so slowly because of the cost difference. Its only a matter of time before they replace HDD's for local personal computers.

Wednesday:

Information Security

Key points in making a system secure are:

1.) Authenticity
2.) Integrity
3.) Authorization

Checking all of the above for every requested action is known as **Complex Mediation**.

With the prevalence of networking, attacks can come from anywhere and it is pertinent to create a robust system that can protect from local and remote adversaries, while at the same time creating an authority structure.

A common goal in a secure system is to enforce some privacy policy.

There are three broad categories of threats:

1.) **Unauthorized Information Release –** Access a user's thing without their permission
2.) **Unauthorized Information Modification –** Modify a user's thing without their permission
3.) **Unauthorized Denial of Use –** Prevent user from accessing their own thing

Security is a negative goal, in the sense it is easy to prove the inverse of a secure system (insecure – just find a flaw) but hard to prove the actual security as one would have to think up of all the possible cases and prove that the system can defend against them, which is extremely difficult.

When designing a secure system the designer must be explicit so that all assumptions can be reviewed, and he/she must design for iteration, so as to build up on creating more robust systems by continually building. The three things the designer should do are:

1.) Certify the security of a system by a third party/formal tools.
2.) Maintain audit trials of all authorization decisions so that we can use it for forensics after an attack.
3.) Design the system for feedback so that when it is attacked we can get back the reason as to it.

Designers of the system must also take into account the environment of use of the system (network, local, disgruntled and weak fucking employees that can be bribed etc.), the dynamics of use (being able to change permissions) and must always assume the system is not robust enough and must design for depth, continually making more and more security measures.

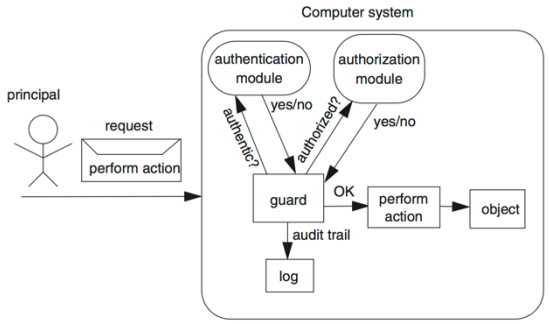


FIGURE 11.2

The security model based on complete mediation. The authenticity question includes both verifying the integrity and the source of the request.

Open Design Principle – Let anyone comment on the design (especially professionals in the field). You need all the help you can get.
Minimize Secrets – They probably wont stay secrets for as long. Doing so abides by being explicit and leaves fewer surprises when you get fucked.
Economy of mechanism – The less there is, the more likely you will get it right. Because there is so much to consider and protect against, keeping the design simple can go a long way.
Minimize common mechanism – Shared mechanisms provide unwanted communication paths.

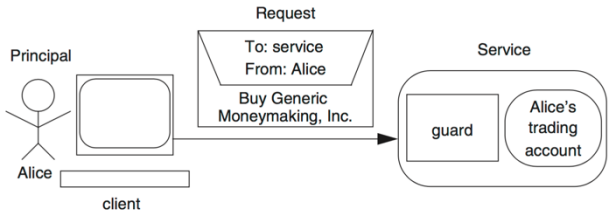


FIGURE 11.3

Authentication model.

Fail-safe defaults – Most users wont change them, so make sure the defaults do something secure.
Least privilege principle – Restrict access to as much important/safety critical information as you can, and if you must give access, make sure to re-lock this safe once its done.

There exist many ad hoc security implementations around because when systems were built they did not anticipate adversaries, and so built them without security. Such as the Internet and the personal computer. This is why it is important to learn from history and use foresight to see potential security flaws in your design before you implement them.

When trying to mediate every action to make a more secure system we ask the question "Is the principle who requested the action authorized to perform said action?". From this the system then asks a guard (**reference monitor**) whether it should or shouldn't perform said action. The guard then verifies the authenticity of the request, if valid then it performs it, else it rejects it. In both cases a log is maintained.

Denial of Service – Flood a communication link with continuous requests thus preventing service to other users, and it can thus overload a server and crash the system. Hard to protect against.

Trusted Modules – Modules whose working are pertinent to the security of the system (like privileged instructions) that are stored in a **trusted computing base (TCB)**. This should be kept small and simple so that we don't promise as many guarantees and restrict service to such a large extent.
Untrusted Modules – The failure of these modules don't damage the security of the system. This is like user mode code in UNIX.

The TCB can be built by: Specify security requirements -> Design a minimal TCB -> Implement the TCB -> Run TCB and try to break security.

When it comes to authentication, there must be a service of trust that forms, through some sort of identifier. When a guard tries to verify the authenticity of a user it does so through two stages:

1.) A rendezvous step, in which a real-world person physically visits an authority that configures the guard, might use a physical trait like fingerprint or a physical object like ID or some unique information such as a PIN.
2.) Verification of identity through an agreed upon identifier.

In terms of the actual means within the computer, passwords given by the user are put through a **cryptographic hash function** that maps it to a unique key within the database of the system. It maps an array of *M* bytes to a fixed length value *V* through *H(M)*. If this function is good then it is hard to get collisions using around 160-255 bytes of length. Also if you have the output of the function, you can't reverse it! The most popular ones are variations of **Secure Hash Algorithm (SHA)**. This hashing is secure because the work factor (amount of time and processing power that would be needed to break it) is several years due to the insane number of combinations you can get! This would all be void if something like quantum computing became more prominent.

Methods of breaking such passwords include brute force attacks or *dictionary attacks* which gather all the information it can about the person and then use that information to generate more likely passwords and compute their hash.

We must minimize repeated use of a secret. A good protocol between the user and service must have the following:
1. it authenticates the principal to the guard;
2. it authenticates the service to the principal;
3. the password never travels over the network so that adversaries cannot learn the password by eavesdropping on network traffic;
4. the password is used only once per session so that the protocol exposes this secret as few times as possible. This has the additional advantage that the user must type the password only once per session.

It is always better to use well tested and prominent algorithms that have been tried and tested, but one must be wary that with the rapid change in technology, these older algorithms could suddenly crumble under newer innovations.

When we wish to keep confidential information, confidential, then we must apply stringent protocols to prevent data communication. One such method is used by military such as employing a trusted and untrusted network, and the only way to communicate is through manual transfer. Look up STUXNET for an attack that got around this. Its fucking insane.

There are three primary operations in authorization systems:
1.) **Authorization –** Grant a principal the permission to perform an operation.
2.) **Mediation –** Check whether said principal has the permission to do said operation.
3.) **Revocation –** Remove a previously granted permission if need be.

The principal that makes authorization and revocation is the authority and the guard is the mediator in between.

Simple Guard Model

Use an *authorization matrix* of principals as rows and objects as columns with the data being the permissions. Most models assume that if a principal creates an object then it is the authority for said principal.

Discretionary access-control systems make the creator of the object the authority and allow the creator to change the permission entries at the creator's discretion. The creator can specify the initial permission entries as an argument to the create operation or, more commonly, use the system's default values. Non-discretionary access-control systems don't make the creator the authority but chose an authority and set the permission entries in some other way, which the

creator cannot change at the creator's discretion. In the simple guard model, access control is usually discretionary.

Ticket system:

Each guard holds a ticket for each object it is guarding, a principal holds a separate ticket for each different object its authorized to use. Compare these tickets with a ring with keys. The tickets determine which objects the principal can use. When another principal wishes to gain authorization, it is given a matching ticket. To revoke we either, hunt down and take the ticket back or change the guards ticket and reissue tickets.

List System:

Each principal has a token identifying the principal and guard that holds a list of tokens that correspond to the set of authorities. For mediation the guard just searches through a list of tokens to see if the principals token exists. To revoke access, the authority just removes the principals token form the guard's list. This list of tokens is called **Access Control List (ACL)**. A lot of ACL's also implement a *protection group* which are principals that may be used by more than one user (think g in UNIX).

Table 11.1: Comparison of access control systems

System	Advantage	Disadvantage
Ticket	Quick access check	Revocation is difficult
	Tickets can be passed around	Tickets can be passed around
List	Revocation is easy	Access check requires searching a list
	Audit possible	
Agency	List available	Revocation might be hard

In UNIX principals are users and groups. User's names are stored as fixed length UID's along with the password in a file. Groups are protection groups in UNIX, where all like users belong to the same GID. Super user has UID 0. UNIX stores a 3 entry long ACL for each file in its Inode, describing its UID of owner, GID and other of that file. Web servers running under UNIX operate a little more stringently with more access restrictions than those enforced by ACL. Per ACL entry we have the *rwx* bits for each principal type. The *setuid* can be used to change the permissions of a principal, but this can make the system susceptible to entering superuser mode and never coming back thus violating the least privileges rule. When a password is entered and confirmed against a cryptographic hash of it during login, the super user stores it in a shadow file that can only be written to using setuid. We see UNIX switch between the ticket based guard system (login program) to a list-oriented system (the kernel and the file system), while remaining a complicated example of the simple guard model.

Caretaker Model

This generalizes the simple guard model in an object oriented fashion. It verifies permissions for arbitrary methods and can enforce some constraints on access to an object. An example would be a box that can be opened when two principals agree. Basically this model is harder

to implement and thus can have more pitfalls. The previous discussion was one of discretionary model, whereas this one is of non-discretionary, where the resources are compartmentalized and each section is given different levels of access for a certain amount of time, that is all dependent upon one user's decisions (think military with classified documents). Such methods can also use *confinement* to prevent certain programs from using external sources, and keeping it isolated to its specified subset. Each principal is given access to a certain number of compartments.

Complete confinement of a program with shared resources is difficult, which brings about *covert channels* which are hidden communication channels through which information can flow unchecked. Finding these and blocking them are proven to be extremely difficult.

When it comes to protecting data on storage devices, options to encrypt the data (such as through Apple's File Vault) can be used. Upon boot time the user is prompted with an encryption password that if valid encrypts the data automatically upon every write to the storage, and is decrypted for use whenever necessary. Such forms of protection aren't useful upon one's own machine. It can't protect from buffer overflow attacks or adversaries posing as super users, and this cryptography is an expensive operation. The advantage it does give is that other OS's won't be able to access the data. That is without you at the helm of a computer, anybody who gets this encrypted storage device, won't be able to access it. *Homomorphic cryptography* allows for certain data operations to take place without decryption. Another important case of *at-rest security* is password vaults (such as keychain in Apple) which is encrypting passwords and login details by a user entered password (that could be stored at RAM).

WEEK 9

Monday:

Distributed Systems:

There are several reasons as to why we built distributed systems, with a large part of the driving factor being the ability to collaborate with others in similar work. Other reasons include:

1.) Client/Server Usage Model – Equipment I may need can be used through access to a server. I can use the server as a Network Attached Storage (NAS) so that I can always access things from anywhere, along with giving others access to it as well.
2.) Reliability and Availability – By storing data through distributed independent servers we see that we have *no single point of failure* and so our data can be stored across multiple sources and can be safely accessed!

3.) Scalability – Build a system that starts out small, but upon success gained, incrementally increase its ability to store more and more by simply adding more computers (servers) and storage. This is scaling out.
4.) Flexibility – Through distributed systems, we can test on various types of machines.

But distributed systems are not easy to build. The reasons as to why they're not easy to build are as follows:

1.) New and More Modes of Failure – One node of the system can crash and others could continue to run normally, some network messages may be delayed or lost, a switch failure may interrupt communication between some nodes, but not others. With such systems, there are many possibilities as to the failure, such as:
    a. Node failed
    b. It is slow
    c. Network link failed
    d. Our own network interface has failed

    Diagnosing the problem correctly is important. We must make sure that when a component fails, we break any resource locks it was holding, and then restore the resources to a clean state and prevent the previous owner form attempting to continue using said resource.
2.) Complexity of Distributed State – With single systems we can place all operations serially and we can with confidence say what the state of every resource on the system is. However, because distributed systems have distinct nodes that operate independently, serializing communication becomes difficult through different nodes sending messages back and forth. The state is a vector of the state that the resource is considered to be in by each node in the system. Normal single system locking mechanisms for such parallel events such as mutexes may be far too expensive on distributed systems, and so new locking mechanisms need to be created.
3.) Complexity of Management – Each system in this distributed system may be configured differently with different databases, services of different options etc. And creating a distributed management service to push management updates to all nodes, some nodes may not be up during the push and networking issues can create isolated islands of nodes that are operating with a different configuration.
4.) Much Higher Loads – As the load increases, new and unknown weaknesses are uncovered, and new bottlenecks may be found. More nodes means more messages which has higher overhead which means weaker scaling and performance drops, and longer delays may result in race-conditions which may have otherwise been unlikely.
5.) Heterogeneity – Each node may have different ISA's, different OS's and different versions of software and protocols. The high number of combinations and constant evolution of possible component versions and interconnects render exhaustive testing impossible, creating issues in interoperability.

6.) **Emergent Phenomena –** New behaviors come about from trying to work with more complex systems, and learning to understand them comes through hard experience that may result in failures.

So now that we have touched upon the tip of the problems that can arise form distributed systems, we must ask ourselves. How exactly do distributed systems work? It comes in the form of **communication**. Said communication is undeniably unreliable over a network. Some bits could get flipped during transmission, some packets are lost or corrupted or network cables get damaged.

One of the most fundamental of these issues are packet loss due to a lack of buffering within a network switch, router or endpoint. When a packet arrives at a router, it must be placed in memory within the router for processing. If we receive a high influx of packets, then it is possible the router may not be able to accommodate all the the packets and will have to **drop** some. This is an example of overwhelming the machine's resources that leads to packet loss. So how should we deal with such loss of packets?!

1.) **Don't deal with it –** Some applications are already built to deal with packet loss, and so it may be useful to just let them communicate through an unreliable messaging layer (this is an example of the **end-to-end argument**). The most famous of these layers is the **UDP/IP** networking. A process will use the *sockets* API to create a *communication endpoint*, while processes on other machines will send UDP **datagrams** (fixed-sized message to a max size) to the original process. With UDP packets get lost and so the sender is never informed about this loss and the receiver can't do anything about it. However, UDP does include a **checksum** to detect some forms of packet corruption.

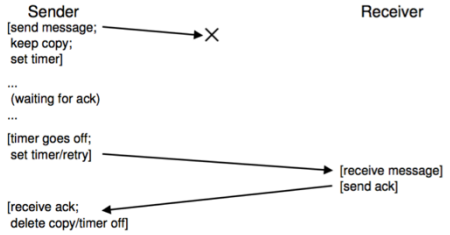


Figure 47.4: **Message Plus Acknowledgment: Dropped Request**

2.) **Reliable Communication Layers –** To build a reliable system, we need to be informed about all the sending and receiving of packets. It is important for the sender to know when a packet has been received. This is done through an **acknowledgement (ack)**. When a sender sends some piece of data the receiver sends back a short message (ack). The system also has a **timeout/retry** method wherein, the sender sets off a timer after it sends its message (while holding a copy of the message as well), and if no ack is received in the time interval specified, the sender assumes the packet was lost and thus tries again by sending the copy of the original message. However, it is imperative that the message is received **exactly once** since we don't want duplicate data (such a situation may arise if the ack from the receiver is not sent back even though the data was received). One way of doing this with little memory usage is **sequence counting**. This is where the sender and receiver will agree on a counter value and pass it

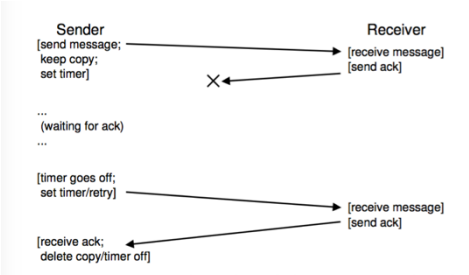


Figure 47.5: **Message Plus Acknowledgment: Dropped Reply**

back with the message. If the counter value received is the same then we know we've received a duplicate and just send back an ack but not pass the data to the application. The most well known reliable communication is **TCP/IP** which has a lot more sophistication such as congestion control.

Distributed Shared Memory (DSM) – Allow processes on different machines to share a large virtual address space – which makes the system seem like a multithreaded application, but rather over different machines. When a page is accessed on one machine we can have one of two things happen. In the best case the page is already local to the machine and thus the data is fetched quickly. The second, a page fault occurs and the page fault handler sends a message to some other machine to fetch the page, install it in the page table of the requesting process and continue execution.

However, DSM isn't actually in use in modern systems today, because of many inefficiencies. When machines fail, through DSM we could have portions of the address space get lost! Also making these page requests amongst remote servers can become pretty expensive. DSM programmers had to carefully organize computations so that almost no communication occurred which kind of defeats the purpose of such an approach.

Remote Procedure Call (RPC) – Make the process of executing code on remote machines as simple and straightforward as calling a local function. The server simply defines some routines it wishes to export, and the RPC system handles the rest through its two key pieces – **Stub Generator** and **Run-time library**.

Stub Generator – Remove some of the pain of packing function arguments and results into messages by automating it. The input to this compiler may just be a simple interface, and the stub generators will generate a **client stub** for the client which will contain this interface code for the client program, and then when it wishes to use the RPC service, it would link with the client stub and call into it in order to make RPC's. The function in the client stub will internally do something like:

1.) **Create a message buffer –** Contiguous array of bytes of some size
2.) **Pack needed information into the message buffer –** Things like identifiers for the function to be called, along with arguments for the function. The process of putting this information into a single contiguous buffer is known as **marshaling** of arguments or **serialization** of the message.
3.) **Send the message to the destination RPC server –** communicate with the RPC server through the run-time library.
4.) **Wait for the reply –** Because function calls are usually synchronous.
5.) **Unpack return code and other arguments –** Grab the results of the function's return. This process is known as **unmarshaling** or **deserialization**.
6.) **Return to the caller –** Return back into the client code from the stub.

For the server side, a stub is also produced that does things like:

1.) **Unpack the message –** Take information out of the incoming message.
2.) **Call into the actual function –** Execute the actual remote function through the ID and pass the desired arguments.
3.) **Package the results –** Put the results into a reply buffer.
4.) **Send the reply –** Send the reply back to the client.

Most of these servers are constructed in some sort of concurrent fashion such as **thread pools**, wherein a finite number of threads are created when the server starts, and when a message arrives it is dispatched to one of these threads which does the work of the RPC call, while a main thread keeps receiving other requests and dispatching them to other threads. This adds parallelism, but includes complexities such as the need for locking mechanisms for the RPC calls.

Run-time Library – this does much of the heavy lifting of an RPC system. One of the first challenges that we must overcome is how to locate a remote service. This is a **naming problem** and is common in distributed systems. In such a system, the client must know the hostname or IP address of the machine running the desired RPC service, as well as the port number it is using (a port number is just a way of identifying a particular communication activity taking place on a machine, allowing multiple communication channels at once). Once a client knows about which server to talk to, it decides on a transport-level protocol. One would think TCP/IP the reliable communication would be the one of obvious choice, but it turns out that with the way TCP/IP works along with RPC, the inefficiencies are huge (sending two extra messages in the ack process), and so UDP is chosen instead. Thus UDP is chosen, along with some modifications to allow for serialized and more reliable communication – that similar to the timeout/retry method. The RPC run-time library also has *explicit acknowledgements* that let a client know that the server is still working on a request through periodic questioning. It is also necessary that sometimes the run-time should handle calls with arguments larger than that that can fit into one packet. This can be achieved through **sender-side fragmentation** and **receiver-side reassembly**, by breaking larger packets into smaller ones and then reconstructing them. With distributed systems we must also pay attention to the endianness of the system. RPC packages often handle this by providing a well-defined endianness within their message formats. If the endianness is the same, then just handle it like normal, else make the relevant conversion (this is what the **eXternal Data Representation XDR** does). Finally, typical RPC's are made **synchronously** but because this can take long it also allows you to invoke an RPC **asynchronously**, where the package sends the request and returns immediately so that the client can continue work, and then when the client needs to see the response of this asynchronous RPC it calls back to the RPC layer and tells it to wait for any outstanding RPCs to complete.

When dealing with communications over the network, we must make sure that the message is reliable and authentic. That is the guard of the receiving application should be able to verify that the message is authentic. This can be done through chaining messages to make sure its

coming from the same source as previous messages. Generally, in message authentication there are two closely related matters we must pay attention to:

1.) Data integrity to ensure the data has not been changed since it was sent
2.) Origin authenticity to make sure the message is coming from the same learnt source as it was before.

Some people mistake authentication and confidentiality, when in fact they are different. There are four important combinations of the two:

1.) Authentication and confidentiality (like in electronic banking)
2.) Authentication only (like DNS)
3.) Confidentiality only (this is uncommon as it is generally important to know who sent the message when regarding its confidentiality)
4.) Neither of the two (used only if there are no intentionally malicious users or separate code of ethics)

Plaintext/Cleartext – Content that is publicly readable when used with cryptography.

When sending messages, we assume the communication path is insecure and that wire-tappers can intercept information that is being communicated. It is thus important to make sure that the data being sent is SIGNED and VERIFIED correctly. That is we need to secure the signature and the data being sent so that even interceptors would not know how to decipher the data. With *closed design principle* we make everything a secret, but this then increases the risk in a compromise in that we would have to change the entire system to fix the break. Instead *Kirchhoff's open design* states that we only use *cryptographic key* to make the signature and verification secret.

We have two models through which we can create *authentication keys*. This is through shared-secret cryptography (one of the keys can be computed from information about the other key) and public-key cryptography (one of the two keys is kept secret while the other is kept public). The latter is more frequently used for the simple fact that it has to keep fewer secrets which make life a little simpler.

We can think of sign basically generating a tag with the associated message using the first key. Verify will then take the received message, tag and use the second key to decipher it and verify it. If the computation returns ACCEPT then the message and its tag match, else a REJECT is generated. Sometimes if the message is long then it may be less expensive to verify the cryptographic hash of the message instead of signing it.

Shared Secret Key *K* must guarantee the following:

1.) Verify returns ACCEPT if the received message is equal to the original message and the received tag is equal to that generated by Sign.
2.) Without knowing *K* it is difficult for an attacker to compute the correct messages for successful verification.

3.) Knowing the message, tag, algorithms for SIGN and VERIFY shouldn't allow the attacker to compute the key.

Public Key Authentication systems require the following:

1.) The same as (1) for shared secret key model, but using different keys in verify and sign.
2.) Without knowing K_1 it is difficult for an attacker to compute the correct messages.
3.) Knowing everything other than K_1 shouldn't allow for the attacker to compute this key.

The length of the keys is generally proportional to the work factor in compromising SIGN and VERIFY. The longer the keys the better. The famous RSA-public key cipher is typically 1024 or 2048 bits, while with shared-key its about 128 to 256 bits which generate the same work factors. It is also advisable to regularly change the keys to limit the damage in case a key is compromised.

Generally speaking, attacks on authentication systems tend to fall in five categories:

1.) Modifications to the message and tag.
2.) Reordering the message.
3.) Extending the message by prepending or appending information.
4.) Splicing several messages and tags.
5.) Directly attack Sign and Verify since they are based on cryptographic transformations.

Window of validity – Minimum time to compromise the signing algorithm, the time to compromise the hash algorithm, the time to try out all keys and the time to compromise the signing key.

With shared secret authentication, the signing of the message and verification of the tag uses the same shared secret key. This authentication tag computed in this way is often called a message authentication code (MAC).

With public key authentication, the sender signs the message with their private key, and the receiver verifies the tag with the sender's public key. An authentication tag computed in such a way is known as a digital signature.

It is easy to *repudiate (disown)* a message authenticated through MAC since you can just say that the person you shared the secret with, forged the message. However, with a digital signature, the only way to forge the message is if another party knows the sender's private key, which is less likely. This is why it is more appropriate in electronic checks and banks.

When we talk about verification of keys we must recognize that the key must be securely distributed, otherwise an attacker can pretend to be someone they are not. This problem is sometimes known as *name-to-key binding* problem. Secure key distribution is based on a name discovery protocol, which starts, perhaps unsurprisingly, with trusted physical delivery.

With cryptographic keys, we distribute them over the network, and nowadays the cryptographic systems are strong enough to do so, without fear of compromise.

Sometimes third party services are used in the key distribution in order to get either the public key or the shared secret key to the receiver in an authenticated manner. SSH just installs the public key into the receiver's system if the user agrees to do so. The message that asks you to trust the system counters an attack wherein an attacker can return an IP address to the attacker's computer when the DNS lookup for the address you are sshing into is instantiated by intercepting it.

Sometimes as we mentioned before, it is also important to make the conversation between two people confidential. The methods are generally the same. We wish to **encrypt** the plaintext into ciphertext and **decrypt** the message back into plaintext. This can be done through cryptographic transformations using *encryption keys*. This should be done in such a way the attacker cannot decrypt the message if intercepted. This is done through the ENCRYPT and DECRYPT primitives.

The shared-key encryption is similar to the shared-key authentication model, where the message is encrypted using the same shared key.

With public-key encryption, the encryption is done based on who is sending the message. The message is encrypted using the receivers public key and the receiver decrypts the message using their own private key.

Implementation of ENCRYPT and DECRYPT should be good enough to block against the following attacks:

1.) Ciphertext-only attack (the attacker has examples of ciphertext and algorithms for the primitives, and through repeated patterns or redundancy in the original message we can get the plaintext through brute-forcing the patterns)
2.) Known-plaintext attack (attacker has access to ciphertext and plaintext that corresponds to some part of the ciphertext)
3.) Chosen-plaintext attack (attacker has access to ciphertext that corresponds to plaintext chosen by the attacker)
4.) Chosen-ciphertext attack (attacker selects a ciphertext and then observe the message that results upon the decryption)

Increasing the key length can generally increase the work factor for the attack on the encryption.

REST:

Representational State Transfer is an architectural style that comprises of six constraints to improve performance, scalability, simplicity of interfaces, modifiability of components, visibility of communication, portability and reliability. The constraints are as follows:

1.) Client-server (separation of the two from each other)
2.) Stateless (no client context stored between requests. Each request contains all the information necessary to service the request and the session state is held in the client.)
3.) Cacheable (responses must be cacheable or not to prevent clients from reusing stale or inappropriate data in response to further requests)
4.) Layered System (client cannot tell whether it is directly connected to the end server or to an intermediary which helps in scalability.)
5.) Code on demand (Servers can temporarily extend or customize the functionality of a client by the transfer of executable code.)
6.) Uniform interface (Individual resources are identified in the requests like URI's and the resources are separate from the representations of the data sent to the client such as JSON. The metadata should be enough to modify the resource. Each message includes enough information to describe how to process the message. Clients make state transitions only through actions that are dynamically identified within hypermedia by the server such as hyperlinks within hypertext.)

Services that abide by REST are known as RESTful such as the popular HTTP methods GET, PUT, POST, DELETE etc.

Lease:

In computer science, a Lease is a contract that gives its holder specified rights to some resource for a limited period. Because it is time-limited, a lease is an alternative to a lock for resource serialization.

Leases are commonly used in distributed systems for applications ranging from DHCP address allocation to file locking, but they are not (by themselves) a complete solution:

- There must be some means of notifying the lease holder of the expiration and preventing that agent from continuing to rely on the resource. Often, this is
- done by requiring all requests to be accompanied by an access token, which is invalidated if the associated lease has expired.
- If a lease is revoked after the lease holder has started operating on the resource, revocation may leave the resource in a compromised state. In such situations, it is common to use Atomic transactions to ensure that updates that do not complete have no effect.

Consensus:

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.

A consensus protocol tolerating halting failures must satisfy the following properties:

Termination

Every correct process decides some value.

Validity

If all processes propose the same value v, then all correct processes decide v.

Integrity

Every correct process decides at most one value, and if it decides some value v, then v must have been proposed by some process.

Agreement

Every correct process must agree on the same value.

SSL

The **Secure Sockets Layer** protocol is an accepted protocol for data encryption and authentication between clients and servers. The **Transport Layer Security** is based on SSL.

The SSL exists in the network layer based on top of the TCP/IP transfer protocol, but below the higher level application protocols such as HTTP, LDAP and IMAP.

Its provides **SSL Server Authentication**, **SSL Client Authentication** and **an encrypted SSL connection**. SSL has two sub protocols, the record protocol which defines the format to transmit data, and the handshake protocol to exchange a series of messages between the client and server when they first establish a connection. The handshake allows the server to authenticate itself to the client using public-key techniques, then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server.

1.) The client sends the server the client's SSL version number, cipher settings, randomly generated data, and other information the server needs to communicate with the client using SSL.
2.) The server sends the client the server's SSL version number, cipher settings, randomly generated data, and other information the client needs to communicate with the server over SSL. The server also sends its own certificate and, if the client is requesting a server resource that requires client authentication, requests the client's certificate.
3.) The client uses some of the information sent by the server to authenticate the server (see Server Authentication for details). If the server cannot be authenticated, the user is warned of the problem and informed that an encrypted and authenticated connection

cannot be established. If the server can be successfully authenticated, the client goes on to Step 4.

4.) Using all data generated in the handshake so far, the client (with the cooperation of the server, depending on the cipher being used) creates the premaster secret for the session, encrypts it with the server's public key (obtained from the server's certificate, sent in Step 2), and sends the encrypted premaster secret to the server.
5.) If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case the client sends both the signed data and the client's own certificate to the server along with the encrypted premaster secret.
6.) If the server has requested client authentication, the server attempts to authenticate the client (see Client Authentication for details). If the client cannot be authenticated, the session is terminated. If the client can be successfully authenticated, the server uses its private key to decrypt the premaster secret, then performs a series of steps (which the client also performs, starting from the same premaster secret) to generate the master secret.
7.) Both the client and the server use the master secret to generate the session keys, which are symmetric keys used to encrypt and decrypt information exchanged during the SSL session and to verify its integrity--that is, to detect any changes in the data between the time it was sent and the time it is received over the SSL connection.
8.) The client sends a message to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished.
9.) The server sends a message to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished.
10.) The SSL handshake is now complete, and the SSL session has begun. The client and the server use the session keys to encrypt and decrypt the data they send to each other and to validate its integrity.

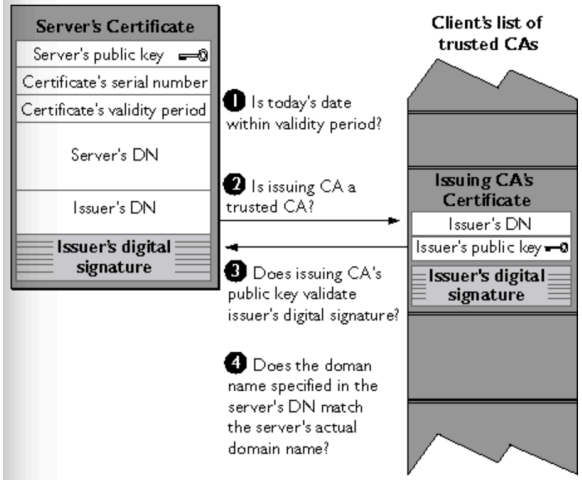
Before continuing with the session, Netscape servers can be configured to check that the client's certificate is present in the user's entry in an LDAP directory. This configuration option provides one way of ensuring that the client's certificate has not been revoked.
It's important to note that both client and server authentication involve encrypting some piece of data with one key of a public-private key pair and decrypting it with the other key:

- In the case of server authentication, the client encrypts the premaster secret with the server's public key. Only the corresponding private key can correctly decrypt the secret, so the client has some assurance that the identity associated with the public key is in fact the server with which the client is connected. Otherwise, the server cannot decrypt the premaster secret and cannot generate the symmetric keys required for the session, and the session will be terminated.
- In the case of client authentication, the client encrypts some random data with the client's private key--that is, it creates a digital signature. The public key in the client's certificate can correctly validate the digital signature only if the corresponding private

key was used. Otherwise, the server cannot validate the digital signature and the session is terminated.

Ciphers (cryptographic algorithms) are supported by SSL. Clients and servers may support different **cipher suites** or sets of ciphers.

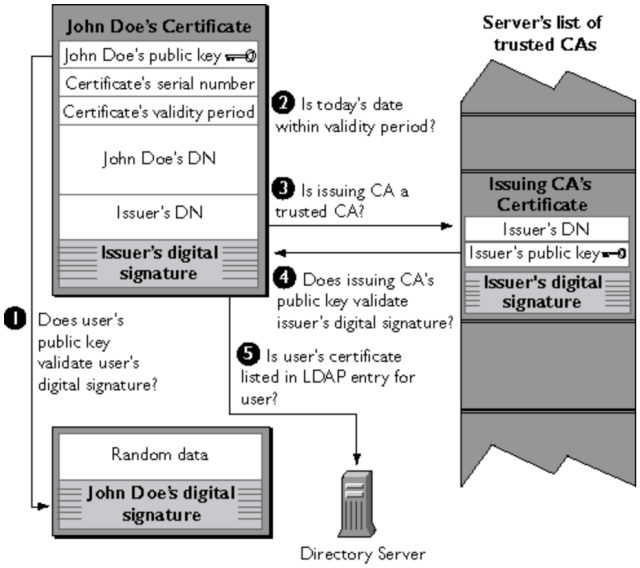
Figure 2 How a Netscape server authenticates a client certificate



The "man in the middle" is a rogue program that intercepts all communication between the client and a server with which the client is attempting to communicate via SSL. The rogue program intercepts the legitimate keys that are passed back and forth during the SSL handshake, substitutes its own, and makes it appear to the client that it is the server, and to the server that it is the client.

Wednesday:

Figure 3 How a Netscape server authenticates a client certificate



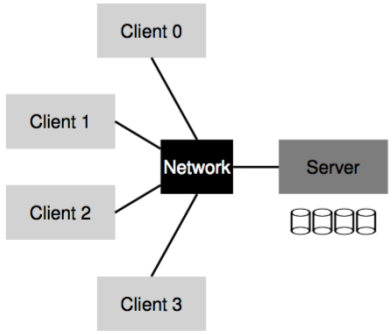


Figure 48.1: **A Generic Client/Server System**

The client server model allows for easy sharing of data, along with the added benefit of **centralized administration**, along with security by locking the servers in a room.

There exists two sides in this model, the **client side file system** and the **server side file system**. The client side, just makes its system calls as it would regularly do so, completely transparent to the underlying mechanisms, but in reality what happens is that if the data for the system call doesn't already exist in the local cache, a network call will be generated where the server will perform the operation in its own disk store and transfer the data back over the network into the in buffer within the local client machine.

Sun Network File System (NFS):

NFS was the first open protocol which is what led to its rise in popularity as everyone could develop their own NFS servers. In a distributed system, the possibility for the largest source of error comes from the crash of a server leading to the crash of the entire system. NFS protocol 2 (NFSv2) took to fixing this issue as best as it could. To do so, they designed a **stateless protocol**. What this effectively entails are that the server has no information about the state of the client(s). Upon each request the protocol, sends information about each client every time. This way the server doesn't have to consider any fancy **recovery protocols** in the situations of either a client or server failure. With a **shared/distributed state** the server is forced to remember everything within memory, but upon a failure and reboot, it would have lost all its data and then any subsequent calls to this server may end horribly. The goal for NFSv2 was to define a protocol that was both stateless and supported the POSIX file system API.

```
NFSPROC_GETATTR
  expects: file handle
  returns: attributes
NFSPROC_SETATTR
  expects: file handle, attributes
  returns: nothing
NFSPROC_LOOKUP
  expects: directory file handle, name of file/directory to look up
  returns: file handle
NFSPROC_READ
  expects: file handle, offset, count
  returns: data, attributes
NFSPROC_WRITE
  expects: file handle, offset, count, data
  returns: attributes
NFSPROC_CREATE
  expects: directory file handle, name of file, attributes
  returns: nothing
NFSPROC_REMOVE
  expects: directory file handle, name of file to be removed
  returns: nothing
NFSPROC_MKDIR
  expects: directory file handle, name of directory, attributes
  returns: file handle
NFSPROC_RMDIR
  expects: directory file handle, name of directory to be removed
  returns: nothing
NFSPROC_READDIR
  expects: directory handle, count of bytes to read, cookie
  returns: directory entries, cookie (to get more entries)
```

Figure 48.4: **The NFS Protocol: Examples**

To understand how this was done we discuss the notion of a **file handle**. This is effectively metadata about each file, containing three key pieces of information which uniquely describes the file/directory a particular operation is going to operate upon:

1.) Volume Identifier – Tells the server which File system to load/look into
2.) Inode number – The corresponding inode of the file/directory within said file system
3.) Generation number – Upon reusing an inode number, this value is incremented such that the client with an old file handle cannot accidently access the newly-allocated file.

Note that the **mount protocol** is what is used to set up the first connection between client and server.

The client side always tracks the state for file access with a server, so as to correctly update any offsets, so that on the server side the requests are still the same and provide the correct results. Note that for longer pathnames, the number of lookups increases proportional to the number of edges from the root directory.

Idempotent – The ability for a client to retry a request without it negatively affecting the overall result.

Somewhat similar to TCP/IP, the idempotent requests simply mean that if a reply isn't received from the server within a time interval from the moment the request was sent, we retry the request. It relies on the idea, that making the request to the same place multiple times is just as good as doing it once because you simply overwrite the data. Thus if I issue a write to the same offset multiple times, then it doesn't matter because I'm just overwriting any result that I may have thought was not received. Reading/lookups are inherently idempotent, but writes and updates are trickier. Some operations are unfortunately very hard to make idempotent such as MKDIR, upon which you may receive an error upon a retry. Much sad.

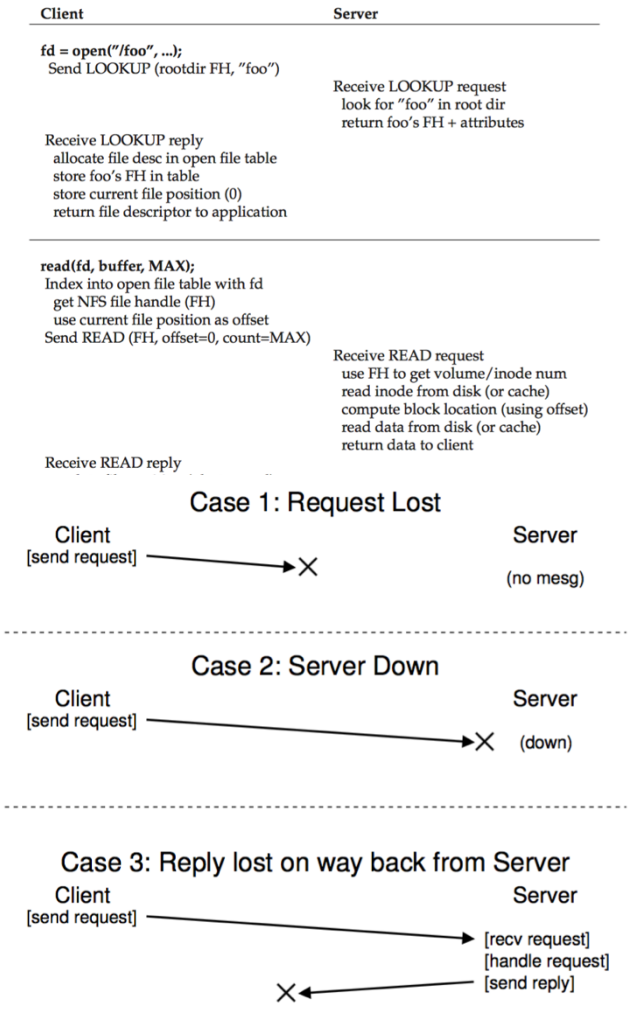


Figure 48.6: **The Three Types of Loss**

As we know from the memory hierarchy, network access is extremely slow. Thus there exists **client side caching** wherein data is cached from the server into local memory for subsequent repeated accesses to be much faster. Another rule as always for updates/writes is **write buffering** wherein we buffer writes and thus perform them in one larger chunk.

Cache consistency problem – How do you trust what's in the current client's cache?

When we have multiple clients working on the same file on the same server, then due to write buffering we could have it such that the clients don't get properly synced up so that some client's may have older data – **update visibility**. The problem described was to do with one client reading before the updating client issues its writes, but the other problem is that if one client simply never re-reads the new and updated data from the server, and continues to operate on old data – **stale cache**.

Flush on close – This is the *solution to update visibility* wherein when a file is written to and subsequently closed by the client, the client with flush all the dirty pages in the cache to the server, thus all reads from then onwards will be latest.

GETATTR and Attribute Cache – To *solve stale caches* the client will issue the getattr protocol to see if the file they are locally modifying has been updated on the server or not. If it has then pull the freshest version else just operate on the cached version. To prevent the server being bombarded with GETATTR requests, the clients were given an attribute cache that had a timer to do constantly get the freshest and juiciest versions.

A problem that existed on the server side was the following. For quick writes and reads, the NFS server would for the most part operate on the data on its memory/cache rather than disk, but informing the client that the request was a success before committing to persistent storage could lead to intermingling of data upon crashes. Thus it was implemented that the request only returns success once the operation is committed onto disk on the server side! This obviously is a bottleneck and thus paved the way for some companies to build faster disk writing file systems and methods to help improve these speeds.

The Andrew File System:

The aim of this distributed file system was the ability to **scale –** support as many clients as possible. The issue with NFS is that its protocol is implemented in such a way that the client has to frequently check with the server for the resources in use, and this limits the number of clients that can be connected to a single server.

Version 1 (ITC Distributed File System):

This relied on **whole-file caching** on the local disk of the client machine that is accessing the file in question. This means that upon issuing a request, the entire file is brought onto the client so that subsequent operations are on the local filesystem and thus much faster. Upon close() the file if modified, would be flushed back to the server. In contrast NFS would only cache blocks in client memory (not on disk as AFS does).

In AFS the client side file system was known as **Venus** and the server side was called **Vice**. Essentially the client would first issue a Fetch protocol by giving the entire pathname to the server which would find the file and ship the entire file to the client. This file would be stored in the clients local disk, and then the operations would work as normal Operating System File Systems, by caching blocks and pages into memory and the TLB and what not. If the file was modified then the client would issue a Store protocol to update the copy in the server. Upon the next request for the same file, a TestAuth protocol would be issued to see if

TestAuth	Test whether a file has changed (used to validate cached entries)
GetFileStat	Get the stat info for a file
Fetch	Fetch the contents of file
Store	Store this file on the server
SetFileStat	Set the stat info for a file
ListDir	List the contents of a directory

Figure 49.1: **AFSv1 Protocol Highlights**

the file on the server was modified (by another client), and if it wasn't then the client could just use the local copy thus avoiding network calls completely!

There were four main problems found with Version 1, as listed below. It should be noted that the first two are the focus of discussion for this course in its current standing:

1.) **Path traversal costs are too high –** Upon Fetch and Store, traversing the entire pathname on the server side proved to be a bottleneck, especially when multiple clients accessed the server at once. Too many CPU cycles were spent on these traversals.
2.) **Clients issued too many TestAuth protocols –** Much like too many calls to GETATTR in NFS, it was found that every client would check far too often if it was okay to use the local cached copy (when in most cases it was fine to do so).
3.) **Imbalanced Loads across servers –** This was solved using **volumes** which an administrator could move across servers to balance the load.
4.) **Single process per client -> overheads –** To reduce the number of context switches, the server was built in Version 2, with threads instead of processes.

The first two problems listed above showed the server CPU as the bottleneck and prevented large scaling before overloading.

Version 2 (AFS):

Client (C_1)	Server
fd = open("/home/remzi/notes.txt", ...);	
Send Fetch (home FID, "remzi")	
	Receive Fetch request look for remzi in home dir establish callback(C_1) on remzi return remzi's content and FID
Receive Fetch reply write remzi to local disk cache record callback status of remzi	
Send Fetch (remzi FID, "notes.txt")	
	Receive Fetch request look for notes.txt in remzi dir establish callback(C_1) on notes.txt return notes.txt's content and FID
Receive Fetch reply write notes.txt to local disk cache record callback status of notes.txt local `open()` of cached notes.txt return file descriptor to application	
read(fd, buffer, MAX);	
perform local `read()` on cached copy	
close(fd);	
do local `close()` on cached copy if file has changed, flush to server	
fd = open("/home/remzi/notes.txt", ...);	

Foreach dir (home, remzi)
if (callback(dir) == VALID)
use local copy for lookup(dir)
else
Fetch (as above)	

if (callback(notes.txt) == VALID) open local cached copy return file descriptor to it	
else Fetch (as above) then open and return fd	

Figure 49.2: **Reading A File: Client-side And File Server Actions**

The server introduced the **callback state** which basically meant that the AFS server would inform the client any time the file they were using was changed. This avoided the continuous TestAuth protocols (analogous to the benefits of interrupts over polling). It also introduced the **file identifier (FID)** like a file handle in NFS to cache directories in path names onto local disk, so that in tandem with the callback state (for all the directories and files in the pathname), next accesses to files within the same or close to same pathnames would be a lot faster (like it would be on a local file system). The FID has a volume identifier, a file identifier and a 'uniquifier' for reuse of volume and files ID's upon file deletion.

The cache consistency problems faced by AFS are different to those faced by NFS.

Different machines – When a client opens and modifies a file from the server, then the server will break all callbacks for all the other clients using that file and ship the new contents out to all the other clients. This avoids the stale cache problems! If however, two clients are operating processes

that operate on the *same file at the same time*, then the **last winner (closer) wins**. This way we don't have mixed up results as we would in NFS.

Same machine – AFS behaves exactly the same as UNIX semantics. If multiple processes on a single client wish to operate on a file, then because the entire updated file is cached in local memory, the changes are immediately visible to all local processes.

With crash consistency we see that if a client is rebooting while a server is sending callbacks informing all clients of updates to a file, then the client that was rebooting may have missed this critical information and would thus have to issue a TestAuth to make sure. If however, the server crashed then its saved callback state would be lost and thus all clients would have to treat their cached contents as suspect. The server would have to upon rebooting tell all clients that "Woah man, your cache shit may not be reliable, do some TestAuth to make sure!". This could be implemented through **heartbeat messages** (a dead pulse is sent to all clients when the server crashes, thus informing the clients of the crash). NFS on the other hand makes it such that the client hardly ever notices a server crash.

In comparing the two studied file systems we notice that for the most part AFS and NFS have similar times on various workloads, with AFS taking advantage of its locality and NFS taking advantage of its smaller grabs (in blocks rather than whole files). Thus NFS tends to work better on smaller data subsets and sequential over-writes (since we don't have to fetch the whole file before overwriting as in AFS), while AFS tends to work better on large-file sequential re-reads.

Workload	NFS	AFS	AFS/NFS
1. Small file, sequential read	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Small file, sequential re-read	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Medium file, sequential read	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Medium file, sequential re-read	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Large file, sequential read	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Large file, sequential re-read	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Large file, single read	L_{net}	$N_L \cdot L_{net}$	N_L
8. Small file, sequential write	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Large file, sequential write	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Large file, sequential overwrite	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Large file, single write	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

Figure 49.4: **Comparison: AFS vs. NFS**

Other AFS awesomeness – Global namespaces (so all clients have the same file names across their systems) – NFS would allow clients to mount their own system thus it would be hard to do this without a dedicated administrator! Security mechanisms to authenticate users and keep certain files private are greater in AFS. AFS also allows for flexible user-managed access control. AFS also adds tools to enable simpler management of servers fro administrators.

The end result however is that because of its open standard, NFS has dominated (along with CIFS) the marketplace. NFSv4 somewhat resembles some of the AFS standards!

Kerberos – Authentication service for Computer Networks:

Note that passwords for user login are not secure over a computer network as a man in the middle can easily intercept the password and impersonate the user. The following terms are redefined as per B.Clifford Neuman and Theodore Ts'o:

Authentication – Verification of the identity of a party who generate some data, and of the integrity of the data.
Principal – Party whose identity is verified.
Verifier – Party who demands assurance f the principal's identity.
Confidentiality – Protection of information from disclosure to those not intended to receive it.
Authorization – Performed after authentication.
Non-repudiation – Ability of verifier to prove to a third party that the message originated with the principal.

Authentication by assertion – Applications assert the identity of the user and the server believes it. This is extremely weak authentication. Cryptography based methods are much stronger than this. This can be thwarted by either modifying the application or spoof asserted network addresses (since this is how the assertion is done!).

Kerberos is a distributed authentication service that allows a process (a client) running on behalf of a principal (a user) to prove its identity to a verifier (an application server, or just server) without sending data across the network that might allow an attacker or the verifier to subsequently impersonate the principal. The Kerberos Authentication System uses a series of encrypted messages to prove to a verifier that a client is running on behalf of a particular user.

Kerberos is not effective against password guessing attacks; if a user chooses a poor password, then an attacker guessing that password can impersonate the user. Similarly, Kerberos requires a trusted path through which passwords are entered. If the user enters a password to a program that has already been modified by an attacker (a Trojan horse), or if the path between the user and the initial authentication program can be monitored, then an attacker may obtain sufficient information to impersonate the user. Kerberos does not itself provide authorization, but V5 Kerberos passes authorization information generated by other services.

Kerberos uses DES encryption, and different keys are used for encryption and decryption. If the ciphertext is modified, then the checksum of the message will not match the data. The user (the client) is given and encryption key, and so is the application server which is shared with the authentication server. When the client authenticates itself to a new verifier it relies on the authentication sever to generate a session key and give it to both the application server and the client. This is done through a **Kerberos ticket –** a certificate encrypted by the server key, containing the random session key, name of principal to whom the session key was issued, expiration time of the key. The ticket is sent to the client who sends it to the verifier during the application request. The client cannot modify the ticket since it is encrypted with the server key.

In the application request, the verifier along with the ticket will check an authenticator – contains current time, checksum, optional encryption key all encrypted with the session key from the ticket.

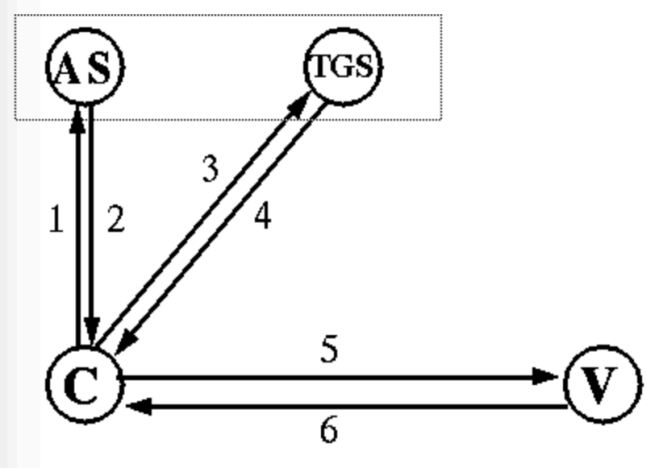
1.) Verifier decrypts ticket, gets session key and decrypts the authenticator with this. If all is well then the checksum of the authenticator will match.
2.) Verifier also checks the timestamp of the authenticator to make sure it is fresh and not used anywhere else within this time interval, which prevents authenticator replay.

Because the client requires separate tickets and session keys for each verifier, it does so through the authentication server.

The basic Kerberos authentication protocol allows a client with knowledge of the user's password to obtain a ticket and session key for and to prove its identity to any verifier registered with the authentication server. The user's password must be presented each time the user performs authentication with a new verifier. This can be cumbersome; instead, a system should support single sign-on, where the user logs in to the system once, providing the password at that time, and with subsequent authentication occurring automatically. The obvious way to support this, caching the user's password on the workstation, is dangerous. Though a Kerberos ticket and the key associated with it are valid for only a short time, the user's password can be used to obtain tickets, and to impersonate the user until the password is changed. A better approach, and that used by Kerberos, is to cache only tickets and encryption keys (collectively called credentials) that will work for a limited period.

When the user first logs in, an authentication request is issued and a ticket and session key for the ticket granting service is returned by the authentication server. This ticket, called a ticket granting ticket, has a relatively short life (typically on the order of 8 hours). The response is decrypted, the ticket and session key saved, and the user's password forgotten.

Subsequently, when the user wishes to prove its identity to a new verifier, a new ticket is requested from the authentication server using the ticket granting exchange. The ticket granting exchange is identical to the authentication exchange except that the ticket granting request has embedded within it an application request, authenticating the client to the authentication server, and the ticket granting response is encrypted using the session key from the ticket granting ticket, rather than the user's password.



The Kerberos system defines two message types, the *safe message* and the *private message* to encapsulate data that must be protected, but the application is free to use a method better suited to the particular data that is transmitted.

In a system that crosses organizational boundaries, it is not appropriate for all users to be registered with a single authentication server. Instead, multiple authentication servers will exist, each responsible for a subset of the users or servers in the system. The subset of the users and servers registered with a particular authentication server is called a realm (if a råealm is replicated, users will be registered with more than one authentication server). Cross-realm authentication allows a principal to prove its identity to a server registered in a different realm. To prove its identity to a server in a remote realm, a Kerberos principal obtains a ticket granting ticket for the remote realm from its local authentication server. This requires the principals's local authentication server to share a cross-realm key with the verifier's authentication server. The principal next uses the ticket granting exchange to request a ticket for the verifier from the verifier's authentication server, which detects that the ticket granting ticket was issued in a foreign realm, looks up the cross-realm key, verifies the validity of ticket granting ticket, and issues a ticket and session key to the client. The name of the client, embedded in the ticket, includes the name of the realm in which the client was registered.

Other methods exist too such as one-time passcodes that generate new passwords for the user each time authentication is required. Public key cryptography is another method but is expensive for high performance servers since the RSA algorithm is expensive.

ACID:

This is a set of properties for database transactions.

1.) Atomicity – Each transaction be all or nothing. If one part of the transaction fails, then all of it fails and the database state remains unchanged.
2.) Consistency – Any transaction will bring the database from one valid state to another. Thus all data writes must be valid according to defined rules.
3.) Isolation – Ensures the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially – main goal of concurrency control.
4.) Durability – Ensures that once a transaction has been committed, it will remain so even in states of power loss, crashes or errors.

WEEK 10

Monday:

Virtual Machine Monitors (VMM)

Nowadays we have the ability to use multiple operating systems within one machine. Such as ‘dual booting’ or even ‘triple booting’ a system, which allows for multiple different OS’s on one machine. This task is accomplished by a VMM. The main goal for a VMM is to provide *transparency* to the OS by allowing it to think that it still interacts with the physical hardware.

Why do we need VMMS?

1.) To allow for *consolidation* of multiple operating systems. Such that administrators can easily manage servers running multiple types and versions for operating systems.
2.) Improved functionality through the ability to use multiple operating systems.
3.) Testing and debugging, to allow to run your program and code on different operating systems allow your product to be portable and robust.

Running a **virtual machine** uses the same basic technique of **Limited Direct Execution** wherein now the VMM boots a new OS on top of itself, by jumping to the address of the first instruction. Switching between OS's is similar to the concept of *context switching* in that we perform a **machine switch**. In doing so, the VMM must save the entire state of the OS (just like a process, but with more stuff to save!) such as registers, PC and **privileged hardware state**, and then restoring the machine state of the OS to be run and jumping to the PC of the to-be run VM.

The OS itself cannot be allowed to perform privileged instructions because then it controls the machine rather than the VMM beneath it. When we run on a single OS, when trying to execute a privileged instruction a trap may be raised which would pass the control into the trap handler which was loaded during boot time, while moving from user mode to kernel mode. When it comes to VMM's, if the OS wants to perform a system call it will first save its state on the stack/registers but now the VMM controls the machine. However, the VMM doesn't know where to store the trap handler for each OS, so what it does instead is that it stores the location of each OS's trap handler within the OS itself. This happens when the OS boots up, it tries to set up its trap handlers which is a privileged instruction and thus it transfers control to the VMM which records the information regarding the location of the OS specific trap handler and then the OS returns with an iret or rett instruction.

Process	Operating System
1. System call: Trap to OS	
	2. OS trap handler: Decode trap and execute appropriate syscall routine; When done: return from trap
3. Resume execution (@PC after trap)	

Table B.2: **System Call Flow Without Virtualization**

Process	Operating System	VMM
1. System call: Trap to OS		
		2. Process trapped: Call OS trap handler (at reduced privilege)
	3. OS trap handler: Decode trap and execute syscall; When done: issue return-from-trap	
		4. OS tried return from trap: Do real return from trap
5. Resume execution (@PC after trap)		

Table B.3: **System Call Flow with Virtualization**

But what mode should the OS run in? It can't run in **kernel mode** since that's the VMM's job, but it can't run in pure user mode since it needs some of its data structures (page tables, free lists, malloc arena etc.) from the user processes. In some systems the OS was given **supervisor mode** using MIPS, others just use the VMM to secure the OS data structures to the OS only by using protection with access granted to the OS.

We have already looked through basic virtualization where each process has a page table to handle virtual page numbers translation to physical memory. With VMM's we have an extra level of indirection through the use of the **machine memory** (the real physical memory of the system). The OS maps virtual to physical addresses through per process page tables, the VMM maps the resulting physical mappings to underlying machine addresses through the per OS page table.

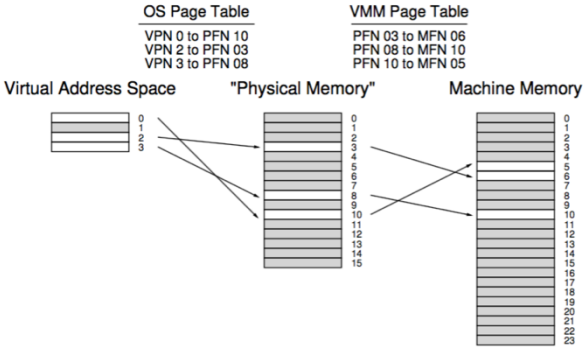


Figure B.1: **VMM Memory Virtualization**

In normal systems, TLB miss would result in the system raising a trap to look through the page table using the VPN from the virtual address, and if the page is present and valid it would get the PFN and update the TLB, return from the trap and resume the same instruction this time getting a TLB hit! However, now the situation is a little trickier. A TLB miss generates an OS trap which raises the VMM trap which calls into the OS trap handler (loaded at boot time) which then extracts the VPN and does the page table lookup and grabbing of the PFN if valid, updating the TLB – but this is a privileged instruction and so we jump into the VMM which adds a VPN-MFN translation in the TLB instead and jumps back to OS, returns and resumes execution. This means the VMM has to hold a set of per-machine page table for the VMM TLB miss handler. This is all of course for a **software managed TLB**. In a hardware version, the VMM is not as involved and simply holds a **shadow page table** which just maps the virtual address of each process to the VMM's desired machine pages (per process). This can get quite slow, so there exists a software VMM TLB which is consulted first before the entire sequence of execution runs.

Process	Operating System	Virtual Machine Monitor
1. Load from memory TLB miss: Trap		
		2. VMM TLB miss handler: Call into OS TLB handler (reducing privilege)
	3. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid, get PFN, update TLB	
		4. Trap handler: Unprivileged code trying to update the TLB; OS is trying to install VPN-to-PFN mapping; Update TLB instead with VPN-to-MFN (privileged); Jump back to OS (reducing privilege)
	5. Return from trap	
		6. Trap handler: Unprivileged code trying to return from a trap; Return from trap
7. Resume execution (@PC of instruction); Instruction is retried; Results in TLB hit		

Table B.5: **TLB Miss Flow with Virtualization**

Just as there is an **information gap** between the OS and the application programs, the same exists between the VMM and its OS's. This can lead to various inefficiencies such as the examples given below:

1.) If an OS runs a while loop that is idle (for spinning for example), while another OS is trying to do something useful, it would be improve performance for the VMM to know that more CPU time should be given to the latter OS.
2.) **Zeroing pages –** Before a process is gives up its page it must zero it out to clear out all its contents. With VMM's this could happen twice as the page is zeroed for the OS and then the OS zeroes it for the process. This isn't super effective.

Some solutions such as **implicit information gathering** or **para-virtualizing** could be used to fix this, but its not worth going in depth in this course.

The key difference between OS virtualization to processes and VMM virtualization to OS's is that the interface of OS-process is a lot nicer, because VMM-OS is very much like the abstraction provided by the hardware itself. There is a lot more to be delved into with virtualization such as **hosted configurations** and I/O but such is a topic of later discussion.

Back to Multi-Processors

We talk about multi processors because with computers we always demand speed. Horizontal scaling in distributed systems (Just getting more servers – more expensive) is not feasible sometimes, and trying to make CPUs faster is becoming harder to due to limitations by physics. Thus speedups are achieved through smarter pipe-lining, parallelization and putting more cores per chips and more chips per board and more boards per computer system.
Hyper Threading – The CPU uses shared levels of registers to parallelize by running micro instructions concurrently. Thus when one register is waiting for a value from memory the other can be put into execution mode. This can get 1.2~1.8 times speedup per core.
From a performance point-of-view, it is important to understand that both hyper-threads are running in the same core, and so sharing the same L1 and L2 cache. Thus hyper-threads that use the same address space will exhibit better locality, and hence run much better than hyper-threads that use different address spaces.

A **Symmetric Multi-Processor** has some number of cores, all connected to the same memory and I/O busses. Unlike hyper-threads these cores are completely independent execution engines, and (modulo limitations on memory and bus throughput) N cores should be able to execute N times as many instructions per second as a single core.
As mentioned previously, much of processor performance is a result of caching. In most SMP systems, each processor has its own L1/L2 caches. This creates a potential **cache-coherency** problem if (for instance) processor 1 updates a memory location whose contents have been cached by processor 2. Program execution based on stale cache entries would result in incorrect results, and so must be prevented.

It is not feasible to create fast memory controllers that can provide concurrent access to large numbers of cores, and eventually memory bandwidth becomes the bottleneck that prevents scaling to larger numbers of CPUs. A **Non-Uniform Memory Architecture** addresses this

problem by giving each node or CPU its own high-speed local memory, and interconnecting all of the memory busses with a slower but more scalable network.

Operations to local memory may be several times faster than operations to remote memory, and the maximum throughput of the scalable network may be a small fraction of the per-node local memory bandwidth. Such an architecture might provide nearly linear scaling to much larger numbers of processors, but only if we can ensure that most of the memory references are local. The Operating System might be able to deal with the different memory access speeds by trying allocate memory for each process from the CPU on which that process is running. But there will still be situations where multiple CPUs need to access the same memory. To ensure correct execution, we must maintain coherency between all of the per-node/per-CPU caches. This means that, in addition to servicing remote memory read and write requests, the scalable network that interconnects the nodes must also provide cache coherency. Such architectures are called *Cache Coherent Non-Uniform Memory Architectures* (CC-NUMA), and the implementing networks are called *Scalable Coherent Interconnects.*

With multi-processor systems, it can be difficult to schedule tasks, and it may not be intuitive when to do the scheduling. As a general rule of thumb, we would like to keep all the cores busy at all times, but some cores may run some tasks more efficiently than others:

- dispatching a thread from the same process as the previous thread (to occupy that core) may be much less expensive because re-loading the page table (and flushing all cached TLB entries) is a very expensive operation.
- a thread in the same process may run more efficiently because shared code and data may exploit already existing L1/L2 cache entries.
- threads that are designed to run concurrently (e.g. parallel producer and consumer communicating through shared memory) should be run on distinct cores.

Sharing data also becomes troublesome as disabling interrupts no longer becomes an option, and implementing a global kernel lock essentially does the same thing as running a single core. Thus we must pay attention to finer grained locking, with the use of better locking mechanisms given the different situations, this becomes a cumbersome task and for the most part most OS's choose simplicity over scalability.

We might want to choose carefully which cores handle which I/O operations:

- as with scheduling, sending all operations for a particular device to a particular core may result in more L1/L2 cache hits and more efficient execution.
- synchronization between the synchronous (resulting from system calls) and asynchronous (resulting from interrupts) portions of a device driver if they are all executing in the same CPU.
- each CPU has a limited I/O throughput, and we may want to balance activity among the available cores.

some CPUs may be bus-wise closer to some I/O devices, so that operations go more quickly initiated from some cores.

CC-NUMA is only viable if we can ensure that the vast majority of all memory references can be satisfied from local memory. Doing this turns out to create a lot of complexity for the operating system.
When we were discussing uni-processor memory allocation, we observed that significant savings could be achieved if we shared a single copy of a (read only) load module among all processes that were running that program. This ceases to be true when those processes are running on distinct NUMA nodes. Code and other read-only data should be having a separate copy (in local memory) on each NUMA node. The cost (in terms of wasted memory) is negligible in comparison performance gains from making all code references local.
When a program calls *fork(2)* to create a new process, *exec(2)* to run a new program, or *sbrk(2)* to expand its address space, the required memory should always be allocated from the node-local memory pool. This creates a very strong affinity between processes and NUMA nodes. If it is necessary to migrate a process to a new NUMA node, all of its allocated code and data segments should be copied into local memory on the target node.
As noted above, the operating system is full of data structures that are shared among many cores. How can we reduce the number or cost of remote memory references associated with those shared data structures? If the updates are few, sparse and random, there may be little we can do to optimize them ... but their costs will not be high. When more intensive use of shared data structures is required, there are two general approaches:

- move the data to the computation
    - lock the data structure.
    - copy it into local memory.
    - update the global pointer to reflect its new location.
    - free the old (remote) copy.
    - perform all subsequent operations on the (now) local copy.
- move the computation to the data
    - look up the node that owns the resource in question.
    - send a message requesting it to perform the required operations.
    - await a response.

Clusters – Group of nodes that exist in a network connection, all of whom agree that they are part of the cluster. They are broken into the following types:

- load sharing clusters, which divide work among the members.
- high availability clusters, where back-up nodes take over when primary nodes fail.
- information sharing clusters, which ensure the dissemination of information throughout a network.

Nodes have to have a membership into a cluster. Either potential or active memberships. If they are active then they can communicate with each other, else they can't.

Clusters are great for redundancy because we can have **primary** nodes currently running the service and **secondary** nodes to take over when shit hits the fan. There are three fundamental approaches:

- **Active/Stand-By**
  The system is divided into *active* and *stand-by* nodes. The incoming requests are partitioned among the active nodes. The stand-by nodes are idle until an active node fails, at which point a stand-by node takes over his work.
- **Active/Active**
  The incoming requests are partitioned among all of the available nodes. If one node fails, his work will be redistributed among the survivors.
- **Hot Standbys/Wait to be notified**
  The secondary will mirror the primary which can take more network and CPU load but has faster fail-overs, while the latter consumes fewer resources but takes longer to resume service.

Clusters can experience failures such as system crashes or busy nodes or slow network interfaces, which thus require that each node has a heart beat to let the cluster know whether they are dead or alive within the cluster (if dead then kick them out).

It is common to have a **master** node through which things are run. It serves as a central point of synchronization and control fro operations in the cluster. It can be the point to send heartbeats and the only heartbeat other nodes listen to. Picking a master is a hard task.

Sometimes clusters can be broken apart by network failure into sub clusters which cannot communicate with each other – **split brain/partitioning**. This could lead to incompatible decision making which is not good. We can prevent such an event through:

1.) **Quorum –** A cluster cannot form with fewer than (N/2) + 1 members. This makes it impossible for any portioning to result in two viable sub-clusters, and it ensures that any decision made is remembered by future quorums (if persisted) since at least one member will be carried over.
2.) **Voting Devices -** If there is a single piece of hardware in the cluster, that must be present for the cluster to function, and that can only be owned by one node, that device can be used as a voting device.

Fencing – If a sub-cluster/node has gone rogue and now tries to attack your cluster then you fence them out of the cluster through the following methods:

- **Reservable devices**
  Some devices can be told which interface to listen to, and not to listen to the other interface. This is often done with dual-ported disks. The node that has seized the quorum device will then instruct the quorum device not to accept commands from any other node.

- Remote power control
  Some clustered systems come with remote power controllers, and a node that has seized control of the cluster from a previous (apparently failed) cluster master will often power-off or reset the previous master, to ensure that he does not continue to vie for control of the cluster and its resources.

Horizontally Scalable Systems

Horizontally scalable refers to systems whose capacity and throughput are increased by adding additional nodes. This is in distinction to *vertically scaled* systems, where adding capacity and throughput generally involves replacing smaller nodes with larger and more powerful ones.

These systems started with many of the same goals shared by most distributed systems: scalability of capacity and throughput, reliability, and availability. But whereas earlier distributed systems attempted to provide single-system services and semantics, horizontally scaled systems offer simplified (and decidedly non-Posix) semantics. Most of these simplifications have a few basic (and highly inter-related) goals:

- improve scalability and robustness by eliminating the need for synchronization and communication between parallel servers.
- improve flexibility and performance by enabling the (non-disruptive) addition or removal of servers at any time.
- stateless protocols that permit requests to be arbitrarily distributed and redistributed among those servers.
- turn servers into standardized components, easily deployed and requiring little or no configuration.
- service protocols that are designed to minimize the potential modes of failure and enable simple recoveries.
- service protocols that are optimized for throughput (vs latency) over very large (e.g. continental) distances.
- service protocols that exploit the numerous optimizations and other features that have come to be standard in networks that serve heavy HTTP traffic.

And, as a (not unintentional) side-effect, these systems are much simpler to design, maintain, deploy, and manage than other (more complex) distributed systems.

If a horizontally scalable system is comprised of many servers, and new servers can be added at any time, it is essential that they all have appropriate network connectivity (for client-facing traffic, back-side servers, and peer-to-perr replication). It is not practical to rack new switches and connect new cables each time a new server is to be integrated into a particular service. Rather, large computing facilities are moving towards *Software Defined Networking* where all servers (virtual or real) are connected to large and versatile switches that can easily be reconfigured to create what ever virtual network topologies and capacities are required for the service to be provided.

Storage is a similar problem. New virtual machines need virtual disks for the operating system to boot off of, and to store the content they will manage. Large computing facilities are also moving towards *Software Defined Storage*, where intelligent storage controllers draw on pools of physical disks (or SSDs) to create virtual disks (LUNs) with the required capacity, redundancy, and performance. The new servers have no idea what the underlying storage is, but merely use remote disk access protocols (e.g. Fibre Chanel or iSCSI) to access configured storage.

These types of servers have no single point of failure as their architecture is generally highly independent as they shared nothing physical or virtual with each other. However, a large cluster of these servers could be taken down (say due to a natural disaster) and so backups are placed far away in other cities/countries – *geographical disaster recovery*.

The classic horizontally scaled system is a farm of web-servers, all of which have access to/copies of the same content. As demand increases, more servers are brought on-line. When each server starts up, it is made known to a network switch that starts sending it requests.

- Since HTTP is a stateless protocol, the switch can route any request to any server. This routing may be as simple as round-robbin, or it may be load-balanced based on measured response times to recent requests.
- Parallel servers have no need to communicate with one-another, and the only configuration a web server requires is knowing where to find the content it is serving.

If a web-server crashes, the switch will stop sending it new requests. If the operations in the service protocol are idempotent, the client will time-out and retransmit the request, which will automatically be routed to a different server.

Even (highly stateful) storage systems (including databases) can be designed to be horizontally scalable. Distributed Key/Value Stores often divide the key namespace into a independent subsets (usually called *shards*), each of which is assigned to a small group (e.g. 3) of servers. Assigning multiple servers to each shard enables us to maintain data availability even if some servers fail. Imposing a low cap on the number of servers involved in any particular transaction enables the protocols to scale to thousands of nodes with no degradation in performance. When we need to add capacity, we do not add more servers for a particular shard; Rather we increase the number of shards into which the name space is divided, and assign the new shards to new servers. Similar techniques have been employed to improve the scalability of block storage, object storage, and even file storage services.

The *cloud model* may be more a business model (renting services from an independent service provider, vs. buying equipment) than an architecture, but it is very well aligned with horizontally scalable architectures:

- All services are delivered via (well standardized) network protocols.
- The cloud model opaquely encapsulates the individual servers behind a single highly available IP address. This enables the network to distribute requests among the available servers.

- The resources presented by a cloud service are intended to be abstract/logical and thin-provisioned. Horizontally scaled architectures are well suited to implementing and delivering such services.
- The flexibility to continuously add, remove, and rebalance resources in a horizontally scalable system, makes it possible for a cloud service provider to easily accommodate great fluctuations in demand.
- Public cloud services are (in most cases) unlikely to be co-located with the clients, and so the service protocols must be optimized for (throughput) efficiency over WAN-scale communications links.
- Horizontally scalable systems' ease of deployment/management yields economies of scale that make it possible for cloud service providers to offer their services at very competitive prices. Consider, for instance, the cost of maintaining redundant computing facilities in many different cities. This is normal operation for a large service provider, but would be prohibitively expensive for the typical company to do for itself.

Eventual consistency

Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

A system that has achieved eventual consistency is often said to have **converged,** or achieved **replica convergence.**

In order to ensure replica convergence, a system must reconcile differences between multiple copies of distributed data. This consists of two parts:

1.) Exchanging versions or updates of data between servers (often known as **anti-entropy**).
2.) Choosing an appropriate final state when concurrent updates have occurred, called **reconciliation**.

The most appropriate approach to reconciliation depends on the application. A widespread approach is *"last writer wins"*.

Reconciliation of concurrent writes must occur sometime before the next read, and can be scheduled at different instants:

- **Read repair**: The correction is done when a read finds an inconsistency. This slows down the read operation.
- **Write repair**: The correction takes place during a write operation, if an inconsistency has been found, slowing down the write operation.
- **Asynchronous repair**: The correction is not part of a read or write operation.

Whereas EC is only a liveness guarantee (updates will be observed eventually), strong eventual consistency (SEC) adds the safety guarantee that any two nodes that have received the same (unordered) set of updates will be in the same state. If, furthermore, the system is monotonic, the application will never suffer rollbacks. Conflict-free replicated data types are a common approach to ensuring SEC.