10 Question Midterm

- each has 4-5 sub-question (3 minutes per sub-question)
- Mixture of Reiher and Eggert's midterm
- Try to be neat with your handwriting
- TA's spend less than one minute on each question -> try to itemize everything (bullet points)
- You can draw figures or write examples down if you are not sure
- 20-25% of the questions will be taken from the previous quarters
- Go to UPE review session
- Scans everything into gradescope -> do not write on backside

https://stackoverflow.com/questions/3513045/conditional-variable-vs-semaphore Condition var vs Semaphore. Also do locking

Sample Questions from Discussion

WEEK 1: Possible Questions

Mechanism/Policy Separation

- Why is it important?
    - It is important so that the implementations do not overly constrain the applications. The mechanisms should not put too much constrain on the policies over what an OS can do
    - By separating mechanism from policy, we can use a completely different mechanism to support the same policies (we can use completely different implementations to support the same applications), and therefore updates can be made very easily
- Example
    - When building resource managers, you must be sure to keep track of which resources to given access to and when to revoke them (mechanisms)
        - could have a configurable plugin policy that determines who gets what resource -> to the client
    - Another example could be something like schedulers
        - The policy would be the scheduling algorithm that determines which process will be run to achieve virtualization
        - The mechanism would be the trap handlers that initiate the scheduling process
- Tips
    - Think of mechanism as a how question
        - how does an OS perform context switches
    - Policies are which?

        - which process should the OS run?

Interface vs. Implementation

- Tips
    - Think of interfaces as contracts: An agreement to produce a quality product that meets the clients' needs as well as an assurance for the future
- Why is it important to distinguish interface and implementation
    - Since an interface can be thought of as a contract, if you do not distinguish the two, faulty code and buggy programs may emerge. Additionally, creating updates and versions for interfaces will become near impossible because of the differences
    - If the interface is an implementation-agnostic specification of expected behavior, it is easier to change the implementation while keeping the same interface stability
- Example
    - Think of API's: You can implement APIs in your own way to create applications (interface) as long as the provider promises to deliver what you want
    - By not distinguishing the two, i.e. implementing implementations that work with a particular implementation but not with the interface, APIs can be used incorrectly, leading to buggy programs

Modularity/Information Hiding

- Why is it important?
    - We want to be able to think of a system in a layered/hierarchical manner
    - Modularity means that every component will be able to run independently (modules) but combined achieve what the client desires
    - This allows us to easily make updates and versions to modules and still keep the program from breaking, which displays good information hiding
- What is information hiding and why is it important?
    - An application displays good information hiding when it is able to deliver an interface without uncovering its underlying implementations
    - If you reveal information about implementations, clients can become confused and complexities will lead to perplexed customers
    - Reduces flexibility because it limits the provider's ability to change things in the future to create new versions/updates
- How does modularity improve performance?
    - If we are able to fit most functionality into one module, we will be able to run programs efficiently because there will be less overhead of combining modules to make a single application
- What are some complexity management techniques?
    - putting closely related modules into a single module drastically simplifies the code while improving modularity and decreasing overhead

    - HIerarchical decomposition makes it easy to draw dependencies and understand things one at a time
    - Opaquely encapsulating modules allow simplification of interfaces without confusing clients with implementation specifications

Powerful Abstractions

- What is it and why is it important?
    - Powerful abstractions are abstractions that can be applied to many situations. Some examples include common paradigms, common architectures, and common mechanisms
    - If we have powerful abstractions that can be applied to many different situations, we will be able to simplify systems that would otherwise be overly complex while also properly organizing it

Appropriate Abstraction

- What is it and why is it important?
    - Appropriate abstractions are abstractions that do not reveal the underlying implementation, meaning that it displays good information hiding. An appropriate abstraction is an interface that is well-suited for the clients needs. For example, a faucet that has a knob for adjusting temperature and a knob for adjusting flow rate is an interface that is well-suited for clients
    - It is important to build these so that we can live up to the interface contract that we promised with a client

Indirection/Deferred Binding

- What is it?
    - When a module is not binded until the client requests it, improving the efficiency of programs -> think of DLLs
    - Indirection is when a federation framework stores multiple implementations and enables clients to select their desired implementations

Cohesion

- What is it and why is it important to achieve cohesion?
    - Cohesion is when closely related modules are stored into one module in order to approve efficiency and readability while simplifying implementations
    - Cohesion decreases overhead in the OS which means that the efficiency will dramatically improve and error handling becomes much easier

Opaque Encapsulation

- What is it?
    - Opaque encapsulation means to hide all of the complexities of a module, such as its implementations and appropriately deliver a quality product to a client

- Our overall goal is to be extremely clear with the client and not reveal any complexities
- An application that displays good information hiding can be said to opaquely encapsulate the module

Dynamic Equilibrium

- What is it?
  - The balancing/maintaining the state of equilibrium with two opposing forces
  - Dynamic equilibrium makes it easy to make updates to systems by making once force stronger/weaker

Lecture 2

Basic OS Services

- What are the basic OS services?
  - **Hardware Abstractions**
    - **CPU/Memory Abstractions**
      - processes/aka virtual CPUs
      - virtual address spaces
      - signals
      - CPU
    - **Persistent Storage Abstractions**
      - files and file systems
      - databases, file storage, key value maps
    - **Other**
      - I/O, virtual terminal sessions, VPN, signals
  - **Higher Level Abstractions**
    - Cooperating Parallel Processes -> e.g. locks and mutexes
      - you need locks to save personal data so that no other processes can access them -> e.g. avoiding race conditions with atomicity in critical sections
    - Security
      - user authentication
    - User Interface
      - GUI widgetry, desktop and window management
      - mutli-media
  - **Under the Covers**
    - Enclosure management
    - Software updates and configuration registry
    - dynamic resource allocation and scheduling
      - CPU, memory resources, disk, networks
    - Networks, protocols, and domain services
      - UDP/TCP, USB etc.

    - Summarized answer for test: Hardware Abstractions: CPU/Memory abstractions such as processes and virtual CPUs, Persistent storage abstractions such as databases and file systems, and other abstractions such as I/O, VPN, and signals Higher level abstractions: security for authentication, cooperating parallel processes -> locks and conditions variables, user interface such as GUI widetry and window management. Under the cover services such as software management, network protocols and domain services such as UDP, TCP, and USB

Higher Level OS services

- What are the higher level OS services? (talked about in above examples)
    - **Cooperating parallel processes (and threads), involving locks and condition variables, security with authentication, UI (GUI widgetry, desktop and window management)**
- Just to remember (drill this in)
    - Hardware Abstractions
        - Memory Management Abstractions: virtual CPUs, processes, stuff like the scheduler, signals, and CPUs
        - Persistent Storage: Databases, key-value stores, maps, etc.
        - Other: I/O, disk management, VPN, virtual terminal sessions
    - Under the covers
        - Software management tools, networks, protocols, UDP TCP, USB, etc.

Layers of OS services

- What are they?
    - Libraries: A collection of subroutines
        - convenient functions that we use all the time
        - encapsulates complexities, reusable code that makes programming easier via a single well-maintained code
        - Multiple binding types: static, shared, dynamically loaded
        - Key: libraries are only code and thus do not share any special privileges
    - Kernel
        - Privileged instructions are required to access the kernel
            - ex: I/O operations, allocation of memory (malloc, sbrk),
        - The kernel can be thought of as the minimal set of functions you need to implement -> coming from linear algebra's definition of the kernel
    - System Services
        - not all trusted code has to be in the kernel
            - may not need to access kernel data structures
            - may not need to execute privileged instructions
            - Executing in user mode is significantly faster
        - some are actually privileged instructions

            - login can create/destroy user credentials
            - some can directly execute I/O operations
    - Middleware
        - Something that is not part of the OS but is still widely used
        - Software that is not a part of the OS but plays a key part in the application
        - Examples include Apache, Nginx, Hadoop, Zookeeper, Cassandra, etc.
    - Summarized answer: The core layers of the OS include libraries, or reusable pieces of code with no special privileges that encapsulates complexities and simplifies code, the kernel which a set of privileged instructions such as I/O operations, allocation of memory resources, process-privacy containment, etc., system services which executes privileged instructions that are not in the kernel (such as system calls (because user-mode code is much master)), and middleware, which is software that is not part of the OS but still plays an integral part of the application
- Subquestion: Is all trusted code in the kernel?
    - No, because user-mode code is significantly faster than kernel-mode code which is expensive. Some trusted code may not need to access kernel data structures or may not need to execute privileged operations. Instead, we can use system calls from the user in order to reduce overhead

Service Protocols

- What are the different kinds of service protocols
    - Service delivery via subroutines (a bunch of code that was, in principle, separately packaged and compiled)
        - push parameters, jump to subroutine, and then store registers in the stack
        - Advantages
            - extremely fast
            - enables DLL's which increases efficiency
        - Disadvantages
            - limited ability to change library functionality
            - all instructions executed in the same address space -> could lead to memory problems
            - limited ability to code in multiple languages
    - Service delivery via a system call
        - forces its way into the operating system
        - executes trusted code in the kernel to handle privileged instructions such as I/O requests and memory allocation
        - Advantages
            - able to share/communicate with other processes
            - able to allocate new privileged resources
        - Disadvantages
            - much slower than subroutine calls

            - evolution is very slow and expensive because kernel code must be supported by many platforms
    - Service delivery via messages
        - Remember that a subroutine is a method that executes a service
        - exchanges in response with a server
            - req and res
        - Advantages
            - the server can be located on Earth and is thus highly convenient to use. allows multiple processes to communicate with it, and is also highly scalable
        - Disadvantages
            - Server maintenance, as well as speed is an issue
            - limited ability to operate on process resources
    - Summarized Answer: The main types of service protocols are service delivery via subroutines, via system calls, and via messages.
- What are the advantages and disadvantages of using subroutines as opposed to system calls (and messages)?
    - Subroutines are extremely fast (DLLs improve efficiency) but have limited capabilities in terms of sharing information with other processes because they are executed within a single address space. While system calls are fairly slow and evolution is expensive, it provides a wider range of information to share with processes, allowing communication with other processes to be possible. Messages through servers provide a very scalable solution in sending information from process to process. Because this server can be placed anywhere in the world, it can be highly convenient to use. However, sending messages through a server can be very slow and is tough to operate on process resources

APIs and ABIs

- Key thing to remember: An ABI is a binding of an API to a particular ISA (instruction set architecture)
- APIs: a source level interface specifying included header files, parameters, return values, and possibly an example of an implementation.
- ABIs: a binary level interface (bunch of 0s and 1s) that specifies the fundamental data type size, layout and alignment, exception propagation mechanisms, load module format, and system call invocation mechanisms
- Explain the differences between an ABI and API
    - Answer: An ABI is a binding of an API to a particular ISA. This means that the API is defines a subroutine and how to use them while an ABI gives the machine language instructions in order to execute the APIs. The portability of an API is much greater than the portability of an APi simply because APIs tend to be more language specific while ABIs ensure that a binary application will run on numerous devices such as an x86 machine or an Android phone
- What does a typical API contain?

    - Return values and types
    - Routine names
    - associated header files
    - example code
    - associated parameter types
- What does a typical ABI contain?
    - load module format
    - exception handling mechanisms
    - fundamental data size, layout, and alignment
    - system call invocation mechanism
- What are the consequences of an incompatible ABI change? (from previous midterm)
    - Because the ABI is a binary interface that binds the API to a particular instruction set architectures, an incompatible ABI change will mean that we would have to change existing dependencies on machines that support that ABI. Programs will likely stop working because they will not be supported by the OS
- How would an incompatible ABI change be responded?
    - The Independent Software Vendor may have to redistribute a new copy of the OS to customers that were affected by this incompatible change. They would have to modify the code so that it works with the new interface, recompile everything, and then either release a completely new version or help affected customers
- How would clear interface and implementation differentiation help?
    - Interface standards should specify behavior rather than design, meaning that they have to be as implementation-neutral as possible
    - Clear interface standards provide contracts for users that promises them a certain functionality. By providing clear-cut interface standards, we are able to easily create versioned interfaces when updates are necessary, and to avoid incompatible ABI/API changes. This would also make it easier for the OS provider to realize that they were making changes to an existing interface and therefore third-party applications may be affected

Upwards Compatibility

- What is upwards compatibility and what are the methods to properly achieve it?
    - upwards compatibility is the ability to be compatible with both the current version and future versions. It can be achieved through polymorphism and versioned interfaces. Polymorphism can be thought of as different method signatures that support the same general interface functionality. Versioned interfaces, (e.g. stuff they do in Java), can be released as updates so that the user knows which interface is compatible with their code. We must make it extremely clear how the changes in the interface affect the user
- Why is it important to maintain interface stability?

    - In creating interfaces that are upwards/backwards compatible, it is important for an interface to be stable so that one version does not break old/newer versions of an interface.

Object and Operations

- Possible Question: What is the advantage of using objects in providing services
    - create more convenient behavior
    - encapsulates complexities
    - hides parts that are irrelevant to the user
    - easier to use than original resources
- Another Question: What's the difference between simplifying an abstraction and generalizing an abstraction?
    - simplifying: encapsulate implementation details and offer more convenient or powerful behavior
    - generalizing: generalizing an abstraction requires a unifying model because you are making different things appear the same for simplicity
- More general question: What are some ways to design a system with interface stability in mind?
    - rearrange the distribution of functionalities between components to create simpler interfaces between them
    - Design features that we may never implement, so that interfaces will accommodate them when we need them
    - Introduce a large amount of abstraction so that we have lots of room for future improvement

Resources (Serially reused, Partitioned, and Shareable)

- What is a serially reusable resource and give an example
    - A serially reusable resources are resources that can be used by multiple clients but only one at a time -> temporal multiplexing
    - examples in real life: a toilet
    - examples in OS: a lock for multithreading, a network card -> think of multithreading
- What is a partitionable resource
    - A partitionable resource is a resource that can be split between multiple clients to use
    - Examples in OS: memory -> think of paging and splitting
    - examples in real life: desk space in a classroom
- What is a shareable resource?
    - A sharable resource is a resource that can be used by multiple clients at the same time
    - Examples in real life: air
    - Examples in the OS: a shared or dynamically loaded library
- Remember the differences and give an example

Subroutines vs System Calls

- Subroutines and system calls both provide a way of method of service delivery
- What are the advantages and disadvantages of both?
    - Advantages of Subroutines
        - Subroutines are extremely fast
        - DLLs increase efficiency of program
    - Disadvantages
        - Subroutines are all called within the same address space
        - cannot establish connections/communications between processes
        - limited ability to change library functions
    - Advantages of System Calls
        - System calls provide a means of inter-process communication
        - Has the ability to allocate new privileged resources
    - Disadvantages
        - slower than subroutines
        - evolution of system calls is very slow and expensive
    - Extra: Messages (advantages)
        - sends messages to a server, which an be located anywhere in the world
        - This means that it is highly scalable
    - Disadvantages
        - Server maintenance is required
        - much slower than subroutines or system calls
        - Limited ability to operate on process resources

Versioned Interfaces

- What is a versioned interface
    - A versioned interface is a technique to maintain upwards and backwards compatibility of interfaces in order for clients to be clear with the changes made to interfaces and to make sure that they are running code that fits their implementations
- How does a versioned interface support upwards/backwards compatibility?
    - We can provide a wide array of versions of interfaces so that the user can pick which one they want to use to fit their goals

WEEK 2

Programs and Processes

- Processes can be thought of as an executing instance of a program
- What is the difference between a program and a process?
    - A program is an executing instance of a process

    - From the OS perspective, the program is simply a bunch of 1's and 0's that execute a series of instructions that are combined to create a machine executable program
    - From our perspective, a series of source files that can be translated into machine language is a program

Process Address Space

- What is in a process address space?
    - The process holds many segments.
    - A stack segment that grows upwards/downwards depending on the ISA (x86 stack grows downwards), which stores local/volatile variables, as well as the parameters of a routine, and the return address of the calling function.
    - The code segment is essentially a bunch of read-only code that is to be executed by the machine after being translated into machine language instructions. They contain the output of the linkage editor, which is a load module that will be used to be stored in the program loader
    - The data segment stores initialized data (think of int x= 10) and is read/write only
    - Shared libraries segment: This is loaded into a process' memory space in order to improve efficiency in retrieving library calls and subroutines. It is executable and read-only and are loaded into/executed at run-time
    - thread stacks: each thread has its own stack within a virtual address space to store their own local variable values and parameters. These are read/write-only and probably not managed by the OS

Stacks/Linkage Conventions

- What is stored in the stack of a process?
    - local variables are stored in volatile registers by the caller
    - The stack also stores the return calls of the function
    - new call frames are pushed into the stack whenever a procedure is called, and old ones are popped off whenever they are returned
- What functionalities do the caller and callee support?
    - The caller's job is to support the storing of volatile registers that essentially act as garbage registers when they are returned
    - The callee's job is to store non-volatile registers which store persistent information
- Describe a basic linkage convention
    - Although linkage conventions are highly ISA specific, this is the x86 architecture linkage convention
    - The OS loads code and any static data into memory located in the process's address space
    - Before the routine, you must retrieve the information that is passed in as parameters in order to store on the stack. There must be a subroutine call made to store the return address of the function on to the stack, and then transfer

control to the entry point. The registers of non-volatile information and variables must be stored by the callee so that they can later be restored when the function is returned. Then we must allocate enough space on the stack in order to store all of the local variables necessary
    - When we return from the routine, we place the return value of the routine in the address where we expect to find it, and then pop off the local storage from the stack. Then we restore the information stored in non-volatile registers before the routine started. Then, we return execution to the point of entry before the stack (at return address)
    - Cleaning parameters off the stack is the caller's responsibility

Process States

- What are the process states
    - The three primary process states are running, ready, and blocked
    - There may be a zombie state such as in Linux, which represents that a process has finished execution but has not cleaned up yet
    - The state of running means that the process is currently in execution and the CPU has control over it -> operating system does not work when the CPU has control over the process
    - Ready processes are processes that are good to go for execution but are not currently executed -> usually stored in something like a queue -> see MLFQ later
    - Blocked state represents that there is something preventing the process from being able to run
    - When a state goes from ready to running -> scheduled
    - When a state goes from running to ready -> descheduled
- What can cause a process to be blocked?
    - Executing privileged instructions and waiting for a response from the OS/kernel can cause a process to be blocked. Waiting for I/O operations to complete or an initiation of a trap handler or interrupt can cause this
- What is the difference between a resident process state and a non-resident process state?
    - A resident state is a state that can be needed at any time. For example, the process id, user id, and group id (identification information), **information needed to schedule processes** (run-state, priority, statistics, data needed to signal/awaken a sleeping process), **communication with synchronization resources** such as locks, mutexes, condition variables, semaphores, and signals. **Also includes a pointer to non-resident states**
    - A non-resident state is a state that is not necessarily needed at all times, meaning that it can be swapped out to free memory. Execution state including supervisor mode stack, local variable registers, PC and PS (program counter and program status), pointers to text, data, and stack segments
    - Summarized answer: A resident state is a state that may be needed at any time. This includes identification information such as process ID, user ID, and group

ID, as well as information that the scheduler needs to know about the process such as its running state and priority in MLFQ (with priority boosting), and synchronization tools such as locks and condition variables and semaphores + signals. It also contains a pointer to non-resident state information. Non resident states include information that may not be needed at all times and can be swapped out in memory to free space (think of page swapping later). This includes information such as execution state (local variable stack registers and information, PC, and PS) as well as pointers to data, text, and stack segments
    - Resident states are stored in physical memory where it needs to be accessed all the time, while non-resident states can be stored in the disk where it can be swapped out in order to retrieve the information

User-Mode vs Supervisor Mode

- When would a program switch from user-mode to supervisor-mode?
    - A process would switch from a user-mode to supervisor-mode when it needs to execute privileged instructions with the OS. Examples include initiating disk I/O or wishing to allocate memory through commands such as malloc and sbrk.
- Why is it necessary for context switches to occur?
    - Context switches must occur in order to achieving limited direct execution. Because the OS relinquishes control to CPUs when a process is running, it cannot manage when to switch from one process to another without context switches
- How are context switches made?
    - Context switches are essentially made through system calls or trap/interrupts depending on what kind of limited direct execution it is using
    - If it is using a cooperative approach, the OS will be passive and simply wait for a system call to be executed or simply finish before executing a context switch
    - In a uncooperative approach, the OS has more control over which process is running. A timer interrupt may be implemented in order to halt execution if a process has been running for longer than the allotted time.
    - When context switches are made (think of linkage conventions), the OS must execute a low-level piece of code in order to save the current information we have to registers onto the kernel stack

Traps (exceptions) vs Interrupts (async calls)

- What is the difference between a trap and an interrupt?
    - In the x86 architecture, a trap is an exception in a user process that is used as a way to invoke a system call. They are handled synchronously, meaning that execution of a process is halted until the control is returned to the CPU and occur due to errors such as arithmetic overflow and dividing by zero. Traps are used as a way to call kernel routines from the software side of the applications. Interrupts occur at random times within the program's execution in response to signals from the hardware. Interrupts are used to handle events external to the processor

such as servicing peripheral devices such as the disk. Timer interrupts to issue context switches between processes and allow the OS to take control to decide what to do.

- *note: not a question just something important to remember
    - each trap/interrupt has a numeric code associated that is used to index into a table of PC/PS vectors
    - The PC is used as an index to a PS vector and the new PS is taken from the vector
    - previous PC/PS would be stored in the stack
    - new PC is loaded from the selected vector
- What is the difference between a procedure call and an interrupt or a trap?
    - A procedure call in requested by the running software, and the calling software expects that, upon return, some function will have been performed and the appropriate values returned
    - Procedure calls are initiated by the software while the trap/interrupt handlers are handled by the hardware
- Explain the steps for the trap/interrupt handling process (Question)
    - There is a number associated with every possible external interrupt or execution exception (interrupts/traps)
    - The numeric code associated is called a PC that is used as an index for a interrupt/trap vector table that stores PS (processor status)
    - The CPU loads a new PC and PS from the vector table
    - The CPU pushes the previous PC and PS onto the CPU stack
    - execution continues at the address specified by the PC
    - The selected code (first level trap handler) saves all of the general registers on the stack, gathers information from the hardware on the cause, chooses the appropriate second level handler, and makes a normal procedure call to a second level handler that deals with the interrupy/exception
    - After the second level handler deals with the event, it returns to the first level handler
        - first level handler restores all of the saved registers, executes a privileged instruction to return from the interrupt, re-loads the PC and PS stored in the CPU stack before the trap instruction and execution resumes at the point of interrupt

Limited Direct Execution

- How does limited direct execution help achieve virtualization of the CPU?
    - Limited direct execution performs context switches between user-mode and supervisor-mode in order to choose which process needs to be run. The whole concept of virtualization is to make it seem as if each process has its own CPU, while using time sharing to achieve this. Limited Direct Execution, depending on which approach is used (cooperative vs uncooperative) relinquishes control of the CPU to the opearting system so that it can make the decisions. The goal is to

enter the OS as seldom as possible. Use traps for system calls and interrupts for external hardware interrupts. (achieving time sharing)

- Difference betweeen cooperative and uncooperative approach (explained above)
- How is limited direct execution achieved?
    - At boot time, the kernel loads the PC/PS vector table to set up trap/interrupt handling by executing privileged instructions.
    - Then, the kernel sets up a few things such as the allocation of memory or the allocating of nodes on a processor list before executing a return-from-trap instruction in order to start execution of the process
    - Summarized: At boot time, the system is in kernel mode in order to load things like the PS/PC table, allocate memory, and create the processor list, and then calls a return from trap in order to go to user mode and begin execution

Fork vs Exec

- What is the fork() call used for?
    - The fork() call is used to create a child process that runs the same program as the parents program if exec() is not specified
- What is the difference between the child and parent process?q
    - The child process is not able to send signals to the parent process but the parent process is able to kill/stop the child process. The child receives an exact copy of the parent's address space, but gets its own virtual address space. Therefore, besides the process ID, the child and parent processes are extremely similar
    - However, the CPU scheduler determines which process runs at which time so we do not know whether fork immediately executes the child or the parent
    - Remember that the parent and child share one read-only code segment when fork() is called
- Why is the exec() function used with the fork() function
    - the exec() function is used with the fork() function so that code can be executed before the exec() function and after the fork() function
- What does the exec() function do?
    - The exec() system call lets users execute a different program from the parent process

Linkage Editing

- What does the linkage editor do?
    - The linkage editor turns a relocatable object module into a load module for the program loader to execute. It does this through resolving the unresolved external symbols and references through searching for library routines that satisfy these requirements. Then loads the text and data segments of those object modules into the virtual address space and note where the symbols are referenced. Go through all of the relocation entries referenced in the object modules and relocate entries to reflect the correct chosen address

    - Once all relocatable addresses are relocated and all unresolved references are resolved, the load module created by the linkage editor will be ready to move to the program loader
- What is contained in the object module that is received for linkage editing?
    - Many object modules have different formats specific to their ISAs. The ELF format in UNIX/Linux platforms includes a symbol table or **a list of unresolved external references, a head section, code and data sections, and a collection of the relocation entries which includes the location of the field** (in a code or data section that requires relocation), the width/type of the field to be relocated (32/64), the symbol table entry, whose address should be used to perform that relocation
    - *.o files represent the object modules that are created by the linkage editor

Syscalls vs Procedure Calls

- What is the difference between procedure and syscalls?
    - System calls call privileged instructions within the OS, which provides an interface for user-mode communication. However, procedure calls are simply done within user-mode code without executing privileged instructions

Static Libraries

- What are static libraries?
    - Static libraries are libraries (series of subroutines/reusable code) that are directly and permanently incorporated into the load module. However, because this library maintains permanence, we are not able to receive the newest version of a library if it is already loaded into the address space. Therefore, we would need to reload the new version again -> increases overhead and leads to memory problems

Shared Libraries

- What are shared libraries?
    - Shared libraries are run-time loadable libraries that is used to always get the newest version of an available library. This is done through reserving a space for the library in physical memory (usually a segment)
- How are shared libraries created?
    - We reserve an address space to store the shared library segment
    - We linkage edit the shared library with the read-only code segments, loaded at the address reserved for that library
    - Assign a number to each routine within that library, containing the addresses of each routine in the shared library to be filled in by the linkage editor (external reference resolution)
    - Create a stub module/library that is really **a table of addresses** that you should call (i.e., a table of entry points)

        - the linker leaves some stubs/unresolved symbols to be filled at application time
    - We then linkage edit the stub module with the client program
    - When the OS loads the program into memory, it realizes that a shared library is being used and resolves the dependencies by mapping the shared library segments into the program's address space at the appropriate location
- What are the advantages and disadvantages of shared libraries
    - Advantages
        - You are always guaranteed to get the newest version of the required library
        - You are able to keep one copy of this shared library in memory so that numerous processes can access it
    - Disadvantages
        - They are read into program memory whether you want to use it or not -> wastes memory
        - The name of the library must be known at linkage time and only the routines that are called are delayed until load time -> therefore shared library is always loaded into memory

Dynamically Loadable Libraries

- What are they?
    - Dynamically loadable libraries are libraries that are loaded at run-time, meaning that their binding is deferred until they are actually needed by the client program
    - This means that the newest versions of the library are always available AND we do not always need to keep a copy of the library in memory. Once the library is done being used, it will detach the binding to that library
- What are the differences between DLLs and Shared libraries?
    - The primary difference between a DLL and a shared library is that shared libraries do not have a special linkage editor while a DLL requires a run-time loader and a program linkage table. While shared libraries consume memory for the entire time the process is loading, DLL's only consume library when the client calls for it.
- What are the major capabilities that come with DLLs?
    - The ability to defer loading until the modules are actually called by the client
    - The ability to open and load run-time modules that did not exist at linkage time
    - The ability to perform per-module initialization and shut down
    - The ability to resolve references in the load module back to the main program
- What are the exploitations of these capabilities
    - Deferred binding can significantly improve performance -> if a program seldom uses a library (only needs a certain routine), then it could be beneficial to use DLLs so that the program can get the benefit of the modules that become available after the program begins execution
    - Explicit selection and loading can be exploited by the use of browser plugins

    - Per-module initialization could be used to allocate and initialize data, register instances, etc. Device Drivers require both of these
    - Device drivers exploits solving references in the load module by making calls back into the main program
- What is the primary difference between shared libraries and DLLs
    - The primary difference is the use of a run-time loader. This is because, obviously, the library must be loaded in at run-time
- How are dynamically loadable libraries implemented?
    - The application chooses a library to be loaded at run-time, then the OS loads the library into memory. The OS loads a few entry points (addresses) for library routines, and the application calls the entry points to bind the DLL with the code. When the application does not need this library, it unloads the library by shutting down. This way, we are able to ensure per-module initialization and shut down
    - (implicit approach) Applications are linkage edited with a stub library (like in shared library implementations), which creates a PLT entry (program linkage table) that issues a system call to a run-time loader. The first time one of these entry points are called by, the PLT issues a call to the RTL to retrieve the module. The PLT entry is changed so that it refers to the address of the loaded library routine
-

Signals

- Will calling kill() terminate an operation
    - No, calling kill() will not terminate an operation
- Describe the steps for user-mode signal handling
    - The operating system has numerous types of signals such as exceptions, operator actions, and communications
    - The processes can control how they handle these signals (e.g., ignore them, designate a handler for them, default action such as kill)
    - analogous to hardware interrupt but implemented by the OS and delivered to user mode processes
- Explain the signal handling process
    - When a program raises an asynchronous exception, an exception handler is called
    - Invocation of this exception handler is similar to a procedure call, as it saves the state, then handles the exception, then returns to the interrupted computation
    - However, it is much more complex than a procedure call as it must save the state in volatile registers as well as restore them, but it also may abort instead of return
- What is an asynchronous exception?
    - Unlike some errors like EOF and arithmetic overflow, these exceptions occur at random times (segmentation faults, user aborts (^C), and power failure). These raise asynchronous exceptions that are handled through support for try/catch systems, traps, and system calls (uses exceptions as a way to access the OS)

Lecture 4

Execution State Model

- What are the execution states of a process?
    - Running: process is currently being executed
    - Ready: process is ready to be executed
    - Blocked: process is blocked from execution for some reason such as I/O requests and memory allocationo
    - Swapped out: Process resides in disk in order to make use of limited memory
    - Swap wait: Process is waiting to be swapped into the ready state (physical address)
- How do you dispatch and undispatch a running process
    - We undispatch a process first entering the OS through some kind of trap or interrupt/via a yield system call, then we save the PC, PS and the state of the process onto a user-mode stack (supervisor mode states are also stored on the supervisor stack). Descriptions of the address space, pointers to code, data and stack segment, etc are all stored in the process descriptor -> then we yield the CPU and let the scheduler choose the next process
    - We then re-dispatch a process through restoring the saved registers, stack and code segments, etc. and then restoring the PC and PS of the process -> resume execution of a new process
- Why do we swap?
    - We want to make best use of limited memory, so processes that are blocked or are not needed should be stored in secondary storage (like a disk) until they are actually needed. This also improves CPU utilization because whenever there are no ready processes, the CPU is idle. Idle CPU time is wasted, so we want the CPU to be active. However, swapping processes is very expensive so we want to do it as little as possible
- Explain the process of swapping in and out
    - The process' state is in main memory, including the code and stack segments as well as the pointer to the non-resident state descriptor. Before swapping out, we want to copy these out to secondary storage and update the resident process descriptor. The process is then no longer in memory and we now have a pointer to that process in secondary storage.
    - When we want to swap back in, we reallocate the memory to contain the process (code and stack segments + non resident state descriptor), read the data from secondary storage, and change the process state to ready. The saved registers are on the stack and the user-mode stack is saved in the data segment. The supervisor-mode stack is stored in the non-resident state descriptor, so therefore the swapping process takes a lot of I/O and is very expensive

Metrics: Completion Time

- How is the completion time metric used to determine efficiency?
    - The completion time metric is used to determine efficiency through measuring how much time it takes to complete to decide which process should run when. Algorithms such as shortest-job-first and shortest time to completion first offer great efficiency in the completion time department

Metrics: Throughput

- What is throughput and how does is it used to determine which process should be run?
    - Throughput is the number of operations within the process/second. It is closely related to completion time because a higher throughput could possibly mean a higher completion time. The throughput can be used to determine which schedule runs first through algorithms similar to completion time efficient algorithms such as SJF or STCF

Metrics: Response Time

- What is response time and what algorithm produces the best response time?
    - The round-robin algorithm produces the best response time while. The response time is measured as the time between when the process arrives and when it is executed or time spent on the ready queue. Response time and throughput/completion time are opposites (if one is good the other is bad)

Non-preemptive scheduling

- What is non-preemptive scheduling
    - Non-preemptive scheduling is when the OS does not preempt the execution of a process and simply yields the CPU to the OS
- What are the advantages and disadvantages of non-preemptive scheduling
    - Advantages
        - Simple to implement, less work on the OS's part because it simply has to wait for a process to yield the CPU
        - Works well with smaller, simpler systems because of its simplicity
    - Disadvantages
        - A piggy process can monopolize the CPU and cause others to starve
        - A buggy process can lock up the entire system
- When should non-preemptive scheduling be used?
    - Non-preemptive scheduling can be used in real-time systems where we do not want to preempt a process because it can lead to disastrous events
    - Non-preemptive scheduling can also be used on simpler, smaller systems that do not have many processes
- What are some common non-preemptive scheduling algorithms
    - FIFO: the process that enters the queue first is executed first
        - Advantages
            - extremely simple to implement
            - all processes are eventually served

            - intuitively fair
        - Disadvantages
            - Convoy: Larger processes that enter at the beginning of execution can hog the CPU and starve other processes
            - high variable response time
    - SJF: shortest job first
        - Advantages
            - Once again, fairly simple to implement
            - likely to yield the fastest response time
        - Disadvantages
            - Some processes may face unbounded wait times
            - It is hard to have the ability to know the estimated amount of completion time
    - STCF: Shortest Time to Completion First: Determines which of the remaining jobs have the shortest time remaining and executes that one
        - Advantages
            - Very good turnaround time
        - Disadvantages
            - Has very bad response time for certain processes because if we have a series of shorter processes, the longer process could potentially starve
    - Priority: all processes are given a priority and the scheduler runs the highest priority processes until they yield
        - Advantages
            - Users can control the priority of the processes, meaning that they can choose which processes they want to run
            - Our “goodness” or fairness metric could be improved depending on our definition
        - Disadvantages
            - Still subject to starvation of lower priority processes
            - per-process may not be fine enough control

Time sharing

- What is time sharing?
    - Time sharing is a method to achieve virtualization of the CPU. It has a very fast response time to interactive processes and is intuitively fair because each process gets the same amount of CPU.

Time slice

- What is a time slice?
    - A time slice is the time that each clock interrupt is executed at in preemptive scheduling algorithms. The timer interrupts the process if the allocated time slice

expires, thus handing the CPU over. The time interrupt must be a multiple of the time slice and thus the slowest possible interrupt rate must be the time slice

Performance Under Load

- As the load increases, the throughput decreases, while as the load increases the response time asymptotically increases (in a typical (not ideal) situation)

Scheduling Goals

- What are the primary scheduling goals?
    - The scheduling goals differ from system to system depending on what they want to achieve
    - TIme sharing systems
        - Must have good response time, each user gets a fair share of the CPU, and execution heavily favors higher priority processes
    - Batch
        - Want to maximize total system throughput
        - Delays of individual processes are not as important as long as we have many operations executing per second
    - Real-time
        - Critical operations must happen on time and they must execute completely. Non-critical operations may not be executed at all
    - Basic metrics
        - Throughput, response time, fairness (subjective), and time to completion
    - Bonus question!
        - What are the goals of time sharing?
            - We want to be able to have good response time so that each process has a fair share of the CPU -> means that higher priority processes are heavily favored
        - What are the goals of batch execution?
            - We want to increase the throughput of the scheduling algorithm, meaning that the delays of each individual process become unimportant
        - What are the goals of real-time systems
            - Because real-time systems have critical processes that can potentially harm the lives of users upon failure, the critical processes must be executed in high priority (on time), and the non-critical processes may not be executed at all

Starvation

- Explain starvation
    - Starvation occurs when a process has not received a share of the CPU in a long time, meaning that a process has an unbounded waiting time
    - This is caused by case-by-case discrimination of processes

- What are algorithms that cause starvation?
    - Priority algorithms without boosting
        - The lower-priority processes can be starved with an unbounded waiting time
    - Shortest Job First
        - Some processes can be subject to unbounded waiting times due to waiting for a series of shorter jobs to finish executing first
- How do we prevent starvation?
    - Priority boosting/guaranteeing that every process is run at some point
    - A strict FIFO algorithm: give credit to processes that have not been executed for a long time, and ensure that all processes/queues are eventually run
    - Basically, we want to ensure that all processes are eventually run

Convoy

- What causes convoy?
    - The convoy effect is when processes are not executed because a long process is in front of them in the queue
    - This is typically a result of FIFO algorithms, as a large, piggy process can hog up CPU time and cause other processes to not be executed
- How do we prevent convoy?
    - We can prevent convoy from occurring through the use of preemptive scheduling algorithms such as round-robin or MLFQ
    - This lessens the amount of time the CPU is idle for
- How do we reduce contention?
    - Eliminate mutual exclusion through making the resources truly shareable
    - Reduce mutual exclusion by implementing read/write locks
    - Reduce contention by breaking up one resource into numerous sub-resources
    - Shorten the critical section or make it used less often
    - reduce the likelihood of preemption through moving potentially blocking operations outside of the critical section
- What are the negative effects of convoy?
    - Eliminates parallelism, slows down execution and efficiency, and reduces system throughput
- What is the key to convoy formation?
    - The key to convoy formation is that processes are no longer able to immediately allocate the proper resources and are therefore forced to block. Once this happens, the mean service time for I/O can exceed the mean request time, eventually filling the queue with longer processes that continuously block

Cost of Context Switch

- Why do we want to limit the amount of context switches that occur?
    - Switching from user mode to supervisor mode can be expensive and can also incur an overhead that could be costly in terms of performance. Therefore, our

main technique to approach this is limited direct execution, where we set up interrupts and handlers so that the OS can take control of the CPU
    - Swapping in and out is also extremely important because of the expensive I/O operations that must occur, as well as hardware support + saving general registers and non-volatile information
    - Context switches also introduce wasted CPU cycles where the CPU is idle because it is waiting for the time slice to finish and therefore have the OS handle the I/O request

Optimal Time Slice

- What is the optimal time slice for an operating system?
    - The CPU share = time slice * slices per second. Based on the CPU share and the natural rescheduling interval (time spent blocking for a specific operation), an optimal time slice can be attempted to be used
- What happens if a time slice is too short?
    - If a time slice is too short, it will spend too much time doing context switches instead of running the application, creating performance issues as well as wasting CPU cycles
- What happens if a time slice is too long?
    - For an application that is very interactive (issues lots of I/O requests), it may be a problem to have long time slices because there will be wasted CPU cycles coming from
- Why is it important to find the optimal time slice?
    - If a time slice is too short -> many context switches occur -> reduced performance + idle CPU. If a time slice is too long -> CPU waits for interrupts to occur in an interactive system -> causes delays

Graceful Degradation

- What do we do when a system is overloaded?
    - When an OS is no longer able to meet its service goals, we want to continue to resume execution at a degraded performance and drop requests to execute processes until load drops to normal
    - This is called graceful degradation as it slowly degrades until there is an acceptable load
- What do we not want to do when a system is overloaded?
    - We don't want the throughput to drop to zero (all processes stop) or for the response time to grow without bounds

Priority Scheduling

- Explained above but, what is priority scheduling and what are the advantages and disadvantages of it?
    - Priority scheduling is a scheduling algorithm that assigns priorities to certain jobs in order for them to be executed by the OS first

    - Advantages
        - Users are able to control the priority of each process
        - Depending on your measure/definition of fairness, it may be able to achieve it
    - Disadvantages
        - Can still cause starvation of lower priority processes
        - per-process may not be fine enough control

Real Time scheduling

- What systems would want to use real-time scheduling?
    - Systems such as space shuttles would want to use real-time scheduling so that critical processes are always executed
- What are hard real time schedulers?
    - Hard real time schedulers ensure that the critical processes are always run without preemption. Hard real-time algorithms do not use preemptive scheduling. The non-critical processes may starve but that is okay.
    - We want to avoid non-deterministic outcomes such as interrupts or garbage collection
    - We want to emphasize predictability over speed, as it is important for us to meet deadlines
    - Scheduling order may be hard-coded so that the algorithm always follows that specific order
- What are soft real time schedulers?
    - Soft real time schedulers are sometimes called “best effort” schedulers, as they do their best to meet the deadline set. Compared to hard-real time, these deadlines are not as important and are not enforced.
    - All tasks do not need to run to completion
    - If a higher priority task arrives, a process may be preempted in order to run the higher priority one first
- What are some soft real-time algorithms?
    - Most common algorithm is Earliest Deadline First
        - Assign a priority to each job based on how close it is to the deadline
        - Sort and store these in a queue and always run the first job on the queue
- What are examples of some soft and hard real time schedulers
    - A video playing device (soft)
    - A nuclear power plant (hard)
- What makes real time systems easier to schedule?
    - The amount of time that each process takes may be known
    - You can hard code which processes you want to run first in what priority
    - Starvation of non-critical processes is acceptable

Preemptive Scheduling

- What are the advantages and disadvantages of preemptive scheduling?

    - Advantages
        - Avoids convoy through preempting processes and ensuring that each process runs
    - Disadvantages
        - Introduces potential resource management problems
        - Context switches need to be executed, leading to performance issues
- How can an executing process be preempted?
    - If a process of higher priority arrives and thus is dropped to a lower priority queue
    - If it uses up the time slice specified by the OS
    - If the priority of the job drops
- How can executing processes be blocked?
    - If an I/O operation is executed and the resource is not immediately available
    - If we need to swap a resource out from secondary to primary storage since a process cannot run until it is available in memory
- How can we force processes to yield the CPU?
    - Clock interrupts/timer interrupts and system calls can be used to relinquish control of the CPU

Round Robin Scheduling

- How is round-robin executed?
    - The Round Robin algorithm is a preemptive scheduling algorithm that assigns time slices for a certain process to be able to run and once the process uses up that time slice, selects another process
    - The algorithm runs processes in a circular queue
- What are the advantages and disadvantages of RR?
    - Advantages
        - Greatly reduces time from ready to running -> response time
        - intuitively fair
    - Disadvantages
        - Some long processes will need many time slices
        - extra interrupts and context switches can cause overhead

Multi-level Feedback Queues

- What are the advantages of MLFQs
    - MLFQs provide a good in between for response, throughput, and turnaround time
- Disadvantages
    - MLFQs are difficult to implement, require extra context switches -> basically the disadvantages of preemptive scheduling algorithms
- What are the rules for an MLFQ (Arpaci's)
    - If a priority (A) > priority (B)
        - run A
    - If priority (A) = priority (B)
        - run A and B in round robin

    - When a job enters the system, it is placed in the highest priority queue
    - Once a job uses up its time allotment in the queue, its priority is dropped (could be preempted to drop)
        - This is important in order to prevent gaming of the CPU so that one process does not hog all of the CPU time
    - After some time period, move all of the lower priority processes to the highest level queue in order to avoid starvation
- What are the rules for a dynamic MLFQ (Kampe's)
    - Instead of assigning priority directly, create a more interactive/dynamic MLFQ
    - Each queue is separated in regards to how many yields it can do, the amount of time you have spent in the queue, and the time slice that is allotted for each round robin queue
    - By building credit for the amount of time a process stays in a queue, we can avoid starvation by making sure that all processes are eventually run
    - If you stop running before preemption, it still counts as a yield
    - Pretty much, if you exhibit good behavior, you are moved to a better queue while if you don't your priority is dropped

Dynamic Equilibrium

- What are the two opposing forces that drive dynamic equilibrium in scheduling?
    - There exists a dynamic equilibrium between competing processes and negative feedback
    - Therefore, dynamic equilibrium is the net result of these processes and are always self-calibrating and able to adapt to changing circumstances
    - example: priority boosting and dropping in MLFQ

Mechanism/Policy Separation

- Why is mechanism policy separation important in schedulers?
    - The main goal in mechanism policy separation is for the mechanisms to not overly constrain the policies
    - In this case the mechanism is how we run the process, or for MLFQ, it is to always run the highest priority processes and the formulae to compute the optimal time slices for each queue and priority
    - The policy is controlled by the user and can specify initial, minimum, and maximum time slice length, queue in which each process should be started (based on ID, etc), the CPU share, and what causes priority changes (exhibiting bad behavior through hogging up the CPU (# of yields) or going over their allocated time slice

WEEK 3

Lecture 5

Physical Address Space

- The physical address space is the primary storage unit of all executing processes, we achieve virtualization of memory through swapping out unnecessary/blocked processes and making it seem as if though each process has its own virtual address space
- What is stored in the physical memory?
    - Physical memory contains pretty much everything that virtual memory address spaces hold (process information and segments such as stack and code and data)

Virtual Address Space

- What are the goals of virtual memory?
    - transparency: we want to make it seem as if this illusion that is provided is actually real and thus invisible to the application
    - Efficiency: The OS should strive to make virtualization of the CPU as efficient as possible -> rely on hardware support such as the TLB (cache)
    - Protection: The OS should make sure that each process address space is independent and that one process does not step on another process' segments

Text Segment

- What is stored in the text segment ? Are there any special privileges?
    - The text segment, also known as the code segment, stores the instructions that need to be executed in order to run the program. It is read-only and executable

Data Segment

- What is stored in the data segment? Are there any special privileges?
    - The data segment stores all initialized variables, and has read/write permissions, meaning that it can be constantly changing, the heap is also in the data segment

Stack Segment

- What is stored in the stack segment? Are there any special privileges?
    - The stack segment stores the stack of the process (registers, parameters, return values, address of the returning function), and is read/write only because we do not want to share the information held on the stack with other processes
- *Other segments include thread stack segments and shared/DLL segments as mentioned in a previous question

Protected Memory Sharing

- Why is important to protect memory of one process from another?
    - If one process is able to access elements of another process, it may be able to potentially exploit buggy code and cause related processes to halt

Internal Fragmentation

- How does internal fragmentation occur?
    - Internal fragmentation occurs from fixed partitioned memory allocation. This occurs because within a process' address space, memory may not be used up entirely, meaning that there is internal space that is wasted
- How do you prevent internal fragmentation?
    - We can prevent internal fragmentation by using algorithms such as the best fit algorithm or by coalescing adjacent chunks of memory
- What are the advantages of fixed-size memory allocation? What are the disadvantages?
    - Advantages
        - It is extremely easy to implement
        - Efficiency is improved because you can simply just select the first one off the list
    - Disadvantages
        - Causes internal fragmentation
        - Inefficient use of memory
        - swapping results in convoys on partition

External Fragmentation

- How does external fragmentation occur?
    - External fragmentation occurs when memory allocation creates leftover chunks that are too small to use, meaning that fragmentation occurs and these chunks will not be used
- What are the advantages and disadvantages of Variable Partition Allocation?
    - Advantages
        - eliminates internal fragmentation
    - Disadvantages
        - Could be slower because some algorithms require the OS to traverse through the entire free list
        - Causes external fragmentation build up
- How do we prevent external fragmentation
    - Three approaches
    - 1) Avoid creating tiny segments that cause external fragmentation in the first place
    - 2) Coalesce adjacent pieces into larger, more usable pieces
    - 3) Defragmentation: Copy these small pieces of memory into another part of physical memory, then rearrange/coalesce so that the chunks can be used efficiently
- In variable sized partition, what information do we need to have?
    - We need to know the size of each chunk, the location, and the neighbors (for coalescing). For fixed-sized partitions these values will all be constant

Common Errors with Memory Management

- Questions on free()

    - How does free() know how much memory was allocated when the malloc() call was called if it only has one parameter to be passed in -- a pointer that was returned by malloc()?
        - When allocating memory, the pointer returned by malloc() actually allocates a little bit more memory than it needs to by constructing a header section that contains information about the allocated memory like the size
    - What happens when we forget to free memory that we malloced
        - This causes memory leaks in the program we made and eventually we will run out of memory
    - Does a garbage collected language help with memory that we forget to free
        - Yes to a certain extent but no. A garbage collection is only run when the OS realizes that there isn't much memory left. If you have a reference to some chunk of memory, no garbage collector will be able to find it
    - What happens when you free memory repeatedly?
        - This is called a double free and will result in undefined behavior
    - What happens when you free memory before you are done using it?
        - It will result in a dangling pointer that is never used again and cause dangerous behavior such as overwriting memory that you previously had
    - Why is no memory leaked when a process exits?
        - The OS will reclaim the memory of the process when the program is finished (stack, code, heap segments) to ensure that no memory is lost. This means that for short programs it could be possible for you to forget to free memory -> bad coding practice don't do it
        - Remember that purify and valgrind are two common tools that are good for detecting memory leaks
- What are some common errors that are made with memory management + consequences?
    - Forgetting to Allocate Memory
        - A segmentation fault will occur because the routine will not know where the memory location is
    - Not Allocating Enough Memory
        - remember buffer overruns can potentially cause dangerous behavior such as stack smashing -> accessing the information of other processes + security vulnerabilities
    - Forgetting to initialize allocated memory
        - Can cause uninitialized read, where it reads random data with unknown value from the heap
    - Forgetting to Free Memory
        - discussed earlier: memory leaks
    - Freeing memory before you are done with it
        - discussed earlier: could overwrite valuable information you previously had
    - Freeing memory repeatedly

        - double free: undefined behavior
    - Calling free() incorrectly
        - can confused the memory allocation library
- Explain the difference between malloc() and sbrk()
    - malloc() is a library call and sbrk() that is a system call made to allocate more memory
    - sbrk() manages the size of the data segment by specifying the ending address
    - malloc() manages the size of the heap (dynamically allocated variables)
    - sbrk() can be used within a malloc library call to increase/decrease the size of the heap

Address Space Layout

- Explain a basic memory address space layout
    - Virtual address spaces typically include the data, code, stack, and heap segments as well as shared library segments

Stack vs Heap Allocation

- What is stored in the stack? What is stored in the heap?
    - The stack stores local variables and parameters within the program while the heap contains dynamically allocated variables created through calls to malloc() (in C) and new (in Java, C++)
- What are the primary differences between stack and heap allocation
    - For stack allocation, the compiler manages the space, data is volatile as it is only valid until stack frame is popped, and memory allocation is automatically managed by the OS (growing and shrinking). However, in heap allocation, the data segment size is adjusted through calls to sbrk, the variables exist until free() is called/garbage collected, and heap space is managed through a user mode library such as malloc()
- What do you do when the heap runs out of space?
    - Page fault handlers can be run / just return NULL

Coalescing

- How does coalescing improve external fragmentation
    - By coalescing adjacent pieces together, small pieces of memory can be combined into a larger, more usable piece that limits the amount of external fragmentation that occurs
- What are the advantages and disadvantages of coalescing
    - Advantages
        - improves external fragmentation
    - Disadvantages
        - Extra information is needed about the pieces of memory
        - Only adjacent chunks of memory can be coalesced
        - Coalescing is an expensive process

- How does fragmentation and coalescing counteract each other (explain with dynamic equilibrium)
    - The rate at which fragmentation occurs and the rate at which coalescing can be done counteract each other as opposite forces, as explained by dynamic equilibrium

Free List Design***

- What does our free list of memory need to contain?
    - a header to store information about which the size of the malloced() memory
- What optimizations should we have in mind when designing a free list?
    - We want to make it easy to coalesce neighboring pieces
    - Must make optimizations of breaking segments into smaller pieces
    - searching for a piece of the desired size
- What are the types of diagnostic information that we can use in designing a free list?***
    - Standard chunk header
        - free bit, chunk length, and next chunk pointer (for coalescing)
    - allocation audit info
        - tracing down the source of memory leaks
    - guard zones/canaries
        - detecting application buffer overruns
    - zero memory when it is freed
        - detect continued use after chunk is freed
- How can diagnostic free lists help with determining errors?
    - We are able to retrieve the source of the errors
    - Enables us to find the state of all memory chunks through the chunk header
    - Record of who last allocated/used the chunk through malloc
    - Guard zones at the beginning and ending of chunks detect buffer overruns/underruns
    - Valgrind is a tool that checks for memory mistakes using these properties

First Fit

- Describe the advantages and disadvantages of first fit + when it should be used
    - Advantages
        - may not need to scan through the entire free list
    - Disadvantages
        - internal fragmentation can occur
        - Searches become longer because the smaller segments populate the beginning of the free list
        - External fragmentation causes the algorithm to quickly slow with age
    - When it should be used
        - A short program that may not need to use that much address space -> before external fragmentation becomes an efficiency problem

Best Fit

- Describe the advantages and disadvantages of best fit + when it should be used
    - Advantages
        - might find the perfect fit
    - Disadvantages
        - A full, exhaustive search needs to be done to find the best fit
        - Builds smaller, unusable chunks of memory very rapidly -> fast fragmentation
    - When should it be used
        - When you know that there is always gonna be a perfect fit chunk of memory

Worst Fit

- Describe the advantages and disadvantages of worst fit + when it should be used
    - Advantages
        - Tends to create very large fragments that are more reusable -> delays external fragmentation
    - Disadvantages
        - Still is an exhaustive search so it is inefficient
        - Just don't use it (according to Arpaci)
    - When should it be used
        - Aside from never, maybe when you know that your short program is going to use little memory so that external fragmentation doesn't occur at all

Next fit

- Describe the advantages and disadvantages of next fit + when it should be used
    - Advantages
        - short searches + creates random sized segments so that they can be used by a wider range of programs
    - Disadvantages
        - Efficiency could possibly be as bad as the first fit
        - Searches become longer
        - First chunks fragment faster and ultimately the efficiency drops
    - When it should be used
        - Kinda like first fit, so it searches wherever it last left off, so when we want variable sized partitions (fairly random) it would be nice to use

Slab Allocation

- How can slab allocation improve efficiency
    - Slab allocation uses a second pool of fixed-size memory spaces that can efficiently be called from by simply popping the first segment off the queue
    - This also reduces external fragmentation
- How is balancing the buffer pool a dynamic equilibrium problem?

    - demand for special purpose pools are constantly changing + memory needs to migrate between them -> opposing forces auto adapting and calibrating
- Explain slab allocation and how it works
    - Slab allocation is an algorithm that aims to be efficient in allocating memory
    - Relies on the idea that initializing and destroying objects can be extremely expensive
    - Maintains a slab of memory (cache) of popular memory sizes (+ holds popular data structures) and caches an initialized state of the object (data structure) so that it does not have to be created and destroyed every time
- What does a slab allocator do when it runs out of memory?
    - Simply asks the main memory for more slabs

Garbage Collection

- Explain the steps of garbage collection
    - *remember that garbage collection is only possible in languages that are able to keep resource references
    - 1) Begin with a list of all the resources that originally existed
    - 2) Scan the process to find all resources that are still reachable
    - 3) Each time a reachable resource is found, remove it from the original list
    - 4) At the end of the scan, anything that is still in the list of original resources is unreachable and no longer referenced by the project thus can be freed
    - 5) After freeing the resources that are no longer referenced by the program, we can resume execution
    - Garbage collection is only initiated when the amount of space we have is dangerously low
- What are the advantages and disadvantages of garbage collection?
    - Advantages
        - saves a lot of time for users as memory management becomes easier
    - Disadvantages
        - Gets worse with time
        - Non-deterministic cleanup of resources -> if you want the memory to be freed immediately, it does not do a good job
        - Only works if it is possible to find all active references through some kind of tag
        - There is an overhead for keeping track of all of the resources
        - program must half execution in order to initiate garbage collection
- When is garbage collection worst done?
    - When there is most contention. The optimal time would be to use it when there is least collection but garbage collection does not work like that
- What is the difference between GC and Reference Counting?
    - In reference counting, you increment the count on each reference creation and decrement the count on each deletion
    - delete objects when reference count hits zero

    - Different from GC because it needs explicit close/release (free) operations, doesn't involve searching for unresolved references, and correct count maintenance may not be free

IMPORTANT EXTRA QUESTION

- Steps for achieving defragmentation

Lecture 6

Stuff from Dmitri's possible questions

- How does the CPU know which mode it is running in?
    - There is a bit stored in some kind of PS that tells the CPU which mode you are running in
- What is the hardware's role in memory management?
    - The hardware can set base and bounds registers so that it sets limits on how much memory can be allocated (ties into concept of paging later)
    - Memory operations are often privileged so therefore need hardware support in order to be run
    - In cases like exceptions like segfaults the program traps the syscall and the OS needs to execute the error handling code with the help of the hardware
    - CPU/hardware needs to raise an exception whenever it tries to access privileged instructions
- What do we do when coalescing fails?
    - We need a way to rearrange active memory to re-pack all processes in one end of memory and create one big chunk of free space in the other end
    - Memory compaction moves a process (relocation): This makes the allocated memory more compact and allows us to coalesce the free space to cure external fragmentation (defragmentation)

Swap Space

Base/Limit Relocation

- How does a base/limit register controlled by the hardware ensure protection?
    - The register pair allows us to place the address space wherever we would like to in physical memory, while ensuring that the process can only access its own address space
- Where are the base and bound registers stored?
    - Base and bound registers are always stored on a chip called the MMU (memory management unit)
- What are the issues with a simple base/bound pair implementation?

    - The OS has a lot of responsibilities when performing context switches because they now also must set base/bound pairs properly while also performing memory management from the free list, allocation, and reclaiming memory from terminated processes.
    - To move a process' address space the OS must first deschedule the process, then copies the address to its new location while saving the registers
    - Internal fragmentation can still occur because the stack and heap are not too big and therefore not all of the memory is used. This occurs because we restricted ourselves to placing an address space in a fixed size slot
- What are the advantages of a simple base/bound pair implementation?
    - Very simple hardware implementation and also does a good job of ensuring protection and privacy

Page Table Entry

- What is a page table entry and what does it store?
    - Also holds the physical frame number for the corresponding VPN
    - A page table entry is within the page table that holds the translations from virtual to physical address spaces
    - Includes a bunch of bits to give us information about the page
    - **Valid bit:** indicates whether the particular translation is valid or not
    - **Protection bit:** Indicates whether the page could be read from, written to, or executed from
    - **Present bit:** Indicates whether the page is in physical memory or is on disk (swapped out)
    - **Dirty bit:** Indicates whether the page is dirty or has been modified since it was brought into memory
    - **Reference bit:** Used to track whether a page has been accessed -> used to determine which pages are popular and should be kept in memory
    - Can also contain a user/supervisor bit to let the CPU know which mode it is executing in

Page Replacement

- Why is paging and page replacement without the TLB slow?
    - The system must first translate the virtual address to a physical address
    - Assuming that the page table base register contains the physical location of the location of the page table, the hardware must first pick out the VPN bits and then to shift to set the offset
    - Hardware must fetch the desired data from memory and put it into the registers after calculating the PFN
    - Extra memory references in paging increases costs and overheads
- Why do pages need to be swapped/replaced?
    - We need to make best use of the limited amount of memory we have -> processes in primary storage (physical memory/RAM) can run

    - If the process is not ready, it does not need to be in memory, therefore we need to swap it out in order to make room for other processes
- How does swapping improve CPU utilization?
    - When there are no READY processes, the CPU is idle because it is not doing any work
    - Therefore, the throughput is very low and CPU cycles are wasted, so we want to have more ready processes

Segment Addressing

- What is a segment?
    - A segment is a piece of memory that needs to be continuously swapped in/out
- What are the reasons why we would need to move a process?
    - the process needs a larger chunk of memory
    - swapped out, swapped back into a new location
    - to compact fragmented free space
    - all addresses in the program will be wrong
    - it is not feasible to re-linkage edit the program

Translation Lookaside Buffer (TLB)

- Describe what happens when a TLB miss occurs
    - 1) When a TLB miss occurs, the software raises an exception that causes the current stream to stop, raises the privilege level to kernel mode, and jumps to the trap handler
    - 2) Looks up the VPN in the page table and finds the appropriate PFN and loads it into the TLB
    - 3) When the OS returns from the trap, the instruction that causes the trap is rerun and a TLB Hit occurs
- How is the TLB used?
    - The TLB is essentially a cache that stores popular virtual to physical address translations in order to improve efficiency since looking in a page table can be a very expensive process. If there is a TLB hit, the translation can be done very quickly as opposed to looking up a page table entry
- How does the TLB process translations quickly?
    - TLB uses temporal and spatial locality in order to increase efficiency and keeps copies of memory in a small chip
    - temporal locality: the idea is that an instruction or data item that has recently been accessed will likely be re-accessed in the future
    - spatial locality: the idea that an instruction or data item that has been accessed at memory address X will access items near memory address X
- What are the advantages and disadvantages of software managed TLBs?
    - flexibility: The OS can use any data structure it wants to implement the page table without changing any of the hardware

    - Simplicity: The hardware doesn't have to do much besides handle the TLB miss as opposed to a hardware managed TLB having to walk through a page table to find the appropriate entry, etc.

Paging MMU

- Describe what the Paging MMU looks like
    - The paging MMU holds a page table that uses the virtual page number as an index into the page table to retrieve the corresponding PFN. Each PTE contains several bits like the valid, present, permission, reference, and dirty bit in order to retrieve information about the page

Demand Paging

- Explain what demand paging is
    - entire process doesn't need to be in memory to run so it makes sense if we start each process with a subset of its pages and load additional pages as the program demands them
- What are the advantages of demand paging?
    - Fewer in-memory pages per process -> less expensive
    - more processes in primary memory
        - more parallelism + better throughput and better response time
    - less time required to page processes in and out + less disk I/O to be run
    - fewer limitations on process size -> process can be larger than physical memory

Page Faults

- How is a page fault handled? (Previous midterm question)
    - Initialize page table entries to not present
    - CPU faults when invalid page is referenced (page not in memory)
    - 1) trap forwarded to page fault handler
    - 2) Determine which page and where it resides
    - 3) Find and allocate a free page frame
    - 4) Block process, schedule I/O to read page in
    - 5) Update page table point at newly read in page
    - 6) Backup user-mode PC to retry failed instruction
    - 7) Unblock process, return to user mode
    - Other processes can run while this happens
- How can we minimize the number of page faults that occur?
    - Keep the right pages in memory
        - we can decide which pages to evict through a replacement strategy such as FIFO or random
    - Give a process more pages of memory
        - Increase the working set (the amount of pages a process needs)

Belady's Optimal Algorithm

- What is Belady's Optimal Algorithm
    - This algorithm is an optimal algorithm that can never actually be implemented but is used as a benchmark of comparison to see how good other algorithms are. It answers the question of which page should be replaced when we swap out pages (a.k.a. which pages are important?)
    - Belady's Optimal Algorithm selects the page that is going to be used farthest in the future

LRU

- Explain LRU and how it is implemented
    - LRU stands for least recently used and basically selects the least recently used page and places that one into disk
- Why is true LRU hard to implement?
    - Because finding the oldest page is prohibitively expensive
- Describe the steps for using a clock algorithm to implement LRU
    - organize all pages in a circular list
    - position around the list is a surrogate for age
    - run a progressive scan whenever we need another page
        - for each page, ask MMU if it has been referenced
        - if so, reset the reference bit (to 0) and skip the page
        - If not, consider that page to be least recently used
- Does LRU work well with Round Robin?
    - No

Working Set LRU Clock Algorithm

- Why is Global LRU bad?
    - Because it is not compatible with round-robin scheduling. A fixed number of pages to represent the working set (number of pages per process) is also bad because different processes exhibit different kinds of locality
- What is the optimal working set for a process?
    - the number of pages needed during the next time slice
- How do you implement a working set
    - 1) Each page is associated process, and a process has an accumulated CPU time
    - 2) Each frame has a referenced time/bit
    - 3) Maintain a target age parameter
    - 4) If page is younger than the target age, do not not replace
    - 5) If it is older, take it away from the current owner and give it to a nearby process

Page Stealing (dynamic equilibrium)

- Explain the page stealing process
    - Keeps a clock

    - In page stealing, there is a dynamic equilibrium between continuously losing pages that it has not referenced in a while and continuously stealing pages from other processes
    - Processes that reference more pages often will have a larger working set
    - When programs change their behavior, their working set sizes are adjusted due to dynamic equilibrium
    - Manages and avoids thrashing

Thrashing

- Explain what thrashing is and when it occurs
    - Thrashing occurs when memory is oversubscribed and the memory demands of the set of running process exceeds the available physical memory
    - Processes with large working sets can hog memory and result in a continuous set of page faults that drastically slow down the process
- How do we deal with thrashing
    - We can reduce the number of competing processes by swapping some of the ready processes out and ensure that there is enough memory for the rest of them to run
    - We can round robin who is in and out
- Explain pre-loading
    - pre-loading is a page/swap hybrid of pure swapping and demand paging. The difference between pure swapping and demand paging is that for pure swapping, the pages are all already in memory, meaning that there aren't any page faults. However, in demand paging, pages are only brought in when they are needed, meaning there will be fewer pages per process but more processes in memory.
    - If we pre-load the previous working set, there will be far fewer pages to be read in when swapping (less page faults), the same disk reads as pure demand paging and fewer initial page faults than pure demand paging

Clean/Dirty Pages

- What are dirty pages?
    - If a page has been modified since being in memory, it is considered dirty. Therefore many page table entries have a modified, dirty bit that is set whenever the page is written and can be incorporated into a page replacement algorithm. If we write it out to disk, the page becomes clean again
- How can the use of a dirty bit improve efficiency of a page replacement algorithm?
    - Because dirty pages that have been modified, it must be written back to disk in order to be evicted, making it very expensive. Clean pages, however, have free eviction and the physical frame can be used for other purposes without additional I/O. Therefore, some page-replacement algorithms can choose to evict clean pages over dirty ones

WEEK 4

Lecture 7

Stream IPC and Message IPC

- What are the differences between Stream and Message IPC
    - a stream is a continuous stream of bytes that are read or written few or many bytes at a time + stream may contain app-specific record delimiters
        - Byte streams are typically unstructured and depend on the implementation of the parser
    - A sequence of distinct messages that are each is own length + delivery of a message is typically all-or-nothing
- What are the advantages and disadvantages of messages and streams
    - streams do not have any data authentication or encryption to protect them
    - Messages can be sent between machines meaning that processes that are not on the same memory bus can communicate with each other, although there is a loss in performance
    - Additionally, there is better security on messages because the messages are sent in through the OS and therefore the OS can do integrity checks to messages that are buffered in

Shared IPC

- What are the advantages of shared IPC over any other methods of communication?
    - The primary advantage is the sheer speed and performance and the simplicity and how easy it is to use.
    - All you have to do is create a file communication, and have each process map that file into a virtual address space. Have the shared segment locked down so that it is never paged out. Anything written into the shared memory will be immediately visible to all processes that have mapped to their address space
    - You are able to send large amounts of data very quickly without the use of system calls that add overhead, as everything can be done in user mode
- What are the disadvantages?
    - The shared memory IPC can only be used on processes in the same memory bus, a bug in one process can easily destroy another, and no authentication of which process comes from where

Synchronous vs Asynchronous IPC

- Synchronous Operations
    - writes block until message sent/delivered/received
    - reads block until a new message is available
    - easy for programmers but no parallelism

- Asynchronous IPC
    - writes return when system accepts message
        - no confirmation of transmission/delivery/reception
        - requires auxiliary mechanism to learn of errors
    - reads return promptly if no message available
        - requires auxiliary mechanism to learn of new messages
        - often involves “wait for any of these”

Threads

- What is a thread?
    - A thread is strictly a unit of execution/scheduling as it has its own stack, PC, registers
    - multiple threads can run in a process
        - they all share the same code and data space
        - they all have access to the same resource
        - this makes the cheaper to create and run
    - sharing the CPU between multiple threads
        - user level threads
        - scheduled system threads (w/preemption)
- When do you want to use a process over a thread? Vice versa?
    - Process
        - running multiple distinct programs (piping)
        - creation/destruction are rare events
        - running agents with distinct privileges
        - limited interactions and shared resources
        - prevent interference between processes
        - firewall one from failures of the other
    - Thread
        - parallel activities in a single program
        - frequent creation and destruction
        - all can run with same privileges
        - they need to share resources
        - no need to protect from each other
        - they exchange many messages/signals (telnet)
- Name disadvantages and advantages of kernel and user mode threads
    - Kernel implemented threads
        - multiple threads can truly run in parallel
        - one thread blocking does not block others
        - OS can enforce priorities and preemption
        - OS can provide atomic sleep/wakeup/signals
    - Library Implemented threads
        - fewer system calls
        - faster context switches

        - ability to tailor semantics to application needs
- What characteristics must a thread exhibit in order for it to be considered safe?
    - thread-safe routines must be reentrant, meaning that it must be safe to have multiple threads call them + can have concurrent or interspersed execution
    - the state cannot be saved in static variables like errno or optarg
    - persistent session state has to be client owned and have the descriptor passed to all subsequent operations
    - Since threads must operate in a single address space, read only data causes no problems but shared data does
    - Signals are sent to processes and delivered to the first available thread to ensure mutual exclusion
- What are the benefits of parallelism
    - Improved throughput (more operations/sec) meaning blocking of one does not stop others from executing
    - Improved modularity that separates complex activities into simpler pieces
    - Improved robustness meaning that the failure of one thread does not stop another thread's execution
    - a better fit to emerging paradigms

Non-deterministic Execution

- What are some things that cause non-deterministic execution
    - Garbage Collection
    - processes block for I/O or resources
    - time-slice and preemptions
    - interrupt service routines
    - unsynchronized execution on another core
    - queueing delays
    - time required to perform I/O operations
    - message transmission/delivery time

Thread state and thread stacks

- each thread has its own registers, PS, PC
- Each thread must have its own stack area
- max size specified when thread is created
    - a process can contain many threads
    - they cannot all grow towards a single hole
    - thread creator must know max required stack size
    - stack space must be reclaimed when thread exits
- procedure linkage conventions are unchanged
- Unix stack convention: data segment (heap) grows up and stack segment grows down (both grow towards the hole in the middle)

Thread Motivations

- Why do we use threads?
    - Better parallelism increases throughput and system efficiency as well as improved modularity through separating complex activities into simpler pieces
    - Threads produce a better fit to emerging paradigms such as client-server relationships and web based services
    - improved robustness meaning that one thread failure does not cause another thread to fail

Race Condition

- How does a race condition occur
    - When there are competing threads that are not ensure a resource atomically/ achieve mutual exclusion
    - **When the results of a multithreaded program are timing dependent, a race condition can occur**
    - **A critical section is a computation whose correctness is timing dependent**

IPC Goals

- What are the goals of IPC?
    - simplicity, convenience, generality, efficiency, security/privacy, robustness and reliability
    - Simplicity: Pipelines
        - very easy to implement + there are no security/privacy issue
    - Reliability and Robustness
        - Reliable delivery: UDP vs TCP
    - Generality: Sockets
        - connections between address/ports
        - has complex flow control and error handling + trust/security/privacy/integrity
        - Using general networking models allows processes to interact with services all over the world
        - However, you must now deal with security issues of sending messages around the world
        - ensuring interoperability between many different operating systems becomes difficult
    - Halfway: Mailboxes + named pipes
        - Named Pipes: Can be thought of as a persistent pipe that acts as a rendezvous point for numerous processes

            - disadvantages: Readers and writers have no way of authenticating one another's identities
            - Do not enable clean fall-overs from a failed reader to its successor
            - once open it may be as simple as a pipe
            - All readers and writers must be running on the same node
        - Mailboxes
            - Data is not a byte stream -> rather, each write is stored and delivered as a distinct message but also gives some indication of which bytes came from where
            - still subject to a single node/OS restriction
            - Unprocessed messages remain in the mailbox after the death of a reader so that another reader can retrieve it
        - Disadvantages: Client/server must be on the same system
- What are the typical IPC operations?
    - channel creation and destruction
    - write/send/put to insert the data into a channel
    - read/receive/get to extract data from the channel
    - channel content query: how much data is currently in the channel
    - connection establishment and query
        - control connection of one channel to another

Flow Control

- What is flow control and how is it utilized?
    - Flow control is a mechanism used by IPC systems such as pipes and sockets in order to control the amount of data in the queue
    - Since queued messages consume system resources, they are buffered in the OS until the receiver asks for them
    - Many things can increase required buffer space: fast sender, non-responsive receiver
    - Therefore, we must limit required buffer space through blocking senders message if too much (like blocking reading if pace of reading is faster than writing). This is usually handled by network protocols

Mutual Exclusion

- Why do we want to achieve mutual exclusion in concurrency?
    - We want to achieve concurrency to ensure that we have atomicity within the program and that we achieve the best possible result + avoid race conditions. Therefore, we want to ensure that only one thread can enter the thread at a time
- How do we achieve mutual exclusion
    - We can use synchronization mechanisms such as locks, condition variables, and semaphores

Atomicity

- What is atomicity?
    - The main principle behind atomicity is “all or nothing” -- you either want an instruction to execute or not as a unit. Grouping of a single atomic action is called a transaction. We want to make operations atomic in order to avoid race conditions and have a deterministic output

Lecture 8

Asynchronous Operations/Events/Completion

- What are asynchronous operations?
    - Operations that are independent of our own timeline and move at their own pace. These include operations that cannot happen immediately such as waiting for a held lock to be released or waiting for an I/O operation to complete or waiting for a response to a network request and delaying execution for a fixed period of time
- What are some approaches to waiting for asynchronous operations?
    - Spinning: “busy waiting”
        - spinning works well if the event is short (nanoseconds) and is independent (happen on its own timeline), basically it asks “is it there” over and over
        - Spinning continuously checks to see if a condition is true, which wastes CPU cycles and may actually delay the actual event. Also wastes memory and bus bandwidth
    - Yield and spin: “are we there yet”
        - while (status == 0) -> yield
        - this allows other processes access to CPU, meaning that it can access another process on the round-robin queue, wastes dispatching processes, and it works very poorly for multiple waiters
    - Both of these asynchronous operations will still require mutual exclusion
    - ***One of the most common races that occur in software is that between the time you check whether or not something has happened and you put yourself on the list of things to be woken up when this event happens, the event happens in between those two things. Checking and putting things to sleep must be atomic
- How do we use condition variables on asynchronous operations?
    - We can create a synchronization object that pretty much waits for a condition to be fulfilled
    - Associate that object with a resource or request
    - requester blocks awaiting event on that object
    - upon completion, the event is “posted” or signaled
    - posting event to object unblocks the waiter

Synchronous Events

- What is a synchronous event?

    - A synchronous event is an event that returns immediately after we call them
    - When the call returns, the result is ready

Correct Mutual Exclusion

- What are some challenges in ensuring mutual exclusion?
    - We cannot eliminate all of the shared resources because it is important for performance and throughput, and we cannot prevent parallelism because it is fundamental in technology
    - What we can do instead is, identify the at risk resources, design those classes to enable protection, identify all the critical sections, and ensure each is correctly protected
- What are some approaches to achieve mutual exclusion
    - Avoid shared mutable resources
    - interrupt disables
    - spin locks: very limited applicability
    - Atomic instructions
    - mutexes
- What are the goals of achieving mutual exclusion?
    - Effectiveness/Correctness
        - ensures before or after atomicity
    - Fairness
        - no starvation (unbounded waits)
    - Progress
        - no client should wait for an available resource
        - susceptibility to convoy formation (waiting in the queue), deadlocks (no one can make progress because everyone is waiting for someone else)
    - Performance
        - delay, instructions, CPU load, bus load
        - In contended and uncontended scenarios

Interrupt Disables

- What are the advantages and disadvantages of interrupt disables in order to achieve mutual exclusion?
    - During an interrupt, a vector table of interrupts associated with the PS/PC is used to find a new PS/PC and store the old PC and PS is stored in the CPU stack. The new PC/PS is used to resume execution at a first level trap handler. The trap handler then saves all of the registers and states, garners information about the trap from the hardware, and selects a second level trap handler to call. From the second level trap handler, the ISR handles the interrupt just like a trap. Upon return, CPU state is restored and code resumes
    - Advantages
        - you can temporarily block some or all interrupts

        - can be done with a privileged instruction + prevent time slice end/timer-interrupts
        - side effect of loading new processor status
        - prevent re-entry of device driver code
    - Disadvantages
        - maty delay operations
        - a bug may leave the threads permanently disabled

Spin Locks

WEEK 5

Lecture 9

Extra questions

- Describe the process of a sloppy counter
    - Have a counter for each CPU core, as well as a single global counter
    - Each thread on each core updates its own counter and then after a certain amount of time, acquires the lock on the global counter and updates the value, then releases the lock
    - threads across many CPUs can update the counter without contention
- Where would be place the locks around in a concurrent linked list, queue, and hash table?
    - Linked list: put the locks around each node so that the code first grabs the next node's lock and then releases the current node's lock
    - Queues: Create a lock for the head and tails of the queue
    - Hash table: Create a lock for every bucket in the hash table
- How would we implement a memory allocation library using locks?
    - When a thread calls into the memory allocation code, it might have to wait in order for more memory to be free
    - Conversely, when memory is freed, the thread must signal that more memory is free and wake up the threads. Using the pthread_cond_broadcast() command, the thread wakes up all of the waiting threads and picks the first one in the queue

Producer/Consumer Problem (Bounded Buffer)

- What is the producer/consumer problem?
    - Consider one producer thread that puts data into a queue and two consumer threads that consume that data

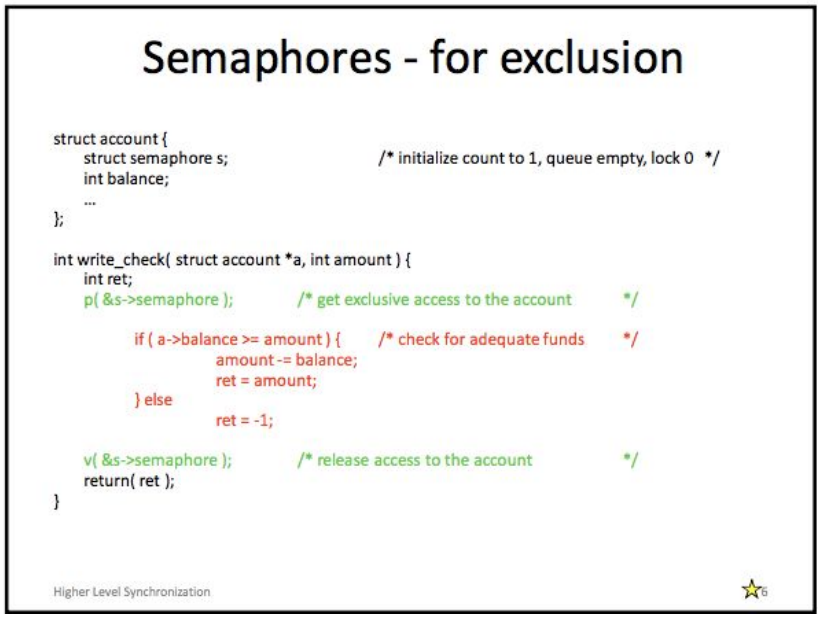
    - A producer thread can place data into the queue until it is full and then sleep, sending a signal to the consumer threads to wake them up. However, before C1 is called, C2 can sneak in and consume the lock so that C1 cannot use it and the state may change by the time the thread wakes up
    - Because bounded buffer is a shared resource, we need a shared buffer
    - The producer waits for the buffer to be empty while the consumer waits for the buffer to be full
- How do we solve this problem?
    - Use multiple condition variables to ensure that only consumers can wake the producers and only producers can wake the consumers
    - Allow for multiple buffer slots to improve efficiency
    - Reduce overhead through reducing context switches
    - Producer only sleeps if all the buffer slots are filled while the consumer only sleeps when all the buffer slots are empty
- What are condition variables used for?
    - Condition variables allow threads to sleep when some program state is not desired -> if a certain condition is not fulfilled
    - Enables threads to atomically release a lock and enter the sleeping state

Binary Semaphores

- What are binary semaphores?
    - Binary semaphores are used as locks because they hold two states (held and not held)
- What should the initial value in a binary semaphore be?
    - The initial value of a binary semaphore should be set to 1

Using Semaphores

- Condition Variables vs Semaphores
    - When a resource is available, semaphores guarantee that the caller has the resource available. However, with a condition variable (after a pthread_cond_wait()) command, we must check to see whether the resource is available.
    - This is because condition variables are merely an asynchronous completion mechanism and do not reserve resources
- Explain differences between CVs and semaphores (from midterm)
    - A semaphore maintains a counter, incrementing while V (waiting) and decrementing when successfully signaling (P). A condition variable is signaled when an event is available, but some other processes may come and do something with the resource before the awakened process can do so. Therefore, condition variables themselves do not ensure mutual exclusion
- When should we use a semaphore? When should we use a counter variable?
    - We should a semaphore when we want to ensure that each event was only claimed by a single process -> mutual exclusion

    - If the event has multiple waiters that need to take a look at the event in order to decide which thread should process the event, a pthread_cond_broadcast signal can be sent to awaken all of the threads
    - Basically, we want to use a semaphore when we want to ensure that only one waiter gets an available resource, while we use a condition variable when we want to decide which thread gets to process the resource (multiple waiters), we would like to broadcast by using a condition variable
- How would we use semaphores for exclusion
    - Set the initial value of the semaphore to one
    - Use P/wait operation to take the lock so that the first will succeed and then subsequent attempts will block
    - Use V/post operation to release the lock by incrementing the counter and restore semaphore count to non-negative. If any threads are waiting, unblock the first one waiting in line
    - 

- How would we use semaphores for notifications?
    - Initialize semaphore to zero
    - use p/wait operation to await completion
    - use v/post operation to signal completion
    - one signal per wait: no broadcasts

- ## Semaphores - completion events

```
struct semaphore pipe_semaphore = { 0, 0, 0 }; /* count = 0; pipe empty */
char buffer[BUFSIZE]; int read_ptr = 0, write_ptr = 0;

char pipe_read_char() {
        p (&pipe_semaphore );                   /* wait for input available   */
        c = buffer[read_ptr++];                 /* get next input character   */
        if (read_ptr >= BUFSIZE)                /* circular buffer wrap       */
                read_ptr -= BUFSIZE;
        return(c);
}

void pipe_write_string( char *buf, int count ) {
        while( count-- > 0 ) {
                buffer[write_ptr++] = *buf++;       /* store next character       */
                if (write_ptr >= BUFSIZE)     /* circular buffer wrap       */
                        write_ptr -= BUFSIZE;
                v( &pipe_semaphore );               /* signal char available      */
        }
}
```

Higher Level Synchronization 8

  - What are the limitations of using semaphores?
    - semaphores are a very spartan mechanism
      - they are simple and have few features + more designed for proofs rather than synchronization
    - They lack many practical synchronization features
      - very easy to develop deadlocks
      - cannot check the lock without blocking
      - do not support reader/writer shared access
      - no way to recover from a wedged V'er (releaser)
      - no way to deal with priority inheritance
      - Semaphores are not repairable
  - How is a semaphore implemented?
    - A simple semaphore can be implemented using locks and condition variables
    - ## Implementing Semaphores

```
void sem_wait(sem_t *s) {
    pthread_mutex_lock(&s->lock);
    while (s->value <= 0)
        pthread_cond_wait(&s->cond, &s->lock);
    s->value--;
    pthread_mutex_unlock(&s->lock);
}

void sem_post(sem_t *s) {
    pthread_mutex_lock(&s->lock);
    s->value++;
    pthread_cond_signal(&s->cond);
    pthread_mutex_unlock(&s->lock)
}
```

Higher Level Synchronization 9

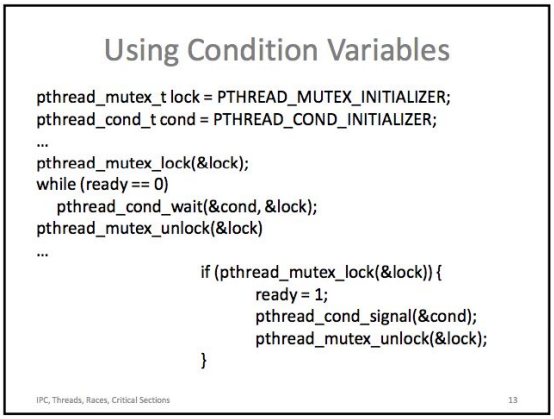
    - ## Implementing Semaphores in OS

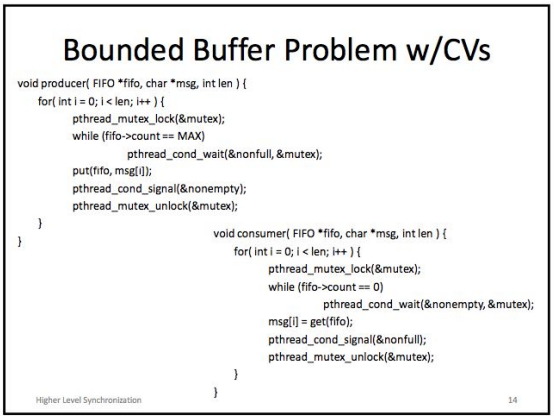
```
void sem_wait(sem_t *s ) {
    for (;;) {
        save = intr_enable( ALL_DISABLE );
        while( TestAndSet( &s->lock ) );
        if (s->value > 0) {
            s->value--;
            s->lock = 0;
            intr_enable( save );
            return;
        }
        add_to_queue( &s->queue, myproc );
        myproc->runstate |= PROC_BLOCKED;
        s->lock = 0;
        intr_enable( save );
        yield();
    }
}

void sem_post(struct sem_t *s) {
    struct proc_desc *p = 0;
    save = intr_enable( ALL_DISABLE );
    while ( TestAndSet( &s->lock ) );
    s->value++;
    if (p = get_from_queue( &s->queue )) {
        p->runstate &= ~PROC_BLOCKED;
    }
    s->lock = 0;
    intr_enable( save );
    if (p)
        reschedule( p );
}
```

Higher Level Synchronization 10

    - there must be a queue to store all of the waiting threads
    -

Using Condition Variables





- From previous midterm!
  - The application can probably call for the use of two semaphores
    - a work semaphore for back-end calls to await requests. The front end would V the wait queue (signal) whenever a new request was added to it and the backend threads would P (wait) the wait queue for work
    - A mutex semaphore to serialize access to the shared queue. P to lock the mutex and V to unlock it

```
server:
        P(mutex)
        append to work queue
        V(mutex)
        V(work queue)

worker:
        P(work queue)
        P(mutex)
        take item off queue
        V(mutex)
```

    -
  - Remember to solve a base bounds problem you would essentially need two semaphores, which invites deadlock but is necessary to ensure atomicity

Contention Reduction

- How do convoys form in contention?
  - If the mean service time exceeds the mean inter-request time, the queue line builds up and therefore becomes permanent
  - When processes are no longer able to immediately allocate the required resources and are always forced to block
- What is the difference between bottlenecks and convoys (differences and similarities)
  - Both involve reduced parallelism -> many threads waiting for a particular resource

    - In a bottleneck, the problem is the resource itself and that it is saturated -> resource throughput limits system throughput
    - In a convoy, the problem is the waiting queue itself
        - the resource may not be 100% utilized, precipitated by preemption in the critical section, and the line persists due to mandatory FIFO queuing
- How can we reduce contention? (previous midterm question)
    - Eliminate the critical section entirely
        - use atomic instructions on private resources (make everything shared)
    - Eliminate preemption during critical section
        - disabling interrupts -> not always a good option, and avoid sharing resources within the critical section
    - Reduce the time spent in the critical section
    - Reduce frequency of critical section entry
        - reduce exclusive use of the serialized resource
        - spread requests out over more resources
        - Reduce use of blocking operations in the critical section
        - Use sloppy counters to add to a global counter
    - Reduce mutual exclusion by implementing read/write locks
        - only writers require exclusive access and read/write locks allow many readers to share a resource

- What are the costs of contention?
    - Reduced parallelism from waiting for resources to become available, increases delay and reduces overall system throughput
    - Convoy occurs when there is a persistent queue of processes waiting to get a popular resource (remember FIFO scheduling)

Advisory Locking

- What is the difference between advisory and enforced locking?
    - Advisory locking follows a convention that “good guys” must follow, meaning users are expected to lock before calling methods, giving users more flexibility in locking + does not prevent reads, writes, and unlinks
    - However, in enforced locking, the locking is done within the object method and is guaranteed to happen, whether or not the user wants it + may sometimes be too conservative

File Level Locking + granularity

- Explain the differences between flock and lockf
    - flock is an advisory lock that is associated with an open file descriptor. It is released when the file descriptor is closed. lockf may or may not be enforced and represents somewhat of a hybrid between advisory and enforced locking meaning that it may or may not be enforced
- Explain the types of lock granularity

    - The term granularity comes from the example of sand: (finer grained sand or coarse grained sand)
    - A coarse-grained lock is a lock used for many objects
        - simple, more idiot proof
        - greater resource contention (threads/resources)
        - Could be worse if there are too many calls for locks, like hand-over-hand Linked list
    - fine grained - one lock per object (or sub-pool)
        - spreading activity over many locks reduces contention
        - dividing resources into pools shortens searches
        - a few operations may lock multiple objects/pools
    - Disadvantages of fine grained-granularity: Overhead of implementing more locks + error prone to manage multiple locks

Extra!

Identify the key criteria in which mutual exclusion mechanisms should be implemented

- Progress
- Fairness
- Correctness
- Performance

Evaluate Each mechanism against this criteria

- Semaphores
    - Progress: Do a good job of making progress
    - Fairness: It is fair
    - Correctness: Achieves mutual exclusion if it is used correctly by the user
    - Performance: In theory, good performance but slightly less than mutexes because of the queue. However, mutexes also have implementations with queues so therefore it doesn't really make a difference
- Mutexes
    - Progress: Not necessarily good, in general yes, but it could keep waking up the same thread over and over
    - Fairness: not always fair if we keep waking up the same thread over and over
    - Correctness: Achieves mutual exclusion
    - Performance: In theory, better than semaphores for above reason

- Spin Locks
    - Progress: deadlock danger occurs unless small critical section + low contention
    - Fairness: possibly unbounded waits -> not fair
    - Correctness: effective against parallelism (multi-process+multicore)
    - Performance: waiting is extremely expensive + wasted CPU cycles
- Disable Interrupts
    - Progress: Could potentially block processes + deadlock risk
    - Fairness: pretty good fairness because some are brief
    - Correctness: Not correct on multi core machines + MP parallelism and cannot be executed in user mode
    - Performance: Slow because of the overhead of having to make system calls

How do we reduce contention?

- We can reduce contention by reducing the amount of time spent in the critical section, and reducing the amount of calls we make to a critical section as well. We can also reduce contention through implementing read/write locks + break up the resources into smaller sub-pools of resources + fine grained locks
- Other methods
    - Eliminate the critical section entirely
    - Eliminate mutual exclusion through making everything shared
    - Eliminate preemption during a critical section
        - disable interrupts are used to disable interrupts during a critical section -> however expensive, not correct on MP parallelism + cannot be used in user mode so very expensive and slow

How is convoy formed?

- Convoy refers to the queue having unbounded wait times. A convoy can occur due to mandatory FIFO implementations + waiting for a resource to become available + preemption of the critical section
- Processes are no longer able to allocate the the required resources and forced to block. Therefore the service time exceeds the inter-request time and forms a convoy

What is replaced on a call to exec()

- Because a call to exec() runs an entirely new program, its stack and data segments + code segment is replaced while the PID, User ID, group ID and such are kept

How do we determine what should be placed in the OS and what shouldn't?

- Functionalities that are implemented in the OS should require privileged instructions that require trusted access of the resources, abstraction on how to use the hardware, security measures to keep the OS from breaking, scheduling mechanisms, virtualization mechanisms, interaction with the hard disk/hardware, etc. Functionalities outside of the OS should include things that do not have require privileged instructions to be

executed/access privileged resources, and use the abstractions provided by the OS to access hardware

Describe the process of memory compaction/defragmentation?

- 1) choose an area of disk where we want to allocate contiguous fhe space
- 2) for each file in that area copy it to some other place in order to free up space in the area we want to defragment
- 3) After clearing up the area, coalesce the segments in order to create a huge contiguous chunk
- 4) choose a set of files to be moved into the newly contiguous free space and move them in
- 5) Repeat this process until all of the files and free space is contiguous

How do paging and segmentation work together in memory allocation?

- Segments are used to find the requested page when a page fault occurs because it can use temporal and spatial locality to locate the page in disk
- operations on segments can use mmap(), find physical frame from virtual frame,etc.
- segment mapping implemented with page mapping

WEEK 6

Lecture 10

Explain the difference between livelock and deadlock

Necessary Conditions * WILL BE ON FINAL EXAM ACCORDING TO KAMPE**

- What are the necessary conditions for a deadlock?
    - No preemption: A thread that cannot be preempted during execution can deadlock. Resources cannot be forcibly removed from the thread/process that's holding them
        - R1 cannot be taken away from P1
    - Circular Wait/Dependency: There exists a circular wait such that each thread holds a resource that that are being requested by the next thread in the chain
        - P1 has R1, and needs R2
        - P2 has R2, and needs R1
    - Hold and Wait: Threads are holding resources allocated to them while waiting for additional resources
        - process already has R1 blocks to wait for R2
    - Mutual Exclusion: Threads claim exclusive control of resources that they require
        - P1 cannot use a resource until P2 releases it

Monitors

- What is the difference between a monitor and a semaphore?
    - A monitor is an object designed to be accessed by multiple threads. The member functions or methods of a monitor object will enforce mutual exclusion
    - A semaphore is a lower-level object that could be used to implement a monitor
        - It is essentially just a counter, where if it is positive, a thread tries to acquire the semaphore then it is allowed, and the counter is decremented
        - If multiple threads are waiting when a thread releases a semaphore, one of them gets it
    - A monitor is like a public toilet: only one person can enter at a time (**serially reusable**)
    - A semaphore is like a bike hire place. If you try and hire a bike and they have one free, then you can take it, otherwise you must wait
    - Essentially, a semaphore is an object that ensures mutual exclusion for a monitor
- Evaluate monitors against the four criteria of synchronization
    - **Correctness: Monitors do a good job of completely ensuring mutual exclusion**
    - **Fairness: Semaphore queue prevents starvation -> intuitively fair -> no unbounded waits**
    - **Progress: Inter-class dependencies could possibly cause deadlocks**
        - **we release the lock when we do the wait on the condition variable**
            - **what if you're in class A and you're calling a method in class B and vice versa?**
                - **circular dependency that arises from locking the entire class**
                - **This means that if anybody is using class A, then you must lock the entire class from others**
                - **The idea is to not worry about the critical sections, and simply just lock the entire object**
    - **Performance: Coarse grained locking is not scalable**
- Explain what a monitor is
    - A monitor is essentially a synchronization object for guaranteeing mutual exclusion
    - Could be implemented with a semaphore but necessarily needs a condition variable and a lock
    - Ensures that there is mutual exclusion (only one thread can access at a time) of an object/ class -> often used in Java as a method of simple synchronization
    - The mutex is associated with the class and not the object -> coarse grained locking to the extreme -> lock all the processes, not just one
- What are the disadvantages and advantages of monitors?
    - Advantages
        - guarantees mutual exclusion so is intuitively correct

        - Avoids starvation through using the semaphore queue/ condition variable
    - Disadvantages
        - Easy to deadlock, could cause problems when inter-class dependence
        - Coarse grained locking is not scalable meaning that the performance is not very good with higher numbers of threads

Spinning

- Explain the difference between spin locks and atomic update loops
    - both involve spinning on an atomic update
        - but they are not the same
    - a spin-lock
        - spins until the lock is released
        - which could have unbounded waiting times
    - an atomic update loop
        - spins until there is no more conflict during the update
        - impossible to be preempted while holding the lock
        - conflicting updates are actually very rare
- Evaluate Lock-Free Options against the four criteria mentioned in class
    - Effectiveness/Correctness
        - effective against all conflicting updates
        - **cannot be used for complex critical sections**
    - Progress
        - no possibility of deadlock or convoy -> cannot be preempted
    - Fairness
        - small possibility of brief spins
    - Performance
        - expensive instructions, but cheaper than syscalls

Java Synchronization Methods

- Evaluate Java Synchronized Methods against the criteria of locking
    - Correctness: correct if developer chooses the right methods -> the automatic protection method is somewhat lost because we aren't always ensured protection
    - Fairness: **has priority thread scheduling which could potentially starve other threads** (doesn't use something like a semaphore queue)
    - Progress: Is safe from single thread deadlocks -> can create multithreaded deadlocks
    - Performance: fine grained (per object) locking, can select which methods to synchronize
- Name the advantages and disadvantages of java synchronized methods
    - Advantages
        - Instead of the class having a mutex, you have an object that has an associated mutex

        - finer lock granularity, reduced deadlock risk (single threaded) (but you can still have circular dependencies)
        - Simplicity: Each object has an associated mutex that is acquired before the synchronized method, nested calls by the same thread do not reacquire, and lock is automatically released upon final return
    - Disadvantages
        - developer must identify serialized methods, we must know which methods cause trouble
        - Can still cause multithreaded deadlocking although it eliminates single threaded deadlock
            - if your thread already owns a mutex on the object, it just reacquires the same locks

Bonus Question

- **When do we want to use compare and swap vs test and set?**
    - we would want to use test and set when we are dealing with bits
        - “make it be true -> and tell me if its false”
        - very limited, can only be used for “has this happened yet”
    - Compare and swap looks at a bunch of data and decides what to do from the data
        - the correctness of the data depends on whether the data is still the same
        - more general than test and set -> you can use test and set to implement compare and swap

Deadlocks/Livelocks

- Explain what a deadlock is
    - A deadlock is when two or more processes cannot complete without all required resources and each holds a resource that the other needs
    - cannot make progress because each process is blocked and is waiting for another to complete
- What is the difference between a deadlock and a livelock
    - In a livelock, you are not actually blocked, but you are still not making progress and thus cannot complete
    - Example: If the professor decides not to put the final grades -> he is not blocked, just not going to do it
    - In a livelock, everything is still running, but no progress is being made because they are depending, not on allocating some resource, but on receiving something that they have not received yet
    - example: it is possible that two threads are attempting to receive a lock by repeating the operation over and over, but continuously fail to receive it
        - not the process’ fault, but the resource’s
    - Deadlocks occur when the process itself is blocked because it is waiting for a resource to be available but is not because of the circular dependency it has

        - i.e. thread A needs a resource that thread B has and thread B needs a resource that thread A has and therefore no progress is being made
- What are some types of deadlocks?
    - Commodity resource deadlocks
        - memory, queue space (the amount of something/which something)
    - General resource deadlocks
        - files, critical sections
    - Heterogenenous multi-resource deadlocks
        - e.g. P1 needs a file, P2 needs memory
    - Producer consumer deadlocks
        - e.g. P1 needs a file, P2 needs a message from P1
- Give an example of a commodity resource deadlock
    - Memory deadlock
        - if we are out of memory and we need to swap some processes out -> if we're out of memory to build the I/O request, then we can't ask for more memory

Dining Philosophers Problem

- Explain it
    - Five philosophers, five plates of pasta, five forks
    - they eat whenever they choose to
    - one requires two forks to eat the pasta but must take them one at a time
    - This means that one of the philosophers will starve
- Dining Philosophers was created to test synchronization mechanisms

Avoidance

- What are the techniques to avoid deadlocks
    - **We can build the scheduler so that we prevent deadlocks from happening**. For example, if we know that thread A needs Lock 1 and Lock 2, thread B also needs L1 and L2, thread C needs L2 and thread D needs none, we can make a scheduling algorithm that makes sure that thread A and thread B do not run concurrently in order to avoid deadlock
    - Resource Reservation
        - The resource manager only grants resources when they are available
        - Oversubscription is detected early
        - We want to decline to grant requests that would put the the system into a state of unsafe resource depletion
        - We want to reject incoming requests that we cannot handle -> graceful degradation
        - We ask processes to reserve their resources before they actually use them
            - therefore, **memory assignments do not happen until the processes start referencing the newly allocated pages**

Prevention (REVIEW FOR FINAL)

- Explain how to prevent deadlocks through avoiding mutual exclusion
    - Mutual exclusion means that P1 cannot use a resource until P2 is done using it
    - You cannot deadlock over a shareable resource
        - perhaps maintained within the atomic instruction
        - even read/write locking can help
            - readers can share, writers may be attacked in other ways
    - you can't deadlock if you have private resources
        - no deadlock is possible
- Explain how to prevent deadlocks through preventing hold and block
    - deadlock requires you to block holding resources
    - 1) allocate all resources in a single operation
        - you hold nothing while blocked
        - when you return, you have all or nothing
    - 2) disallow blocking while holding resources
        - you must release all held locks prior to blocking
        - reacquire them again after you return
    - 3) Non-blocking requests
        - a request that can't be satisfied immediately will fail
    - Basically, you are not allowed to hold anything while asking for anything else. If you want something, you must give up everything -> all or nothing/atomicity
    - Cannot block while holding for resources, although you are allowed to have multiple resources
- Explain how to prevent deadlocks through non-preemption
    - We can use leases rather than locks
        - process only has resource for a limited time
        - after which ownership is automatically lost
        - This is a method of preemption to prevent deadlocks from occuring
    - Forceful resource confiscation
    - termination … with extreme prejudice
    - deadlock prevents forwards progress -> reclaim resources from current holders
- Explain when preemption is feasible
    - We must know whether the access is mediated by the operating system itself
        - e.g. all object access is via system calls
        - we can revoke access, and return errors
    - We must know whether we can force a graceful release of a resource
        - this can be done through a claw-back call to the current owner
    - If confiscation can leave the resource corrupted, we can un-map a segment or kill a process
        - ex: shared segments with complex computations can be interrupted with preemption in the middle of an update
        - therefore, we must have all or none updates

- Explain how we can attack circular dependencies and prevent deadlocks?
    - There are several ways we can eliminate circular dependencies
    - total resource ordering
        - all requesters allocate resources in the same order
        - for example, we can first allocate R1 and then R2 afterwards
        - someone else may have R2 but he doesn't need R1
    - assume we know how to order resources
        - order by ID (e.g. I-node, IP address, mem address)
        - order by resource type (e.g. groups before members)
        - order by relationship (e.g. parents before children)
    - **may require a lock dance**
        - release R2, allocate R1, reacquire R2
    - We can also have partial ordering
        - This is implemented in some parts of the linux system
            - for example, we can separate locks into groups, and then create an ordering system within that group
            - Or we can order by things like memory addresses/lock addresses

Reservations

- When do we use reservations? What is the basic approach
    - Reservation is used as a common technique to avoid deadlock
    - We want to decline requests that would put the system into an unsafe, resource depleted state to prevent deadlock from occuring
- Give an example of reservation
    - malloc() is an example of reservation
        - it reserves the memory until the process gets the resource, preventing oversubscriptions from happening
        - you can refuse to allocate the resource when you don't think you can handle it
    - Swap manager
        - Situation: invoked to swap out processes to free up memory, may need to allocate memory to build I/O request, but no memory is available to do so, so we are unable to swap out processes
        - Solution: pre-allocate and hoard a few request buffers
            - keep reusing the same ones over and over again
            - little bit of hoarded memory is a small price to pay for avoiding deadlock
- How do reservations avoid deadlocks?
    - The resource manager only tracks outstanding reservations and **only grants reservations if resources are available ->** no contention/waiting for resource
    - Oversubscriptions are detected early -> before processes ever get the resource

    - Resource reservations deal with a common critical resource exhaustion deadlock **where a process has faulted for a new page but there are no free pages in memory and there are no free pages on the swap device**
    - Through detecting oversubscriptions before the process is granted the resource, reservations can help avoid deadlocks through rejecting incoming requests if we feel as if it is unsafe or will leave the resource in a depleted state
- When do allocation failures occur?
    - Allocation failures only occur at reservation time (hopefully before a new computation is begun) and once a failure occurs, it is completely up to the client to handle it -> makes system behavior predictable (if reservations cannot be guaranteed, they will not be allocated)

Detection and Recovery

- What are some techniques used for deadlock detection (hang)?
    - Health monitoring
        - Have an internal monitoring agent watch message traffic or a transaction log to determine whether or not work is continuing
        - Health monitoring can also detect many other things such as infinite loops/waits, crashes, and livelocks
    - Looking for obvious failures
        - process exits or core dumps
    - Passive observation to detect hangs
        - check whether process is consuming CPU time or is blocked and check if a process is doing network I/O and/or disk I/O
- Give an example of a common health monitoring system
    - first line of defense in an internal monitoring agent that closely watches key applications to detect failures and hangs
    - if the internal monitoring agent is responsible for sending health status reports to a central monitoring agent, a failure of the internal monitoring system can be detected
    - An external test service that periodically generates test transactions provides an independent assessment that might include external factors that would not be tested by internal and central monitoring services. (i.e. if the internal monitoring agent failed)
- What is a”hang”
    - not a true deadlock but causes program to not make any progress
        - wouldn’t be found by a true deadlock detection system -> although behavior similar to a deadlock
    - livelock
        - processing is running, but won’t free a resource until it receives a message
        - process that will send the message is blocked for the same resource

        - processes are making no progress because they are dependent on some resource that has not been allocated yet
        - **special case of resource starvation**
            - the state of the process may be changing but no progress is ever being made
    - Sleeping Beauty, waiting for “Prince Charming”
        - a process is blocked, awaiting some completion
        - but, for some reason, it will never happen
- Explain priority inversion
    - example
        - preempted low priority process P1 has mutex M1
        - high priority process P2 blocks for mutex M1
        - process P2 is effectively reduced to priority of P1
    - Basically, when a process’ priority is inverted because of a resource
    - Consequences of priority inversion
        - depends on what the high priority process does -> may go on unnoticed, might be a minor performance issue, might result in disaster
- When does formal detection make sense?
    - formal detection makes sense in situations such as priority inversion because it can result in serious problems in real-time systems and are so very difficult to find
- How do we solve priority inversion?
    - priority inheritance
        - Identify resource that is blocking P1
        - identify current owner of that resource (P3)
        - temporarily raise P3 priority to that of P1 -> until P3 releases the mutex
        - P3 now preempts P2, runs to completion
        - P3 releases lock, and loses inherited priority
        - P1 preempts P2 and runs
        - P2 resumes execution
- What are some methods of recovering from the above problems?
    - Automated Recovery
        - Kill and restart all of the affected software
        - We must design services to automatically fail-over
        - components can warm-restart, fall back to last checkpoint, or cold restart
- What are the different options of restarting once a system has hung or failed?
    - highly available services must be designed for restart, recovery, and failover in the case where a system has hung or failed
    - Warm-start
        - restore the last saved state and resume service
    - Cold-start
        - ignore any saved state and restart new operations from scratch -> this is good when the previous saved state may be corrupted
    - Reset and Reboot

        - reboot the entire system and cold-start the applications
    - Progressively escalating restarts
        - restart only a single process and expect it to sync up with the other processes
        - maintain a list of all of the processes involved in the delivery of a service, and restart all processes in that group
        - restart all of the software on a single node
        - restart a group on nodes, or the entire system
    - Non-disruptive rolling upgrades
        - if a system is capable of operating without some of its node, it is possible to achieve non-disruptive rolling software upgrades
        - we can take nodes down one at a time, upgrade each to a new software release and reintegrate them into the system
            - new software must be upwards compatible with the old software
            - there must be a fallback feature to revert back to the old form if the new software is not working
    - Prophylactic reboots
        - automatically restart every system at a regular interval

Lecture 11

Goals and challenges of performance analysis

- What are the goals of performance analysis
    - we want to quantify the system performance for competitive positioning and to assess the efficacy of previous work/identify future improvements that can be made
    - Understand the system performance, as in what factors are limiting our current performance and what choices make us subject to these limitations
    - We want to be able to predict system performance so we can see how proposed changes affect performance
- What are the challenges of performance analysis
    - Components operate in a complex system, so there are many steps/components in every process + ongoing competition for processes
    - When to run the performance analysis/choosing a definition of ‘good’ and ‘bad’
        - there is a clear lack of requirements
    - difficulty to make clear/simple assertions + systems may be too large to replicate in a lab

Performance Principles

- What are the main performance principles?
    - **The Pareto Principle**
        - 80% of cycles are spent in 20% of the code
    - **“Data trumps opinion”**

        - intuition often turns out to be wrong
        - we can't optimize what we don't measure
    - **"Rust never sleep"**
        - continuous measurement and comparison
        - if we aren't getting faster, we're getting slower
    - **Performance is merely about design**
        - code optimization is only occasionally useful

Typical Sources of Performance Problems

- What are some sources of performance problems?
    - overload of a resource such as memory bandwidth or CPU cycle, solution in built in software doesn't scale
    - Non-scalable solutions
        - cost per operation becomes prohibitive at scale
        - worse-than-linear overheads and algorithms
        - queuing delays associated w/high utilization
    - bottlenecks
        - one component that limits the entire system throughput
    - accumulated costs
        - layers of calls, data copies, message exchange
        - redundant or unnecessary work
- How do we figure out why performance is bad?
    - You can often get clues without running any new experiments. The OS will tell you, on request, how much memory is being used, CPU utilization, and many other statistics
        - ex: if there's plenty of free memory in the system but is still running poorly -> waste of time to run experiments
    - We must run tests on certain areas to discover what is bad (e.g. scalability, memory, etc.)

Common Measurement Errors

- What are some common measurement errors that occur when taking performance measurements?
    - measuring time but not utilization (measuring latency without considering utilization)
        - everything is fast on a lightly loaded system -> test different loads
    - capturing averages rather than distributions
        - outliers are usually interesting -> can skew data as well
    - ignoring start-up, accumulation, and cache effects (special cases)
        - not measuring what we thought
    - ignoring instrumentation artifact
        - it may greatly distort both times and loads
- How do cache and accumulation start-up effects change performance measurements?

    - Cached results may accelerate some runs
        - random requests that are unlikely to be in cache
        - overwhelm cache w/new data between tests
        - disable or bypass cache entirely
    - startup costs distort total cost of computation
        - do all forks/opens prior to starting actual test
        - long test runs to amortize startup effects down
        - measure and subtract start-up costs
    - system performance may degrade with age
        - reestablish base condition for each test
- What is execution profiling used for?
    - execution profiling is an automated measurement tool that extracts data about the CPU cycles, number of calls, times per call, percentage of time, etc. to identify bottlenecks in the system
    - uses statistical execution sampling
        - timer interrupts execution at regular intervals
        - increments a counter in table based on PC value
        - may have configurable time/space granularity

Performance metrics

- What are different types of performance metrics that we would like to measure
    - competitive performance metrics
        - used to compare with competing products
        - standard transactions/second
    - engineering performance metrics
        - used to spec components
        - used to analyze performance problems
- What is a metric?
    - **A standard unit for measurement or evaluation of something**
        - metric must be quantifiable (time/rate, size/capacity, reliability, etc)
        - metric must be measurable (or computable)
        - must be an interesting/valuable quality/characteristic
        - metric must be well-correlated with that quality
- What are some common types of system metrics?
    - **duration/response time**
        - mean latency for a benchmark request
    - **processing rate**
        - how many web request handled per second
    - **resource consumption**
        - how much disk is currently used?
    - **reliability**
        - how many messages delivered without error?
        - mean time between failures

- How should we choose our metrics?
    - We should pick metrics based on
        - **completeness**: do these metrics span “goodness”
        - **redundancy**: each metric provides new info?
        - **variability**: how consistent is it likely to be?
        - **feasibility**: can i accurately measure that metric?
        - **diagnostic/predictive value**: yields valuable insight

Load Generation

- What is a load generator?
    - A load generator is a system that can generate test traffic, corresponding to a specified profile, at a calibrated rate
    - a good load generator should be tunable in terms of both the overall request rate and the mix of operations that is generated
        - may also generate random requests according to a distribution
- What is a factor in performance testing?
    - “controlled variations, to enable comparison”
        - things you intentionally alter in performance experiments to determine which of several alternatives to use are called factors
- What is a level in performance testing?
    - A range of values/choices for each factor that can be categorized in terms of numerical ranges, booleans, categorical levels (Ext3 vs XFS), etc.)
- Give examples of how a load generator can be used
    - if the system to be tested is a network server, the load generator will pretend to be multiple clients, sending requests
    - if the system to be tested is an application server, the load generator will create multiple test tasks to be run
    - if the system to be tested is an I/O device, the load generator will generate I/O requests
- How does accelerated aging improve testing measurements?
    - accelerated aging is when the load generator cranks up the rates at which requests are issued in order to simulate aging
    - Because systems can do poorly as time goes on, load generators can be used to create traffic to simulate normal traffic for long periods of time
- Compare and contrast/ give the main strengths and weaknesses of simulated work loads, captured sessions, testing under live loads, and standard benchmarks
    - Simulated Workloads (Artificial load generation)
        - Advantages
            - controllable operation rates, mix of operations
            - scalable to produce arbitrarily large loads
            - can collect excellent performance data
            - on-demand generation of specified load
        - Disadvantages

                - random traffic is not a usage scenario
                - wrong parameter choices can yield unrealistic loads
        - Captured Sessions (captured operations from real systems)
            - Advantages
                - represent real usage scenarios
                - can be analyzed and replayed over and over
            - Disadvantages
                - each represents only one usage scenario
                - multiple instances not equivalent to more users
                - danger of optimizing the wrong things
                - limited ability to exercise little-used features
                - they are kept around forever, become stale
        - Testing under Live Loads (instrumented systems serving as clients)
            - Advantages
                - represent a combination of real usage scenarios
                - measured against realistic background loads
                - enables collection of data on real usage
            - Disadvantages
                - demands good performance and reliability
                - potentially limited testing opportunities
                - load cannot be repeated/scaled on demand
        - Standard Benchmarks (Carefully crafted/reviewed simulators)
            - Advantages
                - heavily reviewed by developers and customers
                - believed to be representative of real usage
                - standardized and widely available
                - well-maintained
                - comparison of competing products
                - guide optimizations
            - Disadvantages
                - inertia, used where they are not applicable

Traces/Logs + internal instrumentation

- How can traces/logs be used to analyze data?
    - we can create a log buffer and routine (time stamped event log) and have routines store time and event in a buffer
    - extract buffer, archive, mine the data
        - time required for a particular operation
        - frequency of operations
        - combinations of operations
        - also useful for post-mortem analysis

End to End Measurement

- Notes
    - client-side throughput/latency measurements
        - elapsed time for X operations of type Y
        - instrumented clients to collect detailed timings
- What are the advantages and disadvantages of end-to-end testing?
    - Advantages
        - easy tests to run, easy data to analyze
        - results reflect client experienced performance -> realism
    - Disadvantages
        - don't provide information on why it took that long
        - no information about resources consumed as opposed to execution profiling

Analysis Techniques

- Can you characterize latency and throughput?
- Can you account for all the end-to-end time -> processing, transmission, queuing delays
- Can you explain how these vary with load?
- Are there any significant unexplained results?
- Can you predict the performance of a system?
- Ex: Why throughput falls off
    - dispatching processes is not free
        - it takes time to dispatch a process (overhead)
        - more dispatches means more overhead (lost time)
        - less time (per second) is available to run processes
    - how to minimize the performance gap
        - reduce the overhead per dispatch
            - minimize the number of dispatches
- Ex: why response time grows w/o limit
    - response time is a function of server and load
        - how long it takes to complete one request
        - how long the waiting line is
    - length of the line is function of server and load
        - how long it takes to complete one request
        - the average inter-request arrival interval
    - if requests arrive faster than they are serviced
        - the length of the waiting list grows
        - and the response time grows with it

WEEK 7
Lecture 12

I/O Bus

- Explain what an I/O bus is
    - A bus is simply a common set of wires that connect all the computers and chips together
    - Connects I/O devices to the CPU -> can be used to send/transmit information from external devices into the CPU
    - Memory type busses
        - initially back-plane memory to CPU interconnects
            - consists of a few bus masters and many bus slaves
            - arbitrated multi-cycle bus transactions
                - request, grant, address, respond, transfer, ack
                - operations: read, write, read/modify/write, interrupt
- Terms: Define bus master, slave, and arbitration
    - bus master
        - any device or CPU that can request the bus
        - One can also speak of the “current bus master”
    - bus slave
        - a device that can only respond to bus requests
    - bus arbitration
        - process of deciding who to grant the bus
            - may be based on time, geographic location/priority
            - may also clock/choreograph steps of bus cycles
            - bus arbitrator may be part of CPU or separate

Device Controllers

- What are device controllers?
    - Device controllers connect a device to a bus (e.g. an I/O device that interacts between the computer and other media such as a disk, network, keyboard, etc.)
    - Communicate control operations to the device
    - relay status information back to the bus
    - manage DMA transfers to the device
    - generate interrupts for the device -> usually device and bus specific instructions
- How do device controllers interact with the bus
    - through device controller registers
        - Simplified interface comprised of three registers
            - **status:** current status of the device
            - **command:** tells the device to perform a particular task
            - **data:** pass data to the device, or get data from the service
        - registers in controller can be addressed from the bus
        - writing into registers controls device or sends data
        - reading from registers obtains data/status -> data register
- How do we reduce the overhead created by repeatedly checking whether data is available in the registers?

    - Because a simplified device controller keeps polling while waiting for data to become available, it can waste CPU cycles
        - **Interrupts**
            - OS issues a request, put the calling process to sleep, and then context switch to another task
            - When the device is finally finished with the operation, it will raise a hardware interrupt which causes the CPU to just to a predetermined interrupt handler
            - interrupts thus allow for overlap to computation and I/O
            - With interrupts, the OS can run another task instead of spinning until an I/O request is completed
        - **Coalescing**
            - A device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU
            - multiple requests many soon complete and thus multiple requests can be coalesced into a single interrupt delivery and reduce the overhead of interrupt processing
            - However, waiting too long would increase the latency of the process
- What are the disadvantages of interrupts?
    - When a device is able to perform its tasks very quickly, interrupts could possibly slow down the program
    - For fast tasks, we should use polling while for slower tasks we can use interrupts so that the CPU cycles won't be wasted
    - for networks, a huge stream of incoming packets each generating an interrupt can cause the OS to livelock

.

Disk Geometry

- Describe the basic parts and terminology of the disk
    - Spindle
        - a mount assembly of circular platters
    - head assembly
        - read/write head per surface, all moving in unison
    - track
        - ring of data readable by one head in one position
    - cylinder
        - corresponding tracks on all platters
    - sector
        - logical records written within tracks
    - disk address = cylinder/head/sector
    - platter
        - a circular hard surface on which data is stored persistently by inducing magnetic charges to it

Importance of disks

- Why have disks dominated file systems for many years?
    - fast swap, file system, database access
    - organize seek overhead
        - organize file systems into cylinder clusters
        - write-back caches and deep request queues
    - minimize rotational latency delays
        - maximum transfer sizes
        - buffer data for full-track reads and writes
    - we accepted poor latency in return for IOPS

Disk Performance

- What are some ways to optimize disk performance?
    - **don't start I/O until disk is on-cylinder/near sector**
        - I/O ties up the controller, locking out other operations
        - other drives seek while one is doing I/O
    - **minimize head motion**
        - do all possible reads in current cylinder before moving
        - make minimum number of trips in small increments
    - **encourage efficient data requests**
        - have lots of requests to choose from
        - encourage cylinder locality
        - encourage largest possible block size
- What are the goals in term of performance for disks
    - we want to maximize throughput, have low latency, good scalability, reliability, and availability
    - **What are the advantages and disadvantages of SSTF (Shortest Seek Time first)**
    - advantages
        - intuitively fair because the shortest seek time first will guarantee that all reads will be done
    - disadvantages
        - drive geometry is not available to the OS as it is simply seen as an array of blocks
            - can use the nearest block first algorithm instead
        - **Starvation:** just like the SJF approach with process scheduling, if there is a steady stream of requests on the inner track, where the head is positioned, requests to any other tracks would be ignored in a pure SSTF approach
- What are the advantages and disadvantages of the elevator approaches (FSCAN, SCAN, C-SCAN)
    - uses sweeps, which are simply a single pass across the disk

    - advantages
        - solves the starvation problem in SSTF through storing requests in a queue
        - F-SCAN (freeze) freezes the queue to be serviced when doing a sweep so that requests that come in during the sweep will be placed in the queue to be serviced later. -> avoids starvation through placing incoming requests in the queue to be serviced later
        - C-SCAN (circular): instead of sweeping back and forth across the disk, the algorithm sweeps from outer to inner, and then resets the outer track to begin again
    - disadvantages
        - don't take into account rotation and seek times -> + additional overhead from these times
- What are the advantages and disadvantages of SPTF (shortest positioning time first)
    - advantages
        - uses the relative time of seeking as compared to rotation to solve the problems addressed in elevator algorithms
        - if the seek time is much higher than rotational delay, then SSTF is fine
        - if seek time is faster than rotation rotational delay, then it should most likely service the outer track sectors first
    - disadvantages
        - unlike elevator algorithms, queues are not used to store incoming requests so incoming requests could possibly decrease performance
- Where is the disk scheduling performed on modern systems?
    - disks have sophisticated internal schedulers themselves because they have information about the seek and rotational delays
    - The OS scheduler usually picks what it thinks the best few requests are and issues them all to the disk
        - disk uses SPTF to process the requests in the best possible order
    - Sometimes, the disk can take advantage of I/O merging to merge requests together to one request to reduce overhead
- How long should the system wait before issuing an I/O to disk?
    - There are two techniques in determining this
        - **Work-conserving:** immediately issue the requests to the drive
        - **anticipatory disk scheduling:** waits a bit before issuing the request to drive
            - by waiting, a new and better request may arrive and it may be possible to coalesce/merge the requests before sending them to disk

Random vs Sequential I/O

- Explain the difference between random and sequential I/O

    - Random I/O is when we have read requests to random locations on the disk vs sequential I/O where it is on consecutive sectors on the disk
- I/O time: How do you calculate the time of the I/O and rate of the I/O
    - T(I/O) = T(seek) + T(rotation) + T(transfer)
        - the rate of I/O is more easily used for comparison between drives
    - R(I/O) = size(transfer)/T(I/O)
- Explain what striping is and evaluate RAID 0 (striping) against random and sequential I/O
    - In a RAID system, striping is when you spread the blocks of the array across the disk in a round-robin fashion so that you can get the most parallelism from the array when requests are made for contiguous chunks of the array
    - Achieves maximum throughput on both sequential and random I/Os as RAID 0 presents upper bounds on both capacity and performance, but data loss is guaranteed when a disk fails
    - Single request latency: latency of a single I/O request to a RAID reveals how much parallelism can exist during a single logical I/O operation
    - Steady-state throughput: total bandwidth of many concurrent requests
- Evaluate RAID 1 (Mirroring) against random and sequential I/O
    - In a mirrored system, we simply keep a copy of each block in the system -> copy is placed on different disk so that we can tolerate disk failures
    - In terms of the three criteria of performance, capacity, and reliability
        - Mirroring provides extremely good reliability in terms of disk failure because we can simply retrieve another copy of the same data from another disk
        - From a capacity standpoint, mirroring is expensive because we can only use ½ of the actual total capacity of the disk
        - Performance
            - Latency: for reading, it is equivalent to only having one disk because we can choose one of the two disks to read
                - However, for writing, even though the writes occur in parallel (so it is roughly the same as a single write), because the logical write must wait for both physical writes to complete, it suffers from the worst case seek and rotational delays
            - Throughput
                - Sequential: Since each logical write must result in two physical writes, the max bandwidth will be half of the peak bandwidth. -> also only half the peak bandwidth for reads as well
                - Random: best case for a mirrored RAID is random read because we can distribute the requests across the disk and achieve the full bandwidth -> however, random writes only achieve half of the peak bandwidth as expected

- Evaluate RAID 4 against random and sequential I/O
    - Uses parity to overcome the space penalty paid by mirrored systems
    - in terms of capacity, RAID-4 uses one disk for parity so the capacity is (N-1)*B
    - Reliability: RAID-4 tolerates at most one disk failure, otherwise it is simply impossible to reconstruct
    - Performance
        - Throughput
            - **Sequential Read**: effective bandwidth of (n-1)*B because it can use all disks except for the parity
            - **Sequential write**: done through an optimization called **full-striped write** where the parity is calculated in parallel with the blocks that are written to -> effective bandwidth of (n-1)*B
            - **Random reads**: A set of random reads will be spread across the data disks but not the parity -> (N-1)*B
            - **Random writes:** We can use additive or subtractive parity to update both the overwritten block and the parity block
                - throughput is awful for small writes and does not improve as you add disks to the system
        - Latency
            - Latency of a single read is just mapped to a single disk so it is equivalent to that of a single disk request
            - Latency of a single write request requires two reads and two writes -> reads can happen in parallel, as can the writes, but the latency just ends up being twice that of a single request
- **Bonus question: Why do random writes for RAID-4 use two reads and writes?**
    - When using additive or subtractive parity, we must read in all of the other data blocks in the stripe in parallel and XOR those with the new block (additive).
    - When using subtractive parity, we read in the old value of the block we want to read in and the old value of the parity. Then, we have to compare the old data and the new data, and update the parity bit accordingly, meaning that two logical I/O's/four physical I/Os are necessary -> 2 reads, 2 writes
- Evaluate RAID 5 against random and sequential disk I/O
    - RAID-5 conquers the issue presented by RAID-4 in which small, random writes provide terrible performance as the bandwidth is awful
    - Effective capacity and failure tolerance is essentially the same as RAID-4
    - Performance
        - Sequential read and write performance is the same as RAID-4
        - Random Read: random read performance is slightly better because we can utilize all disks instead of leaving out the parity disk (parity block is included within each disk)
        - Random Write: improves noticeably over RAID-4 because it allows for parallelism across requests
- When would you want to use RAID-4 over RAID-5

    - RAID-4 provides simplicity in terms of the parity disk, but because the performance for small random writes are bad, it should only be used when you know that large random write requests will not occur

Transfer Size

- Why are bigger transfers of data better?
    - **disks have high seek/rotation overheads** so larger transfers help amortize down the cost/byte
    - all transfers have per-operation overhead
        - instructions to set up operations
        - device time to start new operation
        - time and cycles to service completion interrupts
    - larger transfers have lower overhead/byte
        - not limited to s/w implementations
    - Basically, larger transfers of data optimize bandwidth while also helping amortize the cost/byte since seek/rotation overheads can be costly.
        - By batching transfers, we can minimize the amount of seek/rotation overhead and take advantage of disk/cylinder locality

Polled I/O (Direct)

- Describe the advantages of polled I/O
    - it is very easy to implement in terms of both hardware and software -> however, polling for data to be available/request to be sent can be inefficient in terms of wasting CPU cycles
- Evaluate the performance of direct I/O
    - If the request can be performed very quickly, poll I/O can be fairly efficient and provide good performance
    - If the data transfers are CPU intensive, it could waste CPU cycles while waiting completion because busy-waiting ties up CPU until I/O is completed
    - devices are idle while the I/O is being handled
    - If devices are idle, the throughput drops -> longer system queues -> slower response times

DMA (Direct Memory Access)

- What is a DMA?
    - A DMA engine is essentially a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention
    - To transfer data to the device, the OS would program the DMA engine by telling it where the data lives in memory, how much data to copy, and which device to send it to
    - Without DMA, the CPU is unable to run other tasks while handling I/O requests such as accessing data from the disk.

    - The bus facilitates data flow in all directions between
        - CPU, memory, and device controllers
    - CPU can be the bus master -> requests the bus
        - initiating data transfers w/memory, device controllers
    - device controller can also master the bus
        - CPU instructs controller what transfer is desired
        - controller does transfer w/o CPU assistance
        - controller generates interrupt at end of transfer
- How do we keep key devices busy?
    - allow multiple requests pending at a time -> queue them, just like processes in the ready queue and requestors block to await eventual completion
    - use DMA to perform the actual data transfers
        - data transferred, with no delay, at device speed + minimal overhead imposed on the CPU
    - when the currently active request completes -> send completion interrupt
        - device controller generates the completion interrupt
        - interrupt handler posts completion to requestor
        - **interrupt handler selects and initiates next transfer**

Completion Interrupts

- How does the CPU know when the I/O request is done when using DMA?
    - The busses have ability to send interrupts to the CPU and the device controller can signal when the requests are done/ready
    - when the device finishes, controller puts interrupt on the bus
    - The DMA uses a completion interrupt to notify the CPU that a buffer transfer is complete
- how does the OS send data to specific device registers?
    - Uses explicit I/O instructions provided by third party providers such as IBM
    - in x86, the in and out instructions can be used to communicate with devices -> send data by using the in command

Memory-mapped I/O

- What is memory-mapped I/O
    - the hardware makes the device registers available as if they were memory locations
        - to access a particular register, the OS makes a load or a store to the address
        - the hardware then routes the load/store to the device instead of main memory
    - Implement as a bit-mapped display adaptor
        - ex: 1m pixel display controller, on the CPU memory bus
        - each word of memory corresponds to one pixel
        - application uses ordinary stores to update display

    - has low overhead per update, no interrupts to service + easy to program
    - the memory and registers of the I/O devices are mapped to/associated with specific address values
- What are the tradeoffs (in terms of advantages and disadvantages of memory mapped I/O vs DMA)
    - DMA
        - advantages
            - performs large transfers of contiguous sections efficiently and has better utilization of both the CPU and the devices
        - disadvantages
            - however, there is a considerable per transfer overhead because setting up the operation and then processing the completion interrupt can be expensive
    - Memory-mapped I/O
        - advantages
            - has no per-op overhead because it can simply just look up the data at the memory location
        - disadvantages
            - every byte is transferred by a CPU instruction -> no waiting because device accepts data at memory speed
    - Overall, DMA should be used for large, contiguous transfers, (occasional), while memory-mapped I/O should be used for more frequent smaller transfers -> memory mapped devices more difficult to share

Write-back and write-through caches

- What is the difference between a write-back and write-through caches
    - Write-through cache
        - “writes-through” to memory so that the main-memory always has an up-to-date copy of the line -> so when a read is done, main memory can always reply with the newest requested data
    - Write-back cache
        - Sometimes the updated data is in the processor cache, and sometimes it is in main memory. Buffers write requests by keeping a queue and then issuing the write into memory.
        - The write to memory is postponed until the modified content is about to be replaced by another cache block
- Advantages and disadvantages of both write back and write through cache
    - Write back caches
        - advantages
            - low latency and high throughput for write-intensive applications because we buffer the write requests before issuing one big request to write to main memory
        - disadvantages

        - there is data availability risk because if a user requests data from main memory, the most updated copy may not be available even though it is sitting in the cache
    - Write through caches
        - advantages
            - You will always have the newest copy in main memory because the main memory will always have an up-to-date copy of the line
            - it simplifies the design of the computer system -> doesn't need to buffer
        - disadvantages
            - expensive to send requests to main memory from cache -> when there are lots of requests can slow down performance drastically
            - Writing data will experience latency as you have to write to two places every time

Chain Scheduling

- tbh don't really know so i guess i'll write a piece of code
- I guess we can write a chain scheduling I/O program
    -

```
xx_read/write() {
    //allocate a new request descriptor  int request = new request or something
    //fill in type, address, count, loation  request.fillInfo(addr, count, loc)
    //insert request into service queue queue.push(request)
    if(!device_idle){
        disable_device_interrupt()
        xx_start()
        enable_device_interrupt()
    }
    //await completion of request
    //extract completion info for caller
}

xx_start() {
    get next request from queue
    get address count, disk address (location)
    load request parameters into controller
    start the DMA operation
    mark device as busy -> not idle
}

xx_intr() {
    extract completion info from from controller
    update completion info in current req
    wakeup current request
    if (more requests in queue)
        xx_start()
    else
        mark device idle
}
```

Buffered I/O

- What are the advantages of buffered I/O
    - fewer/larger transfers are more efficient -> think about write-back caching or DMA
    - natural record sizes tend to be relatively small
    - by maintaining a cache of recently used disk blocks, we can accumulate small writes and flush out the blocks as they fill

    - read whole blocks, deliver data as requested
    - Enables read-ahead -> **OS reads/caches blocks not yet requested**
- How do we achieve deep request queue (have many I/O operations queued is good)
    - if we have many processes making requests
    - individual processes making parallel requests
    - read-ahead for expected data requests
    - write-back cache flushing

Scatter/Gather I/O

- Explain how scatter/gather I/O
    - Since many controllers support DMA transfers and 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 to 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
    - gather writes from paged memory and scatter reads into paged memory
    - Summarized: method of I/O by which single procedure call sequentially reads data from multiple buffers and writes to it a single data stream, or reads data from a data stream and writes it to multiple buffers

Intelligent I/O Devices

- Describe a basic smart device controller
    - Essentially, we want the best of both worlds from memory mapped I/O and DMA
    - We can have smarter controllers to improve on basic DMA
    - they can queue multiple I/O requests
        - when one finishes, automatically start next one
        - reduce completion/start-up delays
        - eliminate need for CPU to service interrupts
    - they can relieve CPU of other I/O responsibilities
        - request scheduling to improve performance
        - they can do automatic error handling and retries
    - abstract away details of underlying

Read/Write Striping

- What are the advantages and disadvantages of read/write striping
    - read/write striping, as explained in RAID 0, looks to optimize bandwidth in performance and capacity by performing requests in parallel
    - advantages
        - increase throughput + fewer operations per second per target
        - can be used for many types of devices (RAID, disk or server, NIC)

    - disadvantages/potential issues
        - does not protect against data corruption/when a disk fails
        - more/shorter requests may be less efficient
        - striping agent throughput is the bottleneck -> source can generate many parallel requests

Write mirroring

- What are the advantages and disadvantages of write mirroring?
    - advantages
        - write mirroring is spread across multiple targets -> increases aggregate throughput, reduced one target
        - redundancy in case a target fails
        - used for all kinds of persistent storage -> disks, NAS, distributed key/value stores
    - disadvantages
        - added write traffic on source
        - 2x-3x storage requirements on targets
        - deciding which copy is correct -> can cause conflict

Parity/Erasure Coding

- What are the advantages and disadvantages of parity/erasure coding -> think of RAID ⅘
    - advantages
        - parity can be used to recover lost bits and check for errors / inconsistencies
        - Cyclic Redundancy Check
            - multiple bits per record, detect multi-bit burst errors
        - Error Correcting Code
            - fixed ratio, capable of detection and correction
            - Reed Solution
        - Erasure Coding (distributed Reed-Soloman)
            - transforms K blocks into N
            - all can be recovered
    - disadvantages
        - slower and more complex write performance

Asynchronous Parallel I/O

- What are the advantages of asynchronous I/O
    - Advantages
        - When we have huge number of parallel clients with parallel requests, we can form deep I.O queues to improve throughput and make sure completions are processed correctly
        - having thread per operation is sometimes too expensive

        - We can efficiently handle high traffic I/O this way -> form I/O queues through non-blocking I/O

Polling for asynchronous I/O

- Explain the process of asynchronous I/O
    - there is a list of file descriptors to be check, a list of interesting events such as I/O or error, a maximum time to wait, and a signal mask. The loop sleeps until a certain interesting condition occurs and then looks at the file descriptor to execute the appropriate code (I/O operations)
    - efficient multi-stream processing
        - multiple sources of interesting input/event
        - wait for the first available, serve one at a time
- Disadvantages of asynchronous I/O
    - not practical for massive parallelism

Completions

- I honestly can't really think of anything
    - We can poll the status of any operation
    - we can await completion of some operations and then return when one or more complete or timeout
    - we can cancel or force any operations

Non-blocking I/O

- Explain the difference between polling for asynchronous I/O (multi-channel/poll select) and non-blocking I/O
    - Unlike polling for asynchronous I/O, we want to check to see if data/room is available without blocking for it -> enables parallelism and prevents deadlocks
    - NBIO is extremely simple because we only occasionally check for unlikely input + cost of wasted spins is not a concern
- Given an example of non-blocking I/O
    - a file can be opened in non-blocking mode
        - open(name, flags, | O_NONBLOCK)
    - if data is available, read(2) will return it -> otherwise it fails with EWOULDBLOCK
    - can also be used with write(2) and open(2)
- Explain using threads to achieve parallelism
    - Ex: web or remote file system server
        - if we receive thousands of requests/second + each requires multiple (blocking) operations -> so we can create a thread to serve each new request
- What are the disadvantages worker threads?
    - Thread creation is relatively expensive and we will still always have switching and synchronization issues
- How do we solve the problem that thread creation is relatively expensive?

    - Because continuous creation/destruction seems wasteful, we can recycle the worker threads. Threads can block when its operations finish and awaken when a new operation needs servicing
- When should threads be used as opposed to NBIO and Poll/Select
    - Parallel threads should be used for complex operations so that all operations can proceed in parallel, and make sure that blocking operations do not block other threads
    - However, none of the above options are truly good for massive parallelism
- What is the difference between non-blocking and asynchronous I/O
    - non-blocking operations might not be asynchronous because if data is available, then it will return immediately (polling is non-blocking but asynchronous)

Asynchronous Event Notifications

- What is the difference between sigaction and signal
    - sigaction is thread safe (reentrant) and can be used in multithreading applications
    - sigaction uses a mask (sigset_t mask) eliminates reentrancy races + siginfo passes much info about case of signal
- How do we use signals for event notifications?
    - We can use them for continuous events in normal operation
    - loss of even a single event is unacceptable
    - signals need to be safe and reliable

User-Mode drivers

- Why do we want to use user-mode drivers?
    - Kernel-mode code is brittle and if it crashes, it can take the OS with it
    - kernel-mode code is difficult to build and test because correctness rules are extremely complex + debugging tools are relatively crude
    - kernel-mode code is hard to upgrade and often necessitates rebooting the system and is also not necessarily fast

Lecture 13: File Semantics and Representation

File semantics

- Key definitions of some file semantics
    - **File:** a collection of information which is simply a linear array of bytes of which you can read or write
    - Inode number: a low level name that is given to a file
    - Directory: a list of user-readable name, inode number name pairs
    - absolute pathname: /foo/bar.txt is an example
    - File descriptor: a pointer to an object of type file
- Explain the difference between a hard and symbolic link
    - Hard link: refers to the same inode number of the original file so you simply have another file (possibly with a different name) referring to the same inode number

        - This means that if you delete one of the hard links to a file, the original file may still exist -> use link() system call
    - symbolic link is actually a file itself, of a different type. It is formed by holding the pathname of the linked-to-file as the data of the link file
    -
- What is data and metadata
    - data is the contents of the file (e.g. instructions of the program, words in the letter) and metadata is the information about the file, like how many bytes are there and when it was created -> also known as attributes

File Types and Attributes

- What are the basic attributes of files?
    - file type (regular file, directory, device, IPC port, etc.)
    - file length
    - ownership and protection information
    - system attributes (hidden/archived)
    - creation time, modification time, last accessed time
- Where are the attributes of the file stored?
    - typically stored in the file descriptor structure
- What are some extended file types and attributes
    - extended protection information such as access control lists to see who gets permission to do what to the file
    - resource forks such as configuration data, fonts, related objects
    - application defined types
        - load modules, HTML, email, MPEG
    - application defined properties
        - compression scheme, encryption algorithm

Database Semantics

- What is a database?
    - A tool managing business of critical data
    - table is equivalent of a file system
    - data organized into rows and columns where rows are indexed by unique key and columns are named fields within each row
    - supports a rich set of operations such as multi-object, read/modify/write transactions
    - SQL searches return consistent snapshots
    - insert/delete row/column operations

Object Semantics

- Explain what an object store is
    - simplified file systems, cloud storage that is optimized for large but infrequent transfers

    - bucket is equivalent of a file system that contains named, versioned objects
    - objects have long names in a flat namespace and are unique within a bucket
    - an object is a blob of immutable bytes -> get: all or part of the object, put: new version (no update or append), delete

Key-value semantics

- Explain what a key value store is
    - Smaller and faster than an SQL database as it is optimized for frequent transfers
    - table is equivalent to that of a file system -> a collection of key value pairs
    - keys have long names in a flat namespace and are unique within a table
    - value is typically a 64MB string -> you can get/put entire value or delete

File namespace

- How does the file system acknowledge/recognize files?
    - File system knows files by internal file descriptors (information stored in the pointer to an object file) while users know files by names
    - The file system is responsible for name-to-file mapping
        - associating names with new files
        - changing names associated with existing files
        - allowing users to search the name space
- What is in a file name?
    - suffixes and file types
        - file-to-application binding often based on suffix
            - defined by system configuration registry
            - configured per user, or per directory
        - suffix may define the file type (e.g. .txt, .pdf, .jpg)
        - suffix may only be a hint such as a magic number
- What is a flat namespace?
    - there must be one naming context per file system and all file names must be unique within that context
    - all files have one unique name, can be very long due to directory paths
    - file names may have some structure that may be used to optimize searches
- What is the hierarchical namespace?
    - consists of a directory or multiple directories which are simply files containing references to other files
    - the directory can be used as a naming context -> each process has a current working directory and names are interpreted relative to the directory
    - nested directories can form a tree where the file name is a path through that tree, expanding from a root node
- What are the goals of namespace?
    - Definition: set of symbols that are used to organize objects of various kinds, so that these objects may be referred to by name

    - We want to be able to give a unique name for each file, whether it is through hierarchical or flat namespace
    - We want to allow reuse of names in different contexts so that they can be structured in hierarchies to maintain simplicity
- Is DNS more like a soft link or a hard link?
    - The Domain Name system organizes websites into hierarchical namespaces -> the DNS is more like a soft link because if we delete a website URL, then we delete the website

File name conventions

- Explain the difference between true names and path names
    - true names are simply the name of the file in the flat namespace convention where every file has a unique name in the context of the file system
    - Path names are files described by directory entries and data is referred to by exactly one directory entry
- How are files named in Linux/Unix systems
    - Files are described by inodes with unique i-numbers + directories are also associated with inode numbers
    - many directory entries can refer to the same i-node

Hard Links vs Soft Links

- Explain the differences between a hard link and a soft (symbolic) link
    - Hard links (UNIX)
        - file owner sets file protection
        - all links provide the same access to the file
        - anyone with read access to file can create new link
        - but directories are protected files too
        - the file is not deleted until no links remain to the file -> keeps reference count of all links
    - Soft Links
        - another type of special file -> indirect reference/pointer to some other file
        - contents is a path name to another file
- Is a symbolic link a reference to the inode?
    - No it is not a reference to the inode, symbolic links will not prevent deletion
    - Do not guarantee ability to follow the specified path
- Name something that is similar to a symbolic link
    - An internet URL -> if you delete it, the entire site gets deleted

File System Goals

- What are the goals of file system representation?
    - **we want to ensure the privacy and integrity of all files**
    - **efficiently implement name-to-file binding**
        - find file associated with this name

        - list the file names in this part of the namespace
    - **efficiently manage data associated w/each file**
        - return data at offset X in file Y
        - write data Z at offset X in file Y
    - **manage attributes associated w/each file**
        - what is the length of file Y
        - change owner/protection of file Y ot be X
- What are the goals of file system structure?
    - we want to be able to persistently store data in an organized fashion
        - most will store user data, some will store metadata + description of the file system
        - all OS have such data structures to store this data in disk volumes that are divided into fixed-sized blocks

Goals of Free Space Representation

- What are some ways to manage allocated space?
    - **we can have a single large, contiguous extent**
        - one pointer per file, very efficient I/O
        - hard to extend, external fragmentation, coalescing
    - **a linked list of blocks**
        - one pointer per file, one per extent
        - potentially long searches
    - **N block pointers per file**
        - limits maximum file size to N blocks
        - but maybe some blocks contain pointers
- What is an extent?
    - simply a disk pointer plus a length (in blocks) -> instead of requiring a pointer for every block of the file, all one needs is a pointer and a length to specify the on-disk location
- What things should file system free-list organization must address?
    - speed of allocation and de-allocation
    - ability to allocate contiguous or near-by space
    - ability to coalesce and defragment
- What optimizations need to be made in get/release operations in file systems?
    - get/release chunks should be fast operations because they are frequent and we'd like to avoid doing as much I/O as possible
    - additionally, unlike memory, it matters what chunk we choose because it is best to allocate new space in the same cylinder as the file to avoid rotation/seek delays
    - Users may also ask for contiguous storage

Disk Partitioning

- What are some disk partitioning mechanisms?

    - Linux LVM partitions, understood by GRUB
    - mechanisms designed to support multiple OS such as DOS FDISK partitions
    - hierarchical partitioning in logical volumes within an FDISK partition
    - should be possible to boot from any partition: direct from BIOS, or w/help from L2 bootstrap
- Why do we want to divide physical disk into multiple logical disks? (logical disk partitioning)
    - we want to permit multiple OS to coexist on a single disk
        - e.g. a notebook that can boot either Windows or Linux
    - fire-walls for installation, backup and recovery
        - e.g. separate personal files from the installed OS file system
    - fire-walls for free-space
        - running out of space on one file system doesn't affect others
- What are the advantages and disadvantages of static vs dynamic partitioning
    - Static
        - advantages
            - ensures each user receives some share of the resource, predictable performance, simple to implement
        - disadvantages
            - can lead to fragmentation/wasting memory
    - Dynamic
        - advantages
            - better utilization of memory
        - disadvantages
            - decreased performance + adds complexity

BSD File system Organization

- Fast file system supported by Unix systems
- Explain some of the basic data structures supported in the BSD file system
    - Bootstrap
        - code to be loaded into memory and executed when the computer is powered on. MVS volumes reserve the entire first track of the first cylinder of the bootstrap
    - volume descriptors (superblock)
        - information describing the size, type, and layout of the file system: and in particular how to find the other key metadata descriptors
    - file descriptors
        - information that describes a file (ownership, protection, time of last update, etc.) and points where the actual data is stored on the disk
    - free space descriptors
        - list of blocks of currently unused space that can be allocated to files
    - file name descriptors
        - data structures that associate user-chosen names with each file

- What are the advantages and disadvantages of having large block sizes?
    - if we divide the volume into large, fixed-sized blocks, we will have more efficient I/O but incur **higher losses due to internal fragmentation**
- Where are the bootstrap, volume descriptor, inodes, and directories placed?
    - bootstrap is placed on the first block of each volume
    - the second block of each volume is reserved for the volume descriptor
    - we use an inode to describe each file and directories to associate names with files
    - the inodes appear the beginning of the volume, with the data blocks coming after them
    - Each cylinder acts as a mini file-system
- What kind of information does the superblock store?
    - file system identification, file system layout parameters, key device parameters, file system tuning parameters, file system status
- What kind of information is included in the file system layout parameters?
    - block size used by the file system
    - the size of a block fragment
    - total number of data blocks and inodes in the file system
    - total number of cylinder groups in the file system
    - the offset into each cylinder group where the key areas begin
- How did the Berkeley UFS solve internal fragmentation caused by smaller block sizes?
    - They broke certain blocks into smaller fragments of fixed sized
    - if after being written to and closed, a file does not fill up its last block, the last block is replaced with a smaller number of contiguous block fragments
        - the I/O throughput of small requests is better + less internal fragmentation
    - However, this further complicates the allocation and addressing of blocks
        - all of the block addresses in a file are actually fragment addresses
- How does BSD handle long file names?
    - Microsoft's approach: came up with an auxiliary directory entry scheme that could represent the extended filenames in a relatively upwards compatible fashion
    - Berkeley: Decided to use variable length directory entries that contain variable length names -> having a separate length for the directory entry and the file name
- What problems did the creation of a symbolic link address?
    - the inode number could only refer to files in the same system. There was no way to create a directory that referred to a file in another system
    - There are no ways for a user to delete a file if other links exist

FAT File Organization

- Explain some key data structures used in FAT file systems
    - bootstrap: code that is loaded into memory when the computer is powered on

    - volume descriptor (superblock): information describing the size, type, and layout of the file system, and in particular how to find other metadata descriptors in the file system
    - file descriptors: holds information about the file such as the ownership, time of last updates, and points to where the actual data is stored in the disk
    - free space descriptors: lists of blocks of currently unused space that can be allocated to files
    - filename descriptors: data structures that user-chosen names have in each file
    - the beginning actually starts with a branch instruction to the start of the real bootstrap code, then the bios parameter block, then the real bootstrap code, then an optional disk partitioning table, and then lastly a signature for error checking
- Explain the differences between BSD and DOS FAT in terms of partitioning the volume into blocks?
    - while BSD simply divides the volume into blocks (some large for efficient I/O) and segments, DOS FAT divides the volume into fixed sized physical blocks that are grouped into larger logical block clusters
- What is the file allocation table used for?
    - The FAT is used to keep track of the unused blocks, acting as a free list, while also keeping track of which blocks have been allocated to which blocks. It is placed after the bootstrap and volume descriptor
- What are the advantages and disadvantages of having a file allocation table?
    - advantages
        - entire FAT is kept in memory so that disk I/O is not necessary to find a cluster
    - disadvantages
        - searches can still be very long for very large files
        - no support for “sparse” files
- How do you find a particular block of a file in a DOS FAT system?
    - to find a particular block of a file, get number of first cluster from directory entry and follow chain of pointers through the FAT
- What does the BIOS parameter block contain?
    - number of bytes per physical sector
    - number of sectors per track
    - number of tracks per cylinder
    - total number of sectors on the volume
    - number of sectors per logical sector
    - the number of reserved sectors
    - the number of alternate file allocation tables
    - the number of entries in the root directory
- Describe what the FDISK table does in a DOS FAT system
    - a small partition table that would partition the disk into logical sub-disks so that they can incorporate multiple file systems on each disk
    - An FDISK table has four entries, each capable of describing one disk partition

        - a partition type (e.g. primary DOS, or unix)
        - an ACTIVE indication
        - the disk address where that partition starts and ends
        - the number of sectors contained within that partition
- How do the DOS file systems keep track of directories
    - keeps both file description and file naming into a single file descriptor
    - A DOS directory is a file that contains a series of fixed size (32 byte) directory entries
        - each entry describes
            - an 11-byte name (8 characters of base name, plus a 3 character extension)
            - a byte of attribute bits for the file
                - is this a file, or a sub-directory
                - has this file changed since last backup
                - is this file hidden
                - is this file read-only
                - is this a system file
                - does this entry describe a volume label
            - times and dates of creation and last modification, and date of last access
            - a pointer to the first logical block of the file
            - the length of the file
- What is the maximum number of clusters a volume can support?
    - Depends on the width of the FAT entries. Microsoft compromised and adopted a 12-bit wide FAT entry
- Where is the file allocation table kept?
    - Because the file allocation table is so small, it can be kept in memory so that disk I/O is not necessary to find the cluster. This is useful when we have to look up next block pointers in the file allocation table
- Why is Garbage collection necessary in DOS FAT systems?
    - because clusters are not freed when files are deleted, we won't be able to reallocate them until we collect them
- Explain how garbage collection is done
    - starting from the root directory, DOS would find every "valid" entry. It then follows the chain of next block pointers to determine which clusters were associated with each file, and recursively enumerate the contents of sub-directories
    - Any cluster not found in some file was free, and marked them as such in the FAT
- How did the DOS FAT system handle long file names?
    - Microsoft decided to **extend to filenames in additional directory entries**
    - Each file would be described by an old-format directory entry, but supplementary directory entries could provide extensions to the file name

        - **To keep older systems from being confused by the new directory entries, they were tagged with an unusual set of attributes** (hidden, system, read-only, volume label)
            - Older systems would be able to ignore these entries but newer systems would be able to recognize that they were extensions of the additional directory entries
- What are some of the mistakes that DOS made that were fixed in new versions of file systems (ISO 9660 standard)?
    - ISO 9660 made:
        - variable length file names and created a directory that begins with a length (extent)
        - variable length extended attributes section such as file owner, owning group, permissions, creation, modification, effective, and expiration times, record format, attributes, and length information

Indexed data extents (Inodes)

- Evaluate inode performance
    - inode is in memory whenever the file is open and the first ten blocks can be found without I/O, meaning for those ten blocks -> efficient
    - Any block can be found with 3 or fewer reads due to indirect blocks and block I/O that will buffer data in cache
- What are the differences in the steps needed to find a particular block in UNIX and DOS FAT systems?
    - in UNIX systems, we need to read in the indirect blocks from the inodes to find the data blocks while in DOS FAT systems, we get the number of the first cluster from directory entry and follow the chain of pointers through FAT. Since no I/O has to be done with the FAT because it can reside in memory, it is fairly efficient for short searches

Linked Data Extent

- Definition of extent: A contiguous area of storage reserved for a file in a file system, represented as a range of block numbers -> as defined by wikipedia
- What are the advantages of extent allocation?
    - Results in less file fragmentation and can also eliminate most of the metadata overhead of large files that would traditionally be taken up by the block-allocation tree
    - Instead of having a pointer in every block of the file, we can have one pointer that also specifies the length in order to determine the location on disk

Bit Map Free Lists

- Explain the purpose of the bitmap
    - the bitmap is an allocation structure with bits to indicate whether the corresponding block is free (0) or in-use (1)

    - BSD file systems use bit-maps to keep track of both free blocks and free inodes in each cylinder group
- What are the advantages of having bitmaps
    - bitmaps are a very efficient representation with minimal space to store the map and the code is extremely cache efficient to search
    - bitmaps also enable efficient allocation because it is easy to find chunks in a desired area and it is easy to coalesce adjacent chunks

FAT free lists

- Explain how the DOS FAT free lists work
    - If a block is allocated in a file, the FAT entry gives the logical block of the next available logical block instead of returning an error
    - The file allocation table is used as both a free list and to keep track of which blocks have been allocated to which files

BSD Directory entries

- can multiple directory entries point to the same inode?
    - yes, multiple directory entries can point to the same inode because the association of a name with an inode is called a link
    - the actual file descriptors are the inodes and directory entries only point to inodes
- What is in a UNIX directory entry
    - the name (relative to the directory) and the pointer to the inode of the associated file
    - the directory entry is essentially a struct of inode and filename -> information to translate from a filename to an inode and get to the actual file
    - basically the mapping of the filename to the inode -> pointer to the inode
- What are the differences between a DOS system's directory entry and a BSD directory entry
    - the DOS system's directory entry contains a pointer to the first cluster of the file because the FAT entry for that cluster tells us the number next cluster in the file
    - the directory entries are the file descriptors in DOS -> **only one entry can refer to a particular file**
    - **DOS directory entries contain the name, type, location of the first cluster of file, length of file, and other privacy and protection attributes**

FAT File Descriptors

- What is the purpose of a FAT file descriptor
    - FAT file descriptors act as directory entries for DOS systems, meaning that they must be unique and contains information about the name, type, location of the first cluster of the file, length of the file, and other privacy and protection attributes
    - If the first character of a file's name is NULL, the directory entry is unused
    - Also contains byte attribute bits for the file which include, whether the file has changed since last backup, whether the file is read-only, whether the file is a

system file, whether the entry describes a volume label, whether the file is hidden, and whether the file is a file or a sub-directory

The Mount Operation

- What are the goals of file system mounts (UNIX)
    - make many file systems appear to be one giant
    - users need not be aware of file system boundaries -> keep boundaries transparent
- How are unix file system mounts created?
    - the mechanism mounts the device on directory
    - creates a warp from the named directory to the top of the file system on the specified device
    - any file name beneath that directory is interpreted relative to the root of the mounted file system
    - When mounting a file system, the OS will read the superblock first to initialize various parameters, and then attach the volume to the file-system tree

File access layers of abstraction (Object Storage)

- Object storage is a computer data storage architecture that manages data as objects, as opposed to other architectures like file systems which manage data as a file hierarchy and block storage which manages data as blocks within sector and tracks
- What are the advantages of object storage?
    - Object storage systems allow retention of massive amounts of unstructured data
    - Objects contain additional descriptive properties which can be used for better indexing or management
    - abstracts some of the lower layers of storage away from admins and applications
    - Administrators do not have to perform lower-level storage functions like constructing and managing logical volumes to utilize disk capacity
    - allows the addressing and identification of individual objects by more than just file name and file path

Filesystem in Userspace (FUSE/user mode file systems)

- What are the advantages of FUSE?
    - they don't reside in the kernel, so naturally it is easier to distribute the code
    - easier to test, review, and justify
    - you can quickly update the file system if there is an enhancement or a bug fix
    - if the file system crashes for some reason, it doesn't necessarily take down the entire OS
- Explain what the virtual file system layer is
    - **a federation layer to generalize file systems and permits rest of OS to treat all file systems as the same**
    - **support dynamic addition of new file systems**
    - plug-in interface or file system implementations

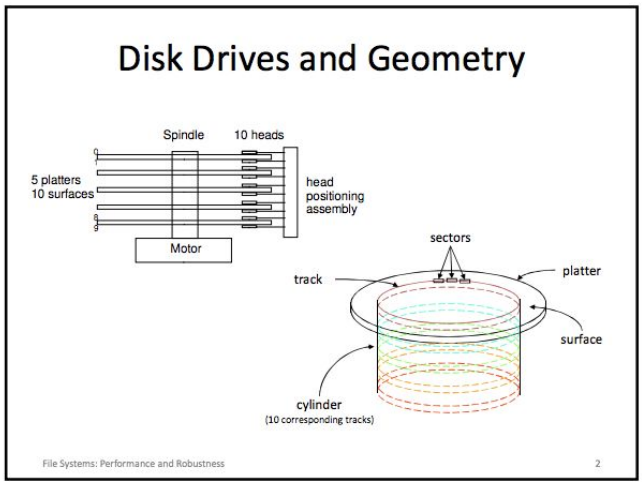
    - implementation is hidden from higher level clients
        - all clients see are the standard methods and properties
    - Unlike file systems, virtual file systems don't actually store data themselves -> act as a view of translation of an existing file system or storage device

WEEK 8

Lecture 14

Disks and file

- What were the problems that UNIX file systems had that were addressed in FFS?
    - the performance of UNIX file systems were horrible and gets worse with time as well
    - The main issue was that the old UNIX file system treated the disk like it was a random-access-memory
        - this meant that data was spread all over the place without regards to how expensive positioning is
            - data blocks were often very far from its inode, inducing an expensive seek whenever one first read the inodes
            - Misuse of data blocks can lead to data fragmentation
- How do you maximize cylinder locality?
    - seek-time dominates the cost of disk I/O (greater than or equal to rotational latency) because and much harder to optimize by scheduling
    - live systems do random access disk I/O which are spread all across the disk, including directories, inodes, programs, data, and swap space
    - since the access is also not uniformly random, we can create lots of mini-file systems to solve the above problems -> group each with inodes, directories, data, and significantly reduce the seek distance
    - Since consecutive block allocation helps reduce disk seek/latency, more requests can be satisfied in a single rotation and thus overhead is significantly decreased
- Draw the disk drives and file system design

- 

Block Size and Internal Fragmentation

- What is the ideal transfer size?
  - The bigger the transfer size, the more efficient the I/O is because we can amortize fixed per-op costs over more bytes/op and multi-megabyte transfers are usually very good. However, this would require space allocation units, and since the transfer size is big, the allocation chunk with also be very large, which can lead to internal fragmentation
    - therefore we need variable partition in order to address this issue
- What are the advantages and disadvantages of having larger block sizes?
  - advantages
    - More efficient I/O because DMA startup time, seek, rotation, and interrupt service overheads are high
    - fewer, larger I/O operations
  - disadvantages
    - Larger block sizes means more internal fragmentation -> mean loss = block size / (2*mean file size)
    - small files with large chunk sizes could cause internal fragmentation
- Why is variable partition disk allocation difficult in file systems?
  - many files are allocated for a very long time, space utilization tends to be high (60-90%), and special fixed-sized free lists don't work as well

Read cache

- What are the goals of read caching?
  - Because disk I/O takes a very long time, we want to have deeper queues since large transfers improve efficiency (created through parallel requests -> asynchronous, many clients making requests at the same time, etc.), but this does not improve the performance significantly
  - Therefore the goal of read caching is to eliminate as much of our disk I/O as possible through making use of locality and read-ahead caching

- Explain how read-ahead works
    - we request blocks before they are requested by the client and store them in the cache until later
    - This reduces wait time and may also improve disk I/O
    - Since the contents will be read from main memory instead of from the disk, there will be no disk I/O necessary
- When does read-ahead caching make sense?
    - when client specifically requests sequential access
    - when client seems to be reading sequentially
- What are the risks of read ahead caching
    - may waste disk access time reading unwanted blocks
    - may waste buffer space on unneeded blocks

Write-through

- What is write-through
    - write-through is when the write is done synchronously both to the cache and to the backing store
    - the data will go to main memory very quickly because when the write request is made, it is not buffered and is instead placed in the cache and main memory synchronously

General/Special Purpose Caches

- How is the performance gain vs the cache size for special purpose and general block caches?
    - the performance gain is better for special purpose caches and they asymptotically increase as the cache size increases
- What does a special purpose cache do?
    - special purpose caches store inodes and recently used inodes likely to be reused + directory entries as well
- Steps to solve special cache math problems
    - consider the hits per byte per second ratio
        - eg. 2 hits/4K block (0.0005 hits/byte)
        - e.g. 1 hits/32 byte dcache entry (0.03 hits/byte)
    - consider the savings from extra hits
        - e.g. 50 block read/second * 1.5ms/read = 75ms
    - consider the cost of the extra cache lookups
        - e.g. 1000 lookups/s * 50ns per lookup = 50*10^-6 ms
    - consider the cost of keeping cache up to date
        - e.g. 100 upd/s * 150ns per upd = 15us
    - net benefit: 75ms - (15+60us) = 74.9335m/s

How to beat LRU

- When can we outsmart LRU?

    - sometimes we know what we won't reread such as
        - load module/DLL read into a shared segment
        - an audio/video frame that was just played
        - a file that was just deleted or overwritten
        - a diagnostic log file
    - By dropping these files from the cache, we can allow a longer life to the data that remains there

Delayed Writes/Write back cache

- What are the advantages and disadvantages of write-back cache
    - advantages
        - eliminates moot writes
        - aggregates small writes into large writes
        - accumulates large batches of writes -> deeper queue enables better disk scheduling
    - disadvantage
        - may not have same copy of data in cache and in main memory
- How does delayed writing improve performance?
    - Decreases the amount of I/O needed to be done + overhead of startup times of seek/disk, etc.
    - By deferring updates of data into permanent storage, delayed write operations save extra disk operations for files that are often overwritten
    - you can use the fsync command to write to disk immediately if you want to
- What is the worst case scenario of deferred writes?
    - process allocates a new block to file A
        - we get a new block (x) from the free list
        - we write out the update inode for file A
        - we defer free-list write-back (happens all of the time)
    - the system crashes, and after it reboots
        - a new process wants a new block for file B
        - we get block x from the stale free list
    - two different files now contain the same block
        - when file A is written, file B gets corrupted
        - when file B is written, file A is corrupted

Detection and Repair

- What can go wrong in a file system?
    - data loss
        - file or data is no longer present
        - some/all of data cannot be correctly read back
    - file system corruption
        - lost free space
        - references to non-existent files

        - corrupted free-list multiply allocates space
        - file contents over-written by something else
        - corrupted directories make files unfindable
- How can we reduce potential damage?
    - Ordered writes
        - **write out data before writing pointers to it**
            - unreferenced objects can be garbage collected
            - pointers to incorrect info is more serious
        - **write out deallocations before allocations**
            - disassociates resources from old files ASAP
            - free list can be corrected by garbage collection
            - shared data is more serious than missing data
- How do we design file system structures for audit and repair (what techniques do we use)?
    - redundant information in multiple distinct places
        - maintain reference counts in each object
        - children have pointers back to their parents
        - transaction logs of all updates
    - all resources can be garbage collected
        - discover and recover unreferenced objects
    - audit file system for correctness (prior to mount)
        - all object-well formatted
        - all references and free lists correct and consistent
    - use redundant info to enable automatic repair
    - Integrity checking a file system after a crash
        - verifying check-sums, reference counts, etc.
        - automatically correct any inconsistencies
        - However, checking a sum can be extremely long and inpractical
    - Journaling (covered later)
- What are the advantages and disadvantages of ordered writes
    - Definition of ordered writes: writing the data out before writing the pointers to the data
    - Advantages
        - can reduce potential damage because unreferenced objects will be garbage collected + pointer will never point to garbage
    - Disadvantages
        - greatly reduced I/O performance because of elimination of head/disk motion scheduling + elimination of accumulation of nearby operations + elimination of consolidation of updates to same block
        - may not be possible because modern disk drives reorder queued requests
        - does not remove incomplete writes and chooses minor problems over major ones

- What are the main crash scenarios that can occur and what are the potential dangers each bring?
    - When we append a file, we are adding a new data block and thus we must update the **inode, the new version of the bitmap, and the data blocks**
        - **Just the data block:** if just the data block is written, it is not a problem for crash consistency because neither the bitmap nor the inode indicate that it has even been allocated
        - **Just the updated inode:** the inode points to the disk address where the data is going to be written but it has not. Thus the inode points to garbage value and we can read garbage from the disk
        - **Just the updated bitmap:** the bitmap indicates that the block is allocated, but there is no inode that points to it so therefore it may never be used by the file system, so a **space leak** can occur
        - **Inode and Bitmap:** If both the inode and the bitmap are written to disk but not the data, everything looks OK from the file system's perspective but the data will point to garbage data
        - **Data block and inode:** If the data block and inode are written to disk but not the bitmap, there will be no indication about this in the bitmap and thus we will have an inconsistency between the inode and the old version of the bitmap
        - **Bitmap and Data block:** We will have no idea which file it belongs to because there is no inode to point to the file

Journaling and Recovery

- Explain the main concept of journaling (what it is, pretty much)
    - When updating the disk, before overwriting the structures in place, first write down a little not (somewhere else on the disk, in a well-known location) describing what you are about to do
    - We create a circular buffer journaling device where journal writes are always sequential, can be batched, and is relatively small, much like NVRAM
    - journal all intended file system updates such as inode updates and block write/alloc/free
    - efficiently schedule actual file system updates
        - write back cache, batching, motion-scheduling
        - journal completions when real writes happens
- Why do we batch our journal write entries?
    - If small writes are continuously requested, it can be highly inefficient, so therefore we accumulate batches until full or max wait time
    - the operation is safe after journal entry has been written, so the caller must wait for this event to happen
- What is the process of checkpointing?
    - Once a transaction is safely on disk, we can overwrite the old structures in the file system -> issue the writes to their disk locations

- Explain the steps of data writing in a Linux ext3 file system
    - 1) Data write: write data to final location -> wait for completion
    - 2) Journal metadata write: Write the begin block and metadata to the log -> wait for the write to complete
    - 3) Journal commit: write the transaction commit block to the block -> wait for the write to complete -> transaction is now committed
    - 4) Checkpoints metadata: write the contents of the metadata to their final locations within the file system
    - 5) Free: later, mark the transaction free in journal superblock
    - Essentially, we force the data writes before writing the pointer so that the pointer does not point to garbage
- Explain the process of journal recovery
    - Because a journal is just a circular buffer, it can be recycled after old operations have completed + timestamps distinguish new entries from old
    - after system restart, review the entire journal and note which operations have completed
    - then perform all writes not know to have completed -> data and destination are both in the journal and all of these write operations are **idempotent**
    - truncate journal and resume normal operation
- Why are journal writes much faster than data writes?
    - All journal writes are sequential and so there is no competing head motion
    - In normal operation, journal is write-only meaning that the file system never reads/processes the journal
    - scanning the journal on restart is also very fast because it is very small and is read sequentially with efficient reads, and the entire process is done in memory so that no disk I/O is necessary
- How can check-sums be used for data recovery?
    - parity can be used to detect single-bit errors as well as restoring lost copies through XORing and then comparing the old values with the new values
    - Error correcting codes can detect double-bit errors and also correct single bit errors
    - Cryptographic hash functions have a very high probability of detecting any change
- ****** WRITE CODE FOR BATCHED JOURNAL ENTRIES**

```
writer:

if there is no current in memory journal page
    allocate new journal page
add my transaction to the current journal page
if current journal page is now full
    do the write, await completion
    wake up processes waiting for this page
else
    start timer, sleep until I/O is done (for writing)


flusher:

while true
    sleep()
    if current-in-memory page is due
        close page to further updates
        do the write, await completion
        wake up other processes waiting for page
```

Metadata Journaling

- Why do we want to journal metadata?
  - It is small and very random so the I/O is very inefficient
  - it is integrity-critical and there is a huge potential data loss
  - Write traffic doubles because for each write, we are also writing to the journal
    - data block is instead written to the file system proper, avoiding the extra write, given that most I/O traffic to the disk is data, not writing data twice substantially reduces I/O
    -
- why do we not want to journal data?
  - it is often large and sequential (I/O efficient)
  - it would consume most of journal capacity/bw
  - it is less order sensitive (just precede metadata)

Copy on Write File Systems

- Explain what a copy on write file system does
  - Never overwrites files or directories in place
  - Rather, it places new updates to previously unused locations on disk
  - After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures
- What is copy on write?
  - Copy on write is an optimization strategy in which we only copy the contents when modifications are made

    - If you change some data, it's written in one or more unused blocks, and the original data remains unchaged
    - if a resource is duplicated but not modified, it is not necessary to create a new resource -> it can be shared between the copy and the original
- How do COW systems provide backups?
    - they can be used to take snapshots (state of computer storage) that store only the modified data close to main memory.
    - Although this is a weak form of backup, it avoids fuzzy backups
    - When implementing snapshots, there are two techniques
        - the original storage is never modified -> when a write request is made, it is redirected away from the original data into a new storage area (redirect on write)
        - when a write request is made, the data are copied into a new storage area, and then the original data are modified

Checksums and Scrubbing

- What are some examples of checksum operations
    - Simple Data Checksums
        - parity and ECC are stored within the data
            - to identify and correct corrupted data
            - controller does encoding, verification, correction
        - very effective against single-bit errors
            - which are common in storage and transmission
        - strategy: disk scrubbing
            - slow background read of every block on the disk
            - if there is a single-bit error, ECC will correct it
                - before it can turn into a multi-bit error
    - Higher Level Checksums
        - store the checksum separate from the data
            - it can still be used to detect/correct errors
            - it can also detect valid but wrong data
        - many levels at which to check-sum
            - inode stores a list of block checksums
                - in de-dup file systems, check sum is block identifier
            - inode stores checksum for the entire file
                - if file is corrupted, go to a secondary copy
            - hierarchical checksums all the way up the tree
    - Delta Checksum Computation
      a checksum of many blocks is expensive because each block must be read and summed
        - updating any block requires a new checksum
            - computer checksum (newBlock) - checksum (oldBlock)
            - add that to the checksum(alloBlocks)

- Explain what scrubbing is
    - an error correction technique that uses a background task to periodically inspect main memory or storage for errors, then correct detected errors using redundant data in the form of different checksums or copies of data
    - In RAID, a RAID controller may periodically read all hard disk drives in a RAID array and check for defective blocks
    - Basically, periodically reads all the blocks in the system and checks whether the checksum is still valid -> reduces the change that a certain data item becomes corrupted
- What are the overheads of checksumming?
    - Space overheads
        - each stored checksum takes up room on the disk which can no longer be used to store data blocks
        - when accessing data, there must now be room in memory for the checksums as well as the data as well
            - however, if the system simply checks the checksum and then discards it once it is done, this is not much of a problem
    - Time overheads
        - Minimally, the CPU must compute the checksum over each block, both when the data is stored as well as when it is accessed to compare it against the stored checksum
        - Can induce I/O overheads, particularly when checksums are stored distinctly from the data + background data scrubbing
- Explain Misdirected Writes and how they are handled
    - arises in disk and RAID controllers which write the data to disk correctly but are written to the wrong location
    - Solution: add a little more information to the checksum -> such as a physical ID to keep the disk and sector number of the block so that it is easy for the client to determine whether the correct information is on the block
- Explain what Lost Writes are and how they are handled
    - Lost writes occur when the device informs the upper layer that a write has completed but in fact it never persisted
    - **Write verify/read-after-write:** read back the data after a write -> system can ensure that the data indeed reached the disk surface

Log Structured File Systems

- What are the issues with log structured file systems?
    - Recovery time to reconstruct index/cache
    - log defragmentation and garbage collection is necessary
- What are the key components of the log structured file systems
    - the journal is the file system
        - all inodes and data updates are written to the log

        - updates are Redirect-on-Write -> instead of copying to a new location -> the system merely redirects the pointer for that block to another block and writes the data there
        - in memory index caches inode locations
    - Tip: remember flash file systems and key/value stores
- How do we manage the read performance of a log structured file system?
    - we have in memory index that points to latest inode versions
    - recent log segments are LRU cached in memory
    - Logical to physical page mapping is used to improve performance -> kinda like a cache
- How do we manage log recovery time?
    - **In memory index (logical to physical page mapping) is the key to performance as searching the entire log is prohibitively expensive**
    - We can reconstruct the index on restart but rescanning the log will be prohibitively expensive
    - **Therefore, we periodically snapshot index to the log and update a most-recent-index pointer**
    - We recover from failure through finding and recovering the last index snapshot and replaying all valid log entries at that point
- How do log structured file systems solve the “small write problem”
    - the small write problem occurs when the FTL must read a large amount of live data from the old block and copy it into the new one
        - however, data copying increases overhead and decreaSes performance drastically
    - Therefore, the FTL keeps a few blocks erased and directs all writes to them\
- Evaluate SSD performance and cost
    - performance
        - unlike hard disk drives, flash-based SSDs have no mechanical components and are similar to DRAM in many ways
        - Performance of SSDs greatly outstrips hard drives, even when performing sequential I/O
    - Cost
        - very expensive so many systems use a combination of hard drives and SSDs
- How does an LFS determine block liveliness?
    - the LFS includes, for each data block D< its inode number and its file offset that is recorded in the head of a structure as the **segment summary block**
    - So therefore we can look in the summary block and find the inode number and the offset
        - look in the imap to find where N lives and read N from disk
        - Using the offset, look in the inode to see where the inode thinks the Tth block in the file is on disk
        - if it’s pointing to the address it’s expecting it is alive, otherwise it is not

    - record the new version number in the imap so that the LFS can short circuit the longer check if the version number is reorder in the on-disk segment
- When should defragmentation occur?
    - the fast and easy way …. offline
        - backup then entire file system to other media
        - reformat the entire file system
        - read the files back in one-at-a-time
    - the slow and hard way
        - find a heavily fragmented area
        - copy all files in that group elsewhere
        - coalesce the newly freed space
        - copy files back into the defragmented space
- How does defragmentation occur?
    - 1) identify stale records that can be recycled
        - versions, reference counts, back-pointers, GC
    - 2) identify next block to be recycled
        - most in need (oldest in log, most degraded data)
        - most profitable (free space ratio, most stable)
    - 3) recopy still valid data to a better location
        - front of the log, contiguous space
        - or perhaps just move it out of the way
    - 4) recycle the now completely empty block
        - for flash, erase it, add it to the free list
- Why is compaction and defragmentation beneficial to the disk?
    - file I/O is efficient if file extents are contiguous
        - easy if free space is well distributed in large chunks
    - with use the free space becomes fragmented so the file I/O involves more head motion
    - therefore, periodic in-place compaction where we move the most popular files to the inner-most cylinders, copy all files into contiguous extents, and leaving the free-list with large contiguous extents can significantly speed up I/O

Lecture 15

Confidentiality

- What is confidentiality in terms of computer security?
    - the system should keep its secrets. If some information is supposed to be hidden from others, don't allow them to find out
    - keep other people from seeing your private data

Integrity

- define integrity in terms of computer security
    - keep other people from changing your protected data

    - if some piece or component is supposed to be in a particular state, then don't allow anyone to change it

Controlled Sharing

- define controlled sharing in terms of computer security
    - you can grant other people access to dat but they can only access it in ways you specify
    - if some information or service is supposed to be available for your own or for other's use, make sure an attacker can't prevent this

Object, Agent, Principal

- Define object, agent, and principal
    - Object: the routine or component that the subject wants to access
    - Principals: (e.g. users) own, control, and use protected objects
    - Agent: (e.g. programs) that act on behalf of principals
- Define authentication, credentials, and authorization
    - authentication
        - confirming the identity of a requesting principal
        - confirming the integrity of a certain request
    - credentials
        - information that confirms the identity of requesting principal
    - authorization
        - determining if a particular request is allowed

Mediated Access

- define mediated access
    - mediated access: the agents must access objects through control points
- What is complete mediation?
    - Security term meaning that you should check if an action to be perform meets the security policies every single time the action is taken
- Explain the steps of complete mediation
    - to get access, issue request to the resource manager
    - resource manager then consults the access control policy data
    - access may be granted directly if the resource manager maps resource into process or it may be granted indirectly if a resource manager returns a "capability" to the process which can be used in subsequent requests
- What are some different types of access mediation?
    - Per-operation Mediation (e.g. file)
        - all operations are via requests
        - we can check access on every operation
        - revocation is simple (Cancel the capability)
        - access is relatively expensive (system call/request required)
    - Open-Time Mediation (e.g. shared segments)

        - one-time access check at open time
        - if permitted, resources are mapped in to process
        - subsequent access is direct (very efficient)
        - revocation may be difficult or awkward
- What are the advantages and disadvantages of per operation mediation and open-time mediation
    - per operation mediation
        - advantages
            - easy to revoke access because you simply need to cancel the capability
            - we can check access on every operation -> finer grained
        - disadvantages
            - can be relatively expensive/inefficient because requests would be done through system calls
    - open-time mediation
        - advantages
            - efficient and often fast because access is directly in process
            - if access is permitted, resources are mapped into the process
            - one-time access check at open time
        - disadvantages
            - can be relatively expensive/inefficient to revoke access

Revocability

- Explain the advantages and disadvantages of direct (e.g. shared segment) vs indirect access (e.g. file)
    - advantages of direct
        - access check is only performed one
        - very efficient, process can access resource directly
    - disadvantages of direct
        - process may be able to corrupt the resource
        - access revocation may be awkward
    - advantages of indirect
        - only resource manager actually touches resource
        - resource manager can ensure integrity of resource
        - access can be checked, blocked, revoked at any time -> through capabilities
    - disadvantages of indirect
        - overhead of system call every time resource is used

Trusted Computing Platform

- Why does the operating system need to be a trusted computing base?
    - all protection information is stored in the OS and applications cannot directly access/modify it

    - OS creates and maintains process state so that it can associate a principal with each process
    - OS implements file, process, IPC operations
    - Last but not least, the OS is a foundation on which apps run and can depend on the processes and files within
- Is all trusted code located in the OS kernel?
    - No. File system management and backup, login and user account management, and network services do not necessarily belong in the OS kernel

Principles of Secure Design

- What are the principles of secure design and explain each.
    - **Economy of mechanism:** keep your system as small and simple as possible
    - **Fault safe defaults:** default to security, not insecurity -> if policies can be set to determine the behavior of a system, have the default of those policies be secure. Essentially, setting a firewall to only allow certain good traffic through
    - **Complete mediation:** security term meaning that you should check if an action to be performed meets security policies every single time the action is taken
    - **Open design:** assume your adversary knows every single detail about the design. Base your security on these assumptions
    - **Separation of privileges:** Require separate parties or credentials to perform critical sections
    - **Least Privilege:** give a user or a process the minimum privileges required to perform the sections you wish to allow
    - **Least common mechanism:** for different users of processes, use separate data structures or mechanisms to handle them
    - **Acceptability:** a critical property not dear to the hearts of many programmer. If your users won't use it your system is worthless -> essentially, don't ask too much of your users

Authentication

- Explain the steps of internal (process) authentication
    - OS associates credentials with each process
        - stored, within the OS, in the process descriptor
        - automatically inherited by all child processes
        - identify the agent on whose behalf requests are made
- How do we ensure internal process authentication correctness
    - we can use capabilities or access control lists to ensure that the commands are coming from the indicated principal
- Explain what external (user) authentication is
    - authentication done by trusted "login" agent
        - typically based on passwords and/or identify tokens
        - movement towards biometric authentication

    - ensures secure passwords so that they may be guessable/brute forceable and stealable
    - ensures secure authentication dialogs
        - protection from crackers: humanity checkers
        - protection from snoopers: challenge/response
        - protection from fraudulent servers: certificates
    - evolving encryption technology can help in a massive way
- What does challenge/response authentication prevent?
    - Challenge/response authentication prevents man-in-the-middle attacks
    - Client and server agree on a complex function and the server issues a random challenge string to the client. The server and client both calculate the function, client sends the answer to the server, and the server validates it
- How can we categorize authentication of users (humans)
    - authentication based on what you know
        - ex: storing a password -> hash of password kept in database so that if it is leaked, since hashes are not reversible in nature, the attacker cannot decrypt
    - authentication based on what you have: ATM cards, hardware allows it to determine that we have the correct card and further authentication can be done through requesting a PIN number
    - authentication based on what you are: fingerprint authentication, characteristics of human behavior that are unique
    - authenticating by non-human: web server -> doesn't have some human user logged in whose identity is unknown

Authorization

- What is the purpose of finer granularity authorization?
    - super users can potentially be extremely dangerous and are permitted to do anything. Therefore, they can accidentally mistype commands and present danger to the operating system
    - Therefore, it is good to have finer granularity of privilege through backups, file system allocation, user creation, etc.
    - It is also good to have finer granularities of operations so that privilege is only granted for one operation at a time
- Why do we want to have finer granularities of operations?
    - with privilege granted for only one operation at a time, we can keep confirmation dialogs in system management tools and ensure that important code is secure

Access Control Lists

- What is an access control list and how is it used?
    - An access control list is a list that each file has that states who has what permission to access the file
    - Per object access control

    - record, for each object, which principals have access
    - each protected object has an access control list
    - OS consults ACL when granting access to any object
- Where is the access control list stored?
    - can be stored in persistent storage like the disk, or for efficiency, can be stored somewhere as metadata like the inode.
- What are the advantages and disadvantages of access control lists
    - advantages
        - short to store and easy to administer
        - if you want to change the set of subjects who can access a resource, you just change the ACL
        - since the ACL is kept near the file (near the metadata of the file) you can get to the file and get to all the relevant access control information
    - disadvantages
        - having to store the ACL near the file and dealing with potentially exhaustive searches of long lists
        - in a distributed environment, you need to have a common view of identity across all the machines for ACLs to be effective

Linux File Protection (ACLs)

- How did Linux solve the issue that we reserving space for all users would be a waste of space?
    - Unix designers used 9 bits for each file ACL, and realized that there were effectively three modes that we actually cared about
    - Read, write, and execute that are partitioned into three groups -> owner of the file, the members of a particular group/group ID, and everybody else
    - solves problem of storage being eaten up
- What is stored in a Unix access control list?
    - subject credentials: user and group ID, established by password login
    - supported operations: read, write, execute, chown, chgrp, chmod
    - representation of ACL info
        - rules (owner:rwx, group:rwx, others:rwx)
- When is an access control list a good fit?
    - When there are more user-centric actions required. Because capabilities are per-agent, it'd be easier to have per-object access control lists
    - In an enterprise system, a user-privilege level may change frequently, which is difficult to manage in a capability list

Capabilities/Linux Capabilities (file descriptors)

- Explain what capabilities are
    - Essentially, capabilities are just a bunch of bits that act as a sort of lock and key system to have access to a resource

    - They are per-agent access control mechanisms that record, for each principal, what it can access. Each granted access is called the “capability” and is required to access any system object
- What are the primary differences between ACLs and capabilities
    - ACLs are per-object access control mechanisms while capabilities are per-agent access control mechanisms
- When do we want to use capabilities over access control lists?
    - When we do not have to change the user-privilege that often, a capability list can be a good fit
- How do Unix files also use capabilities?
    - If a process wants to read or write a file
        - it must open the file, requesting read or write access
        - open will check permissions before granting access
        - if operation permitted, OS returns a file descriptor
    - The file descriptor is a capability
        - it is an unforgeable token conferring access to the file
        - it confers a specific access to a specific file
        - a required argument to the read/write system calls
        - without a file descriptor reads/writes are impossible
    - In summary, after using the ACL to check if the access permissions are valid for the object, it returns a file descriptor that acts as a capability
- How do we ensure that capabilities are truly unforgeable
    - real capabilities come from a trusted source such as the OS who checks access permissions before granting them
    - Therefore, we can ensure that capabilities are unforgeable through keeping them inside the OS
        - We can give the user an index into a per-process table (e.g. user file descriptors are index into a per-process array) and the process can therefore only refer to the capabilities through their index numbers
- Can a system call pass capabilities to others?
    - Yes because only the OS can create the table entries
- What are some very hard-to-forge capabilities
    - random cookies from sparse namespaces
        - can be very verified but difficult to forge and thus is easily achieved as encryption technology
    - Resource manager can decrypt cookies on each request and determine which object is to be used + ensure requestor has adequate access for each operation
    - cookies are easily exchanged in messages

Principle of Least Privilege

- What is the principle of least privilege?

    - As mentioned earlier, the principle of least privilege is a principle of secure design that means that we should only grant the minimum amount of privilege required to perform the operation
    - This means that we should surrender privileges when they are no longer needed and operate in the most restricted possible context
- How do we ensure least privilege?
    - Allow minimum possible access to resources by applying multiple levels of protection
    - run sanity check requests before performing the operation
    - minimize amount of privileged software, minimize the attack surface, and minimize amount of code being audited

Role Based Access Control (RBAC)

- What is an issue that large institutions may face when attempting to use standard access control to implement a security method
    - they saw that people performing different roles in the company required different privileges
    - organizing access control on the basis of particular roles is difficult, and is particularly valuable if certain users are allowed to switch roles
- Explain the purpose of the RBAC
    - users are authorized to perform roles so to the operating system, they are not “a person”. Therefore, they must declare they are operating in a role sand checks their authorization to function in use and create credentials to authorize role based operations.
    - Privileged operations check role credentials -> makes it easy to check for role-specific privileges
- How do we change the role of a user?
    - we can simply strip the label off of a certain position when a user switches roles
- How does RBAC provide superior authorization control?
    - fine grained operation control for limited periods
    - audit records record the “real person” who took the actions

Trojan Horses

- What are trojan horses?
    - accidental bugs in trusted software that creates holes
    - example: phishing -> pretending to be the login program, pretending to be financial institution web page

Cryptographic Hashes

- Explain the steps of using cryptographic hashes
    - 1) start with a message you want to protect
    - 2) compute a cryptographic hash for that message
        - e.g. using MD5

        - 3) transmit the hash over a separate channel
        - 4) recipient computes hash of received text
            - if both hash results agree, the message is intact
            - else message has been corrupted/compromised
        - 5) hash must be delivered over a secure channel
            - encrypted, or otherwise separated and trusted
            - or else bad guy could just forge the validation hash
- Another way to explain above question
    - send message through secure and insecure transmissions
    - use the cryptographic hash to hash the message and then compare the results
    - if the results are the same, the message is in tact
- What are cryptographic hashes?
    - Essentially, cryptographic hashes are very strong checksums that can be described as
        - unique: two messages won't produce the same hash
        - one-way: cannot infer original input from output
        - well distributed: any change to input changes output
    - much less expensive than encryption

Challenges of Security

- Why is so security so difficult to achieve?
    - Complexity of software and systems
        - millions of lines of code, thousands of developers
        - rich and powerful protocol APIs
        - numerous interactions with other software
        - constantly changing features and technology
        - absence of comprehensive validation tools
    - Determined and persistent adversaries
        - commercial information theft/blackmail
        - national security sabotage
- What are the key elements of security
    - **reliable authentication**
        - we must be sure who is requesting every operation
        - we must prevent masquerading of people/processes
    - **trusted policy data**
        - policy data accurately describes desired access rules
    - **reliable enforcement mechanisms**
        - all operations on protected objects must be checked
    - **audit trails**
        - reliable records of who did what, when

Challenge/Response Authentication

- What are some potential weaknesses in the challenge/response authentication method?

    - if the server sends out the same challenge more than once, an attacker who recorded the hash of the first authentication can simply replay the hash without knowing the password

Linux setuid

- What are some special application privileges?
    - privileged daemons: started by the OS
        - many system daemons run as the super user
        - others are run as the owner of key resources
    - privileged commands: run by users
        - UNIX SetUID/SetGID load modules
        - run with the credentials of the program's owner
        - may be able to create/set their own credentials
            - e.g. login, sudo
        - these must be very carefully designed/reviewed
- What does the Linux setuid command do?
    - allows users to run an executable with the permissions of the executable's owner or group and to change behavior in directories
    - Often used to allow users on a computer system to run programs with temporarily elevated privileges in order to perform a certain task

At rest encryption

- What is data at rest?
    - data at rest refers to data that is not "moving"
    - ex: information on your laptop is considered data at rest
- What does at-rest encryption provide?
    - added data protection, beyond file protection
    - disk level protection
        - password must be given at boot or mount time
        - driver or file system does encrypt/decrypt
        - protects computer against unauthorized access
    - file level protection
        - password must be given when file is opened
        - application or library does encrypt/decrypt
        - protects file against unauthorized access
    - Cryptography can be implemented to increase security/protection
    - Encrypted data should remain encrypted when access controls such as usernames and passwords fail

Symmetric Encryption

- What are the advantages and disadvantages of symmetric encryption
    - Advantages
        - privacy and authentication in one operation
        - relatively efficient/inexpensive algorithms

        - no central authentication services required
    - Disadvantages
        - scalability: establishing keys w/many partners
        - authentication: doesn't work w/new partners
        - privacy: shared secret is known by one-too-many
        - weakness: short keys are subject to brute force -> dictionary attacks
    - Symmetric-key algorithms use the same cryptographic keys for encryption of plaintext and decryption of ciphertext

Cryptographic Privacy

- how are cryptographic hashes used for tamper detection
    - checksums are often used to detect data corruption
    - cryptographic hashes are essentially strong checksums

WEEK 9

Lecture 16: Distributed Systems: Goals, Challenges and Approaches

Goals of Distributed Systems

- What are the goals of distributed systems?
    - **scalability and performance**
        - apps require more resources than one computer has
        - grow system capacity/bandwidth to meet demand
    - **improved reliability and availability**
        - 24x7 service despite disk/computer/software failures
    - **ease of use, with reduced operating expenses**
        - centralized management of all services and systems
        - buy (better) services rather than computer equipment
    - **enable new collaboration and business models**
        - collaborations that span system (or national) boundaries
        - a global free market for a wide range of new services
- How did the ideal of scalability and performance change with time?
    - We used to think that buying better components would optimize all avoidable overhead, get the OS to be as reliable as possible, and run extremely fast.
    - Now, we want to spread work over many nodes to improve performance and availability

RPC

- Can we assume that network connections are reliable?
    - Absolutely not. Although you may think that network connectivity becomes a given for commonly used applications, connectivity can be exploited and locations can heavily affect how the distributed system operates

- What happens if the timeout value for acknowledgement of receiving a message is too small? too large?
    - too small: the send will resend messages needlessly and waste CPU time on the send and network resources
    - too large: sender waits too long to resend and perceived performance at the sender is reduced
- What are the advantages and disadvantages of distributed shared memory
    - definition: systems enable processes on different machines to share a large, virtual address space -> so it looks like its a multithreaded application except that the threads run in different machines
    - Advantages
        - fast because no extra connectivity is required
        - Page can be local or on another machine -> can always be found
    - Disadvantages
        - Handling failure on a machine can be difficult -> what happens to the pages on that machine?
        - Bad performance due to expensive page faults from not being located in the same machine
- What are remote procedure calls?
    - definition of procedure calls: primary unit of computation in most languages, unit of information
    - a remote procedure call is a protocol that one program can use to request a service from a program located in another computer
    - We want to make the process of executing code on a remote machine as simple and straightforward as calling a local function
- Explain the steps for making a remote procedure call
    - **1) Create a message buffer:** a contiguous array of bytes of some size
    - **2) Pack the needed info into the message buffer:** remote function is actually executed through the RPC runtime calls into the function specified by the ID
    - **3) Send the message to the destination RPC server:** the communication with the RPC server, and all of the details required to make it operate correctly are handled by the RPC run-time library
    - **4) Wait for the reply:** because function call are usually synchronous, we can wait for completion
    - **5) Unpack return code and other arguments:** also known as unmarshaling/deserialization of the message
    - **6) Return to the caller:** return the client stub to the client code
- Explain the steps taken by a server when it receives a remote procedure call
    - **1) Unpack the message:** deserialization/unmarshaling
    - **2) Call into the actual function:** remote function is actually executed through the RPC runtime calls into the function specified by the ID
    - **3) Package the results:** the return arguments are marshalled back into a single reply buffer

    - **4) Send the reply:** the reply is sent back to the caller
- What are the key features of RPC
    - client application links against local procedures: calls local procedures gets results
    - all rpc implementation is within those procedures
    - client application does not actually know about the RPCs
        - does not know about format of messages
        - does not worry about sends, timeouts, resents
        - does not know about external data representation (XML vs binary ISA specific instructions)
    - all of the above is automatically generated by RPC tools
    - **the key tools is the interface specification**
    - **All rpc implementation is within local procedures**
- What are some challenges in executing RPCs
    - Some lower level network protocols **provide sender-side fragmentation** (of large packets into a set of smaller ones) **and receiver-side reassembly** -> if not, the RPC run-time may have to implement functionality to acknowledge requests
    - Handling endianness in machines -> some RPC packages handle this by providing a well-defined endianness within their message formats
    - Whether to expose asynchronous nature of communication to clients, thus enabling some performance optimizations
        - because waits for messages can be very long, some RPC packages enable you to invoke an RPC asynchronously
        - When issued, the RPC package sends the request and returns immediately -> client is free to do other work, such as calls to other RPCs

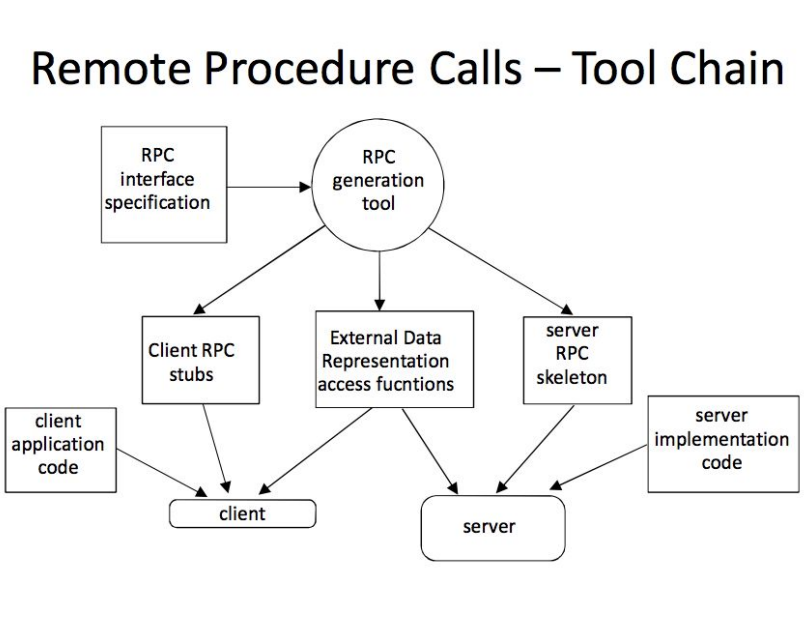
RPC Stubs

- What is an RPC stub
    - an RPC stub is a client generated stub that contains each of the functions specified in the interface
    - acts as a client's local representative/proxy for the remote object
    - a client stub is responsible for converting parameters passed between client and server during RPCs -> makes it look like a local function call
- What is the RPC generator's job?
    - The stub generator's job is to remove some of the pain of packing function arguments and results into messages by automating the process
    - inputs a set of calls a server wishes to export into the stub compiler
    - client program wishing to use this RPC would then link this client stub to their application

RPC skeleton

- What is an RPC skeleton?

    - The skeleton is a server side object that is responsible for dispatching the call to the actual remote object implementation
    - server-side recipient for API invocations
    - Responsible for deconversion of parameters passed by the client and conversion of the results after calling the function
- Draw the remote procedure call tool chain
    -



RESTful interface principles

- What does REST stand for?
    - representational state transfer
    - an architectural style for designing loosely coupled applications with HTTP
    - Does not enforce any rule regarding how it should be implemented at lower level -> just a high level design
- What are the architectural constraints of RESTful interface?
    - **Client-server architecture**: separating the user interface concerns from the data storage concerns improves the portability of the user interface across multiple platforms
        - Improves scalability by simplifying the server components
        - **The server and must be able to evolve separately without any dependency on each other. Clients should only know the resource URIs**
    - **Statelessness**: client-server communication is constrained by no client context being stored on the server between requests
        - Each request from any client contains all the information necessary to service the request
        - **The server will not store anything about latest HTTP request client made**

            - **Will treat each and every request as new . No session, no history**
    - **Cacheability: Caching shall be applied on resources when applicable** and then these resources MUST declare themselves as cacheable
        - Caching can be implemented on server or client side
        - Well managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance
    - **Layered System:** REST allows you to use a layered systems architecture where you deploy the APIs on server A, and store data on server B and authenticate requests in Server C
        - A client cannot ordinarily tell whether it is connected directly to the end server
    - **Code on demand (optional):** Most of the time, you will be sending the static representations of resources in form of XML or JSON
        - when you need to, you are free to return executable code to support a part of your application
            - e.g. clients may call your API to get a UI widget rendering code
    - **Uniform Interface:** you MUST decide APIs interface for resources inside system which are exposed to API consumers and follow religiously
        - A resource system should only have one logical URI, and then should provide ways to fetch related or additional data
        - Any single resource should not be too large and contain each and everything in its representation
        - All resources should be accessible through a common approach such as HTTP GET
            - “Once a developer become familiar with one of your API, he should be able to follow similar approach for other APIs”
        - **Resource identification in requests**: individual resources are identified as requests, for example using URIs in
        - **Resource manipulation through representations:** when a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource
        - **Self-descriptive messages**
            - Each message includes enough information to describe how to process the message -> ex: which parser to invoke
        - **Hypermedia as the engine of application state (HATEOAS)**
            - Having accessed an initial URI for the REST application -> analogous to a human Web user accessing the home page of a website -> a REST client should be able to use server-provided links dynamically to discover all other available actions and resources it needs

Marshal/Unmarshal

- Marshalling/unmarshalling is the process of packing/unpacking contents from the buffer after it is sent over a server
- Refers to the process of converting the data or the objects into a byte stream and unmarshaling is the reverse process of converting the byte-stream back to their original data or object
- Transforming the in-memory representation of an object into a suitable format for storage on transmission

Unforgeable capabilities

- How can we make capabilities truly unforgeable?
    - We can make sure that the capabilities are unforgeable through making sure that they stay in the operating system -> give the user an index to access capabilities such as a file descriptor being used an index into a per-process array
    - Hardware tags: Each word will have a 1-bit tag associated with it. The tag “on” means that programs cannot change or copy that particular word
    - Protected address space: Store capabilities in parts of memory that are inaccessible to programs
    - Language-based security: Use a programming language to enforce restrictions on access and modification to capabilities -> Java
    - Cryptography: take the capability and use encrypting

Distributed Systems Challenges

- Why are distributed systems hard to build?
    - new and more modes of failures
        - one node can crash while others continue running
        - occasional network messages may be delayed or lost
        - a switch failure may interrupt communication between some nodes, but not others
    - complexity and distributed state
        - within a single computer system, all system resources updates are correctly serialized
        - different nodes may consider a single resource in different states meaning that a resource is not in a single state but rather a vector of states
    - complexity of management
        - thousands of different system can have thousands of different systems that may be configured differently
        - even if we configure a distributed management service to push management and updates out to all nodes, some nodes may not be up when the updates are sent -> inconsistencies
    - Much higher loads
        - higher loads uncover weaknesses that have never been caused by lighter ones

        - bottlenecks: more nodes mean more messages -> increased overhead+delays
    - Heterogeneity
        - In a distributed system, each node may run a different ISA, OS, or versions of software

Deutsch's Seven Fallacies

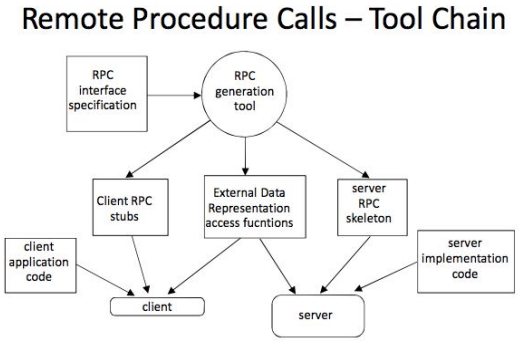
- What are Deutch's Seven Fallacies
    - Network is reliable
        - subroutine calls always happen and messages/responses aren't always guaranteed to be delivered
    - No latency (instant response time)
        - the time to make a subroutine call is negligible but messages can be 1000x slower
    - available bandwidth is infinite
        - network throughput is limited
    - network is secure
        - Once we put our computer in a network, it can be susceptible to man in the middle attacks
    - network topology and membership are stabled
        - routes change and new clients/servers appear and disappear continuously
    - network admin is complete and consistent
        - there may not be a single database of all known clients -> different systems have different admins
    - cost of transporting additional data is zero
        - network infrastructure is not free, and the capital and operational costs of equipment and channels to transport all of our data can be expensive
    - extra: homogeneity
        - nodes on the network are running different versions, different OS, ISA, etc.

RPC Interoperability

- What are some challenges in making RPC interoperable?
    - S/W, API, and protocols constantly evolve so we need to embrace new requirements and functionalities
    - A single node is running a single OS release so all s/w can be upgraded at same time as OS
      a distributed system is unlikely homogenous
        - rolling upgrades do one server at a time
        - newly added servers may be up/down-rev
        - we may not have control over client s/w versions
- How do we ensure interoperability?

    - 1) restricted evolution: all changes must be upward compatible
    - 2) compensation (runtime restriction): all sessions begin with version negotiation
    - 3) better tools that embrace polymorphism: every agent speaks his own protocol version, RPC language and tools are version arware, and equally applicable to message and at-rest data
- How do RPC packages handle endianness consistency?
    - RPC's usually have a well-defined endianness to their message format and will make the write conversions so that the machine endians match
    - RPC language and tools are always version aware

RPC Tool Chain

- Draw the RPC tool chain
    - 

XDR (External Data Representation)

- What is XDR?
    - **XDR is a standard data serialization format for uses such as computer network protocols**
    - Allows data to be transferred between different kinds of computer systems -> distributed systems
- What are some properties of XDR
    - XDR are upwards compatible serialized object formats
        - platform independent data representations
        - client-version sensitive translation
            - old clients never see new-version fields
            - new clients infer upwards compatible defaults
- What is an example of a XDR
    - Google Protocol buffers
        - efficient translation
        - applicable to both protocols and persisted data
        - supports many representations (e.g. binary, json)
        - has adaptors for many languages

Asymmetric Encryption

- Compare and Contrast the key advantages and disadvantages of asymmetric and symmetric encryption
    - Definition of asymmetric encryption: using two different keys (public and private) to encrypt and decrypt a message
    - symmetric
        - advantages
            - symmetric encryption tends to be faster
            - encrypted data can be transferred on the link even if there is a possibility of being intercepted
            - use password authentication to prove user identity
            - system which possesses the secret key can decrypt a message
        - disadvantages
            - symmetric cryptosystems have a problem of key transportation
                - secret key is to be transmitted to the receiving system before the actual message is to be transmitted
            - someone may be able to receive the key before the message
            - cannot provide digital signatures that cannot be repudiated
    - asymmetric
        - advantages
            - no need for exchanging keys, thus eliminating the key distribution problem
            - increased security -> private keys do not ever need to be transmitted or revealed to anyone
        - disadvantages
            - speed -> can be faster in a hybrid approach
- What are the advantages and disadvantages of encrypting a message with a private key?
    - advantages
        - only you could have encrypted it, it must be from you
        - it has not been tampered with since you wrote it
    - disadvantages
        - if you use a key too much someone will crack it
        - no need to encrypt whole message with private key
        - compute a cryptographic hash of your message and then encrypt that with your private key instead

Digital Signatures

- What is the purpose of a digital signature?
    - uniquely identifies the document signer, identifies the document that was signed, and cannot be copied onto another document
    - we know the document has not been tampered with because we can recompute the cryptographic hash at anytime and confirm it matches message the sender signed

    - Makes sure that the receiver knowns the sender's identity and that the message arrived intact
    - Encrypting a message with a private key signs it -> digital signature
- How do you create a digital signature
    - signing software creates a one-way hash of the electronic data to be signed. The private key is then used to encrypt the hash -> **encrypted hash along with other information such as the private key and the hashing algorithm is the digital signature**
    - the signed message (digital signature) is decrypted with a public key into a hash value

Public Key Certificate

- What is a PK certificate?
    - Essentially a data structure
        - name and description of an actor
        - public key belonging to that actor
        - validity/expiration information
    - Signed by someone I trust
        - whose public key i already have
        - a digital Notary Public
    - Testifying that the actor owns the public key and the private key
        - electronic document used to prove the ownership of a public key
- When can we use public key certificates?
    - if I know public key of the authority who signed it
        - i can validate the signature is correct
        - i can tell the certificate has not been tampered with
    - if I trust the authority who signed the certificate
        - I can trust they authenticated the certificate owner
        - e.g. we trust drivers licenses and passports
- What is the chicken and egg problem presented by PK certificates?
    - We'll learn about a company's public key by getting a certificate for it
    - is signed by a third party
    - we'll check the signature by knowing the third party's public key
        - However, where do we get the third party's public key?
- Analogy for PK certificates
    - work much the same way as obtaining your driver's license from the DMV
        - verify your identity and other information about you
    - Certificate authorities are entities that validate identity and issue certificates
    - Clients and servers use certificates issued by the CA to determine the other certificates that can be trusted.
        - Before issuing a certificate, the CA must use its published verification procedures for the type of certificate

    - Certificate issued by the CA binds a particular public key to the name of the entity that the certificate identifies. (ex: name of an employee/server)
        - Only the public key that is certified by the certificate will work with the corresponding private key that is owned by the entity identified with the certificate
        - Most importantly, a certificate always includes the digital signature of the issuing CA
- What is the purpose of the PK certificate?
    - to distribute the public key
- Steps in creating a certificate
    - Anyone who gets a copy of the bits has a key, but we don't know who the key belongs to
    - we have a third party create a signed bunch of bits known as the certificate
        - contains information about the party that owns the public key, the public key itself, and other information including the expiration date
        - entire set of information is run through a cryptographic hash and is signed using the third party's private key -> digitally signs the certificate
    - After obtaining the certificate and checking it, the customer has the public key he needs and can simply store and use it after that point
- When do you want to use symmetric encryption vs asymmetric and vice versa?
    - symmetric encryption should be used for communication because we can use short-lived, disposable, session keys + much less expensive
    - asymmetric encryption should be used only for public key distribution/sharing and for initial authentication/validation due to its complexity and performance

Lecture 17: Remote Data, Synchronization, Security

Local vs Cloud

- What is the difference between local and cloud storage (advantages and disadvantages)
    - Cloud
        - advantages
            - maintenance is not an issue for the client because the server handles all of the transactions
            - scalability -> can ask your storage provider for more space
        - disadvantages
            - speed -> lots of data to store + network connectivity
            - lack of control on a public cloud
    - Local
        - advantages
            - speed
            - storing data on external hard drives is faster than the cloud

                - full control of your backups -> better control of who accesses your data
            - disadvantages
                - creating and maintaining a local storage system is expensive
                - can be expensive as well
- What are the types of remote file transfers?
    - explicit commands to copy remote files
        - OS specific, scp(), rsync, S3 tools
        - IETF protocols: FTP, SFTP
    - implicit remote data transfers
        - browser transfers (lies in HTTP)
        - email clients (move files with IMAP/POP/SMTP)
- What are the advantages and disadvantages of remote file transfer?
    - advantages
        - efficient, requires no OS support
    - disadvantages
        - latency, lack of transparency
- What is the purpose of remote data access?
    - to make remote files appear to be local
        - remote disk access (e.g. Storage Area Network)
        - remote file access (e.g. Network Attached Storage)
        - distributed file systems (NAS on steroids)
- what are the advantages and disadvantages of remote data access
    - advantages
        - transparency, availability, throughput
        - scalability, cost (capital and operational)
    - disadvantages
        - complexity, issues with shared access
- What is the goal of remote disk access?
    - complete transparency
        - normal file system calls work on remote files
        - all programs “just work” with remote files
    - Typical Architecture
        - Storage Area Network
            - very fast, very expensive, moderately scalable
        - ISCI
            - client driver turns reads/writes into network requests
            - server daemon receives/serves requests
            - moderate performance, inexpensive, high scalable
- What are the advantages and disadvantages remote disk access?
    - advantages
        - provides excellent transparency
        - decouples client hardware from storage capacity

        - performance/reliability/availability per backend
    - disadvantages
        - **inefficient fixed partition space allocation**
        - can't support file sharing by multiple client systems
        - message losses can cause file system errors
    - this is THE model for virtual machines
- What are the goals of remote data access
    - transparency
        - **indistinguishable from local files for all uses**
        - **all clients see all files from anywhere**
    - performance
        - **per-client: at least as fast as local disk**
        - **scalability: unaffected by the number of clients**
    - Cost
        - **capital: less than local (per client) disk storage**
        - **operational: zero, it requires no administration**
    - Capacity: unlimited, it is never full
    - Availability: 100%, no failures or downtimes
- What are the challenges of remote data access?
    - transparency
        - despite Deutsch's warnings
        - creating global file namespaces
    - security
        - despite insecure networks and heterogeneous systems
    - preserving ACID semantics, POSIX consistency
        - despite lack of shared memory and atomic instructions
    - performance
        - despite everything being done with messages
    - reliability and scalability
        - despite having more parts and modes of failures
- what are the goals of remote file access?
    - complete transparency
        - normal system calls work on remote files
        - support file sharing by multiple clients
        - performance, availability, reliability, and scalability
- What are the advantages and disadvantages of remote file access?
    - advantages
        - very good application level transparency
        - very good functional encapsulation
        - able to support multi-client file sharing
        - potential for good performance and robustness
    - disadvantages
        - at least part of implementation must be in the OS

        - client and server sides tend to be fairly complex
            - THE model for client/server storage

Client/Server Model

- What are the goals of a client/server model
    - peer-to-peer
        - Most systems have resources (e.g. disks, printers)
        - they cooperate/share with one another
    - thin client
        - few local resources (e.g. CPU, NIC, display)
        - most resources on work-group or domain services
    - cloud services
        - **clients access services rather than resources**
        - clients do not see individual servers

Distributed File Systems

- NFS
    - How do we define a stateless file protocol?
        - example: when we ask the server to read a few blocks from a certain fd, then the fd is a piece of shared state between the client and server
        - the client cannot save the context/state of a system on the server -> refer back to REST protocols
        - *** stateless protocols make fail-over easy -> recovery is a lot easier because client supplies necessary context w/ each request
    - Explain what a file handle is
        - Essentially a data structure used to uniquely describe the file/directory a particular operation is going to act upon
    - What are the three essential pieces of a file handle?
        - **Volume identifier, inode number, generation number**
        - volume identifier informs the server which file system it refers to
        - inode number tells the server which file within that partition it is accessing
        - generation number: needed when reusing an inode number -> increment whenever an inode number is reused, the server ensures the client with an old file handle can't accidentally access the newly-allocated file
    - How does the client handle server failure with idempotent operations?
        - because the operations are idempotent, trying it multiple times would be the same as trying it once, meaning that the client can simply retry the request
    - Is a write request idempotent?
        - Yes because it contains the exact data to be written, the count, and the offset of the data to be written to

- How does the NFS-client side file system use client-side caching to improve performance?
    - The client can cache file data and metadata that it has read from the server in client memory
        - first access can be expensive but subsequent accesses can be cheap because it is stored in client memory and does not need to retrieve data from the server
    - The cache can also serve as a temporary buffer for writes
        - when a client application first writes to a file, the client buffers the data in client memory before writing the data out to the server
        - this means that the application's calls to write succeed immediately because it will simply be stored in the cache
- What are the major issues we face when dealing with cache consistency?
    - update visibility
        - when do updates from one client become visible to others
    - stale cache
        - clients can get stale versions of the file/old versions
    - example of update visibility
        - client C2 may buffer writes in its cache for a time before propagating back to the server
        - while F(v2) sits in C2's memory, any access of F from another client will fetch the old version
    - stale cache
        - C2 has finally finished its writes to the fill server and thus the new version of the file is located in memory for that particular machine
- Explain flush-on-close (a.k.a close-to-open)
    - when a file is written to and subsequently closed by a client application, the client flushes all updates to the server
        - ensures that a subsequent read/another node/client will see the latest version
        - flush-on-close alone does not help to ensure close-to-open consistency
- How do we address the stale cache problem?
    - checks to see whether the file has changed before using its cached contents
    - when opening a file, the client-side system will issue a GETATTR request to the server to fetch the file's attributes
    - the attributes include information as to when the file was last modified on the server
    - if the time of modification was more recent than the one on the server, it invalidates the one on the server and removes it from the client cache
- What problems arose when the stale cache problem was addressed?
    - the NFS server became flooded with GETATTR requests

        - created a attribute cache to remedy this situation -> client would validate a file before accessing it, but would most often use the attribute cache to fetch its attributes
    - Assess (advantages and disadvantages) NFS cache consistency
        - flush-on-close was created to make sense but introduced performance problems
            - if a temporary or short-lived file was created on a client and then soon deleted, it will still be forced onto the server
            - attribute cache can still give you an undesired version

Man in the Middle Attacks

- How can we ensure better network security through considering man in the middle attacks?
    - We can assume that someone is watching all network traffic
        - network traffic is routed through many machines
        - most internet traffic is not encrypted
        - snooping utilities are widely available
        - passwords may be sent in clear text
    - Assume someone can forge messages from you
        - your traffic is being routed through many machines
        - some of them may be owned by bad people
        - they can hijack connection after you log in
        - can replay previous messages and forge new ones

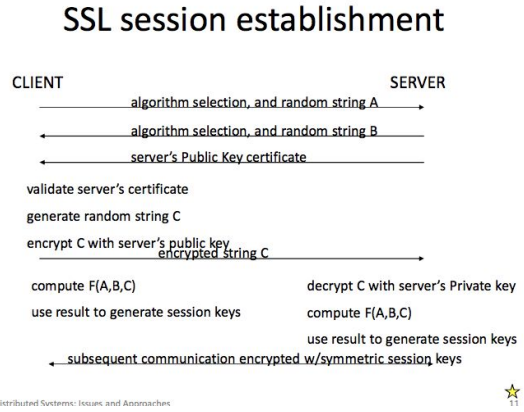
Remote File System Goals

- What are the goals of remote file systems
    - **transparency**
        - indistinguishable from local files for all uses
        - all clients see all files from anywhere
    - **Performance**
        - per-client: at least as fast as local disk
        - scalability: unaffected by the number of clients
    - **Cost**
        - capital: less than local disk storage
        - operational: zero, it requires no administration
    - **Capacity**
        - **unlimited, never is full**
    - **Availability**
        - **always available/never fails**

Reliable Communication

- What are the goals of network security
    - secure conversations

        - privacy: only you and your partner know what is said
        - integrity: nobody can tamper with your messages
    - positive identification of both parties
        - authentication of the identity of message sender
        - assurance that a message is not a replay or forgery
        - non-repudiation: he cannot claim “i didn’t say that”
    - they must be assured in an insecure environment
        - messages are exchanged over public networks
        - messages are filtered through private computers
- How can we use both symmetric and asymmetric encryption to establish secure connection?
    - Use asymmetric to start the session
        - RSA or other PK mechanism
        - authenticate the parties
        - securely establish initial session key
    - Use symmetric key for encryption of the session (much less expensive to encrypt/decrypt + efficiency + short-lived/disposable)
        - e.g. DES or AES
        - very efficient algorithm based on negotiated key
    - Periodically move to new session key
        - e.g. sequence based on initial session key
        - e.g. switch to new key message
- Draw a diagram of an SSL session establishment



    - Establishes secure two-way connection because nobody can snoop (privacy) and nobody can generate
    - certificate based authentication of server so that client knows who he’s talking to
    - optional certificate based authentication of client -> non-repudiation
    - Use PK to negotiate symmetric session keys
        - safety of public key, efficiency of symmetric

Distributed Locks

- Easy to send all locking resources and request and messages to a central who will implement them with local locks
- Opportunistic Locks
    - How and why are opportunistic locks used?
        - Opportunistic locks are used because contention is rare for most resources, and the locking code is only there to ensure correct behavior in unlikely situations
        - A requestor can therefore ask for a long term lease to enable all future locking as a purely local operation
            - if another node requests access to the resource, the lockmanager will notify the op-lock owner that the lease has been revoked -> subsequent locking operations will have to be dealt with through the centralized lock manager

Leases

- Primary differences between a lock and a lease
    - a lock grants ownership/access of the resource until the owner releases it
    - a leases grants access of the resource until the owner releases it/time expires
    - Leases are enforced
        - when a request is sent to a remote server to perform some operation, that request includes a copy of the requestor's lease -> if the lease has expired, the responding resource manager will refuse to accept the request
        -
- What are some problems with leases?
    - if a tardy owner was part-way through a multi-update transaction and the lease expires -> the operation should be done atomically -> meaning that it needs to fallback to its previous state
    - if a lease period is too short, the owner may have to renew it many times
    - if a lease period is too long, system maky take a long time to recover from a failure
    - Correct recovery from lock-server failures are complex
    - Lease expiration duration without a universal time standard
- Evaluate Leases
    - Mutual Exclusion/Correctness: leases are as good as locks + additional enforcement
    - Fairness: Depends on the policies implemented by the remote lock manager, who could easily implement either queued or prioritized requests
    - Performance: remote operations are very expensive but we aren't going to network leases for local objects -> can be very efficient
    - Progress: automatic preemption makes it leases immune to deadlocks
    - Robustness: Clearly more robust than single-system mechanisms
- How do you revoke an expired lease?

    - lease cookie includes a “good until” time and any operation fails involving a “stale cookie”

Distributed Transactions

- How do multi-node transactions introduce new failure modes?
    - one node can see a commit, but another node may not be able to
    - after recovery, different journals may not agree

Distributed Consensus

- What are the properties that must be satisfied in a consensus protocol?
    - **Termination:** every correct process decides some value
    - **Integrity:** every correct process decides at most one value, and if it decides some value v, then **must have been proposed by some process**
        - **if a correct process decides v, then v must have been proposed by some correct process**
    - **Validity:** if all processes propose the same value v, then all correct processes decide v
    - **Agreement:** Every correct process must agree on the same value
- Is consensus impossible in asynchronous systems?
    - No, it is not impossible. It merely means that under the models assumptions, no algorithm can always reach consensus in bounded time, meaning that it is highly unlikely to occur

Two Phase Commit

- What are the phases of 2PC
    - **1) Commit-request phase:** coordinator processes attempts to prepare all the transaction’s participating processes to take the necessary steps for committing or aborting the transaction
        - a) coordinator sends a query to commit message to all cohorts and waits until it has received a reply from all cohorts
        - b) cohorts execute the transaction up until the point where they are asked to commit -> write each entry in their undo and redo logs
        - c) Each cohort replies with an agreement message if the cohorts actions succeed, or an abort message, if the cohort experiences a failure
    - **2) Commit phase:** based on voting of the cohorts, the coordinator decides whether to commit or abort the transaction
        - Success
            - 1) coordinator sends a commit to all the cohorts
            - 2) each cohort sends an acknowledgment to the coordinator
            - 3) The coordinator completes the transaction when all acks have been received
        - Failure
            - 1) coordinator sends a rollback message to all the cohorts

        - 2) each cohort undoes the transaction using the undo log, and releases the resources and locks held during the transaction
        - 3) Each cohort sends an acknowledgment in the coordinator
        - 4) The coordinator undoes the transaction when all acks have been received
- What are the disadvantages/limitations of 2PC
    - **is a blocking protocol**
        - if a coordinator fails permanently, some cohorts will never resolve their transactions
        - After a cohort has sent an agreement message to the coordinator, it will block until a commit or rollback is received
        - Can be subject to unbounded delays because the cohort is blocked if coord fails after they ack
            - coord cannot recover w/o entire cohort present

Three Phase Commit

- 3PC is non-blocking and places an upper-bound on the amount of time required before a transaction either commits or aborts
- Explain the actions of the coordinator and the cohort
    - 1) the coordinator receives a transaction request. If there is a failure, the coordinator aborts the transaction -> otherwise sends canCommit? message to the cohords and moves **to the waiting state**
    - 2) If there is a failure, timeout, or if the coordinator receives a No message in the waiting state, the coordinator aborts the transaction and notifies the cohorts. Otherwise the coordinator will receive yes messages and then send preCommit messages to all cohorts and moved th**e prepared state**
    - 3) If the coordinator succeeds in the prepared state, it will move to the commit state -> aborts transaction
        - in the case where an acknowledgement is received from the majority of cohorts, it will move to the **commit state**
    - 1) The cohort receives a canCommit? message from the coordinator -> sends a Yes message to the coordinator and moves to the prepared state or sends a No message and move to the abort stage
    - 2) In the prepared state, if the cohort receives an abort message from the coordinator, fails, or times out, it aborts. If the cohort receives a preCommit message, it sends an ACK messages and awaits a final commit/abort
    - 3) If, after a cohort member receives a preCommit message, the coordinator fails or times out, the cohort member goes forward with the commit
- What are the limitations of 3PC?
    - Cannot recover in the event the network is segmented in any manner
        - cannot tolerate network partitioning failure
        - protocol requires at least three round trips to complete

Work Tickets

- What are work tickets?
    - Authentication ticket that describes the actor, the permitted operation, and any other relevant information that is usually signed/encrypted with a private key
- What are the disadvantages of work tickets?
    - they are transferable and therefore stealable
- Are work tickets unforgeable?
    - Yes they are unforgeable because signature prevents random agents from creating work tickets
- How can we determine that the ticket can only be used by the authorized agent/actor?
    - We could use a public key of the authorized agent in the ticket, but this would get the server back into the authentication business -> slow (asymmetric)
    - Common approach: Have the authentication server generate a session, and encrypt a copy for the client with the client's public key, and a copy for the server with the server's public key
- What is the purpose of Kerberos style work tickets
    - enables a private client/server session
    - enables a mutually authenticated client-server session

Distributed Authentication/authorization

- Why are PK certificates impractical in a distributed system?
    - may not be practical to require all actors to have registered public/private key pairs, or for all servers to use them as the basis for authentication
    - still need to access and interpret a large access control database to determine whether they are allowed to perform certain operations
    - Therefore, we can use an authentication server to keep track of information about all of the authorized users and what each is authorized to do
- Explain the steps of authenticating through an authentication server
    - An actor contacts an authentication server, and describes the actions to be performed
    - the authentication server both authenticates the actor and determines whether or not that actor was allowed to do the proposed operation
    - authentication server creates a work ticket describing the actor, permitted operation, and any other relevant information ,etc.
    - actor would present the work ticket to the server along with the request
- Describe the distributed authorization process for both access control lists and capabilities
    - access control lists
        - authentication service returns credentials -> server checks against access control list
        - advantage: auth service doesn't know about the ACLs
    - capabilities
        - authentication service returns capabilities -> which server can verify

        - advantage: the server does not have knowledge about the clients

Peer-to-Peer Security

- what is peer-to-peer security in client-side authentication/authorization?
    - peer-to-peer security is the security of interactions between a server and a client when all users are known to all systems
    - all systems are trusted to enforce the access control -> such as basic NFS
- What are the advantages and disadvantages of client-side authentication/authorization?
    - advantages
        - simple implementation
    - limitations
        - assumes all clients can be trusted
        - assumes all users are known to all systems

Immutability

- What is an immutable object?
    - an unchangeable object whose state cannot be modified after it is created
- What are the benefits of immutability in distributed systems?
    - If data is mutable,we need to settle for eventual consistency while rebalancing
    - With immutable data we don't need to worry about stale versions present in different nodes because it a node has some information, that's the only value that it will ever assume

Reliability

- How do we accept the fact that failure in distributed systems is inevitable?
    - We must build robust systems that have additional capacity to survive failures
    - We must have automatic failure detection
    - dynamically adapt to the new reality
    - continue service, despite component failures
- What is reliability?
    - reliability refers to the probability of not losing data
        - for example, disk/server failures do not result in data loss (RAID mirroring, parity, erasure coding, etc.) -> we can keep copies on multiple servers
    - automatic recovery after a failure

Availability

- What is a availability?
    - the fraction of time the service is available
    - disk/server failures do not impact data availability
        - backup servers with automatic fail-over
    - automatic recovery after rejoin
- Explain what fail-over is

    - recovering from failures through backup servers when a service becomes unavailable
- What are the goals of fail-over
    - data must be mirrored to a secondary server
    - failure of primary server must be detected
    - client must be failed-over to secondary
    - session state must be reestablished
        - client authentication/credentials
        - session parameters (e.g. working directory, offset)
    - in progress operations must be retransmitted to preserve ACID semantics -> atomicity
        - client must expect timeouts, retransmit requests
        - client responsible for writes until server acks
- What are the goals of failure detect/rebind
    - We want to have a client drive recovery where
        - client detects server failure, reconnects to the server, and reestablishes the session
    - We want to have transparent failure recovery
        - system detects server failure through health monitoring
        - successor assumes primary's IP address
        - state reestablishment
            - successor recovers last primary state check-point
            - stateless protocol
- What is the difference between a stateless and a stateful protocol?
    - a stateful protocol like TCP
        - operations occur within a context
        - each operation depends on previous operations
        - successor server must remember session state
    - a stateless protocol like HTTP
        - client supplies necessary context w/each request
        - each operation is complete and unambiguous
        - successor server has no memory of previous events
        - stateless protocols make fail-over easy because restoration can be done simply through retrying the operation
- How are idempotent operations used to ensure availability?
    - because idempotent operations can be repeated many times to the same effect, if the client does not get a response, we can simply retry and send the request again
        - if server gets multiple requests, then no harm is done
        - is very powerful because it works for server failure, lost request, and lost response -> however, no ack does not mean that it didn't happen
- Does idempotency solve multi-writer races?

    - No because the competing writers must serialize their updates and cannot be trusted to maintain lock state
- Should a distributed lock manager be stateful or stateless?
    - distributed lock managers should be stateful because recovery can be extremely complex

Idempotent Operations

- What are the characteristics of an idempotent operation
    - Executing the operation once is the same as executing numerous times

- What are some examples of operations that are idempotent and not idempotent
    - Idempotent
        - read, write, delete
    - Non-idempotent
        - read next block of current file
        - append contents to the end of a file

ACID Consistency

- What does ACID stand for and describe each characteristic
    - Atomicity, Consistency, Isolation, Durability
    - Atomicity
        - all transactions should be executed atomically, meaning that they should be executed in an all or nothing fashion
        - if one part of the transaction fails, the entire transaction should fail
    - Consistency
        - ensures that one transaction will bring the database/server from one valid state to another
        - This does not guarantee correctness of the transaction but it merely means that any programming errors cannot violate any of the defined rules
            - transactions must meet all validation rules/conditions
            - if consistency is not met, it will affect the atomicity of the operation as the transaction will not be completed if a certain validitity condition is untrue
    - Isolation
        - Ensures that the concurrent execution of transactions results in a system state that would be obtained if the transaction was executed sequentially
    - Durability
        - Ensures that once a transaction is committed, it will remain in persistent storage
        - transactions must be recorded in non-volatile memory so that it is not destroyed
- Explain what a write-write failure is

    - Two transactions attempt to write to the same data field -> eliminates the effects of one -> revert to the last known good state

Read-after-write consistency

- What does AFS do to ensure read-after-write consistency?
    - AFS employs a last write wins policy where the last write becomes the version of the file on the server
    - this can be done because AFS flushes out the entire file instead of certain blocks
    - This is one advantage that NFS has compared to AFS because our final file may end up being a mixture of both files when we flush out individual blocks to the server

Close-on-open consistency

- Does flush-on-close ensure close-on-open consistency?
    - no it does not because we still have the stale cache problem which needs to be solved through making GETATTR requests and filling the attribute cache
- Explain close-to-open consistency issue on NFS
    - traditional NFS has write-through caching meaning that when a file is closed, all modified blocks are sent to the server
    - This means that the close() function does not return until each byte in each block is safely stored
    - Problem: if two clients on a network are writing to a cached copy of the same file and issue a close command in close proximity, NFS can't guarantee whose modified data will make it to the master copy on the server
- How does NFS solve this issue?
    - Close-to-open is solved through using an attribute cache + prevents two or more clients from modifying the same file at the same time by making sure the client executes a query to the server before modifying it -> GETATTR
    - Remember update visibility and stale cache issues

Classes of Distributed Systems

- Rate AFS vs NFS in terms of design centers, performance, ease of use
    - Design centers
        - both designed for continuous client/server connections
        - NFS supports diskless clients w/o local file systems
    - performance
        - AFS generates much less network traffic, server load
        - yield similar client response time
    - ease of use
        - NFS provides for better transparency
        - NFS has enforced locking and limited fail-over
        - NFS requires more support in the OS
    - Extra

        - scalability
            - scales very well because all file access by user/application is local, updating and checking is relatively cheap, both fetch and update propagation are very efficient
        - robustness
            - no server fail-overs but have local copies of most files
        - transparency
            - mostly perfect -> all file access operations are local
            - pray that we don't have any update conflict
- How are local copies made in AFS (replication) / explain replication in AFS
    - We first check for local copies in cache at open time and if none exists, we fetch it from the server and cache the file on local disk
    - if local copy exists, see if it is still up-to-date
        - compare file size and modification time with server
    - optimizations reduce overhead of checking
        - subscribe/broadcast change notifications
        - **time-to-live** on cached file attributes and comments
- Explain the server side logic of AFS
    - updates sent to server when local copy is closed
        - warns them to invalidate their local copy
        - warns them of potential write conflicts
    - server supports only **advisory file locking**
        - distributed file locking is complex
    - **clients are expected to handle conflicts**
        - noticing updates to files open for write access
        - notification/reconciliation strategy is unspecified

Graceful Degradation

Write Latency

- Compare the write latency of NFS and AFS
    - both perform similarly on new sequential writes to the server
    - However, AFS performs worse on a sequential file overwrite
        - the client first fetches the old file, and then modifies its contents
        - NFS will simply overwrite blocks and thus avoid initial useless fetch from server
    - Overall, workloads that only access a small subset of data perform better on NFS because NFS is a block based protocol that performs I/O proportional to the size of the read/write, while AFS fetches the entire file even if the amount of data being accessed is little

Cache Consistency

- Explain what we need to achieve cache consistency in AFS on different machines and on the same machine
    - Same machine
        - Write to the files are immediately visible to other local processes because a process does not have to wait until a file is closed to see its latest updates -> cached in local disk
        - Makes using a single machine behave exactly as you would expect, based upon typical UNIX semantics
    - Different machines
        - AFS makes updates visible at the server and invalidates cached copies at the exact same time
            - when a file is closed, the new file is flushed back to the server and thus is visible
            - the server breaks callbacks for any clients with cached copies by informing it that the callback it has on the file no longer exists
            - subsequent calls on the modified file will require steps to reestablish callbacks

APIs -> Protocols

- What are the goals of APIs
    - provide transparency
    - main question to be asked: how do users and processes access remote files?
        - how closely do remote files mimic local files (transparency)
- What are the goals of protocol
    - partition work: what messages exchanged, who does what work

Replication and Recovery

- What are the performance challenges in making distributed systems?
    - single client response time
        - remote requests involve messages and delays
        - error detection/recovery further reduces efficiency
    - aggregate bandwidth
        - each client puts message processing load on server
        - each client puts disk throughput load on server
        - each message loads server NIC and network
        - striping files across multiple servers provides scalable throughput
    - WAN scale operation
        - where bandwidth is limited and latency is high
    - aggregate capacity
        - how to transparently grow existing file systems
- What are the costs of mirroring data for replication and recovery?
    - mirroring by primary
        - primary becomes throughput bottleneck

        - replication traffic on backside network
    - mirroring by client
        - data flows directly from client to storage servers
        - replication traffic goes through client
        - parity/erasure code computation on client CPU
- How do we improve the mean time to repair
    - graceful degradation: degradation of service may persist longer
        - restoring lost redundancy may take a while
        - heavily loading servers, disks, and networks
- What are the steps in MTTR (mean time to repair)
    - primary failure detected
    - secondary promoted to primary role (assumes IP address) recent/in-progress operations recovered
    - clients learn of change and rebind
    - session state if any has been reestablished

Distributed Data Striping

- How does distributed data striping improve performance?
    - We can partition the work so that we can send as few messages as possible
    - we can use client-side caching to eliminate read requests
    - aggregation for fewer/larger write requests
    - By partitioning work, we can do as much as possible on the client and do as much as possible on the single server while eliminating multi-node communication
    - Increases throughput
- What are the benefits of direct data path? Explain the architecture, throughput, latency and scalability
    - architecture
        - primary tells clients where and which data resides
        - client communicates directly w/storage servers
    - throughput
        - data is striped across multiple storage servers
    - latency
        - no immediate relay through primary server
    - scalability
        - fewer messages on network
        - much less data flowing through primary servers

Scalable Distributed Systems

- How do scalable distributed systems minimize messages/client/second
    - cache results to eliminate requests entirely
    - enable enable complex operations w/single request
    - buffer up large writes on write back cache

    - pre-fetch large reads into local cache
- What are some bottlenecks in scalability
    - separated data and control-planes
        - control nodes choreograph the flow of data
            - where data should be stored or obtained from
            - ensuring coherency and correct serialization
        - data flows directly from producer to consumer
            - data paths are optimized for throughput/efficiency
    - **dynamic repartitioning of responsibilities**
        - in response to failures/load changes
- AFS
    - Provides good scalability due to client-side caching and handling files locally
        - client side performance can often come close to local performance, simulating local file access
    - What are some other improvements that AFS made?
        - true global namespace -> ensures that all files were named the same way on all machines
        - AFS incorporates mechanisms to authenticate users and ensure that a set of files could be kept private if a user so desired
        - AFS includes facilities for flexible user-manage access control
            - a user has a great deal of control over who exactly can access the files
            - NFS, like most UNIX file systems have much less support for this type of sharing
- Compare the differences of performance between NFS and AFS
    - Large file sequential re-read
        - AFS has large local disk cache so the access of the file will be local if cached
        - NFS only caches blocks in client memory so we would need to refetch the entire file from the remote server
    - Sequential writes of new files perform similarly
    - AFS performs worse on sequential file overwrite
        - the client first fetches the old file in its entirely, only to overwrite
        - NFS will simply overwrite blocks
- Explain the failure and rejoin complication
    - a file server goes down and then comes back up and reports for work, but needs to get all the updates that he missed
    - We need to know what updates he missed
        - we can compare all of his files to ours, but an exhaustive search like that could possibly be very inefficient
        - we can keep a log of all recent updates
            - but we have to know which one he already has
            - maybe have versioned updates

- Explain the Split-Brain complication
    - Suppose we had a network failure that partitioned our file servers
        - each half tried to take over for the other
        - each half processed different write operations
    - we can reconcile the changes by merging updated versions of different files
        - but how about files that were changed in both halves?
    - quorum rules can prevent dueling servers
        - servers that can't make quorum are read only
- Explain the disconnected operation complication
    - consider a notebook and a file server
        - I synchronize my notebook with the file server
        - i go away on a trip and update many files
        - others may change the same file on the server
    - How can we identify all of the changes
        - intercept and log all changes
        - differential analysis vs a baseline such as rsync
    - How can we correctly reconcile conflicts?
        - perhaps some can be handled automatically
        - some may require manual human resolution

Client-side caching

- Why is multi-writer distributed caching so difficult?
    - **time to live is a cute idea that doesn't work**
    - constant validity checks defeat the purpose
    - **one-writer at a time is too restrictive for most filesystems**
    - change notifications are a reasonable alternative
- What are the advantages of client-side caching
    - eliminate waits for remote read requests
    - reduces network traffic
    - reduces per-client load on server