Potential Questions

Lecture 1

Lecture 2

- Why is interface stability important?
    - Work is simplified for programmers when building software
    - Program is more portable
    - Training time for new programmers is reduced
    - Implementation can be changed, but still accomplishes same task. That way we can add efficiency or better implementation and the user doesn't have to change what she calls
- What is upwards-compatibility?
    - Old programs will still work the same way, but new interfaces will enable new software to access new functionality. I.e add new features but maintain backwards compatibility
    - Basically a system that is compatible with a future version (ie, interface) of itself. An example is a mobile app that silently passes over image formats it cannot currently handle but will support in the future
- What are two ways of adding upwards-compatible extensions / maintaining backwards compatibility?
    - Interface Polymorphism - Having different versions of the same method. (with different parameter/return types).
    - Versioned Interfaces - We can change interfaces, as long as we provide the old interface to old clients. Applications can call out which version of an interface they require.
- What are some ways to design a system with interface stability in mind?
    - Rearrange the distribution of functionality between components to create a simpler (more preservable) interface between them
    - Design features that we may never implement, so that the interfaces will accommodate them if we ever decide to build them
    - Introduce a large amount of abstraction in our services to ensure that they leave us enough slack for future changes
- What are some benefits of interface standardization
    - Details debated by many experts over long periods of time
    - Contributors work for different orgs with different strengths, so the standards will likely be platform-neutral.

    - Tend to have very clear and complete specs
    - Give tech suppliers freedom to explore alternative implementations (to improve reliability,performance,portability,mantainability)
- What are some disadvantages of interface standardization
    - Still constrain range of possible implementations. A far better implementation may be possible with a slight change in the interface.
    - Constrains customers who have applications that use non-standard features. I.e windows users didn't want to upgrade because applications would stop working
    - Becomes difficult to evolve standards when many people depend on it. Too many competing opinions.
- What should an API specification include?
    - A list of included routines/methods, and their signatures (parameters, return types)
    - A list of associated macros, data types, and data structures
    - A discussion of the semantics of each operation, and how it should be used
    - A discussion of all of the options, what each does, and how it should be used
    - A discussion of return values and possible errors
- What should an ABI specification include?
    - The binary representation of key data types
    - The instructions to call to and return from a subroutine
    - Stack-frame structure and respective responsibilities of the caller and callee
    - How parameters are passed and return values are exchanged
    - Register conventions (which are used for what, which must be saved and restored)
    - The analogous conventions for system calls, and signal deliveries
    - The formats of load modules, shared objects, and dynamically loaded libraries
- Not a question, but useful note on ABI
    - ABI compliant system means that one compiled binary executable can work on any system that complies with an ABI. WOW PORTABILITY > 9000 (vvv good)
    - Really coool. Say there's an ABI for MIPS systems. Then you know that if your machine code follows this ABI then you can run on any MIPS system int he world
    - Different ISA's correspond to different ABIs
- What are the types of OS Services + examples from each type?
    - Hardware Abstractions
        - CPU/Memory abstractions like processes,threads,virtual address space,signals
        - Persistent Storage abstractions like files/file systems,databases,object stores
        - Other I/O abstractions
            - Virtual terminal sessions, windows,sockets,pipes
    - Higher level abstractions
        - Cooperating parallel processes
            - Locks, condition variables

        - Security
            - User authentication
            - Secure sessions
        - User interface
            - GUI, media
    - ‘Under the covers’ services
        - Enclosure management
        - Hot-plug,power, fans
        - Software updates
        - Dynamic resource allocation and scheduling
        - Networks,protocols,domain services
- What are the different layers of services in a system?
    - Libraries
        - Convenient functions.
        - Reusable code makes programming easier
        - Encapsulates complexity
        - Multiple bind-time options
            - Static,shared,dynamic
        - No special privileges, just code
    - The Kernel
        - Functions that require privilege
            - Privileged instructions like interrupts and I/O
            - Allocation of physical resources
            - Ensuring process privacy and containment
            - Ensuring integrity of critical resources
    - System services
        - Trusted code that isn’t in the kernel. Doesn’t need to access kernel data structures / execute privileged instructions
        - Some are privileged processes, e.g some can directly execute I/O operations / set user credentials
    - Middleware
        - Software that is a key part of the application/service platform but not part of the OS
            - Apache,Nginx
        - User-mode code which means
            - Easier to build,test and debug
            - More portable
            - Can crash and be restarted
- What are the different service delivery methods?
    - Subroutines
        -
        - Access services via direct subroutine calls
        - Advantages

            - Extremely fast
            - DLLs enable run-time implementation binding
        - Disadvantages
            - But all services implemented in same addressspace
            - Limitedability to combine different langauges
            - Limited ability to change libary functionality
    - System Calls
        - Force entry into OS
        - Implementation is in kernel
        - Advantages
            - Able to use privileged resources
            - Able to share/communicate with other processes
        - Disadvantages
            - All implemented on the local node
            - 100x-1000x slower than subroutine calls
            - Evolution is very slow and expensive
    - Messages
        - Exchange messages with a server
        - Advantages
            - Server can be anywhere
            - Service highly scalable and available
            - Service can be implemented in user-mode code
        - Disadvantages
            - 1,000x - 100,000x slower than subroutine
            - Limited ability to operate on process resources

- What does interoperability/compatibility/portability require?
    - Completeness - everything should be included in spec
    - Everything should be explicit
    - Compliance: complete testing impossible on every system, so compliance suites validate the implementations
    - Stability: API requirements are frozen at compile time. Everyone must support those interfaces in the future
- Advantages of using objects (in providing services)?
    - Easier to use than original resources
    - Compartmentalize/ encapsulate complexity
    - Eliminate behavior that is irrelevant to user
    - Create more convenient behavior
- Difference between a simplifying abstraction and a generalizing abstraction
    - Simplifying abstractions encapsulate implementation details and offer more convenient or powerful behavior
    - Generalizing Abstractions make many different things appear the same. It requires a common / unifying model

- What are the different types of resources? / different ways to virtualize physical resources?
    - Serially reusable resources
        - Used by multiple clients, one at a time
        - Requires access control to ensure exclusive access
    - Partitionable resources
        - Different clients use different parts at same time
        - Requires access control for containment/privacy
    - Sharable resources
        - Usable by multiple concurrent clients
        - Often involves mediated access
        - Often involves under-the-covers multiplexing

Lecture 3

Describe the components of a software tool chain.

- **Compiler:** Reads source modules and included header files, parses the input language, and infers the intended computations,
- **Assembler**.
  In user-mode code, modules written in assembler often include:
    - performance critical string and data structure manipulations
    - routines to implement calls into the operating system

  In the operating system, modules written in assembler often include:
    - CPU initialization
    - first level trap/interrupt handlers
    - synchronization operations

  The output of the assembler is an object module containing mostly machine language code. But, because the output corresponds only to a single input module for the linkage editor:
    - some functions (or data items) may not yet be present, and so their addresses will not yet be filled in.
    - even locally defined symbols may not have yet been assigned hard in-memory addresses, and so may be expressed as offsets relative to some TBD starting point.
- **Linkage editor:**
    - The linkage editor reads a specified set of object modules, placing them consecutively into a virtual address space, and noting where (in that virtual address space) each was placed.
    - It also notes unresolved external references
    - searches a specified set of libraries to find object modules that can satisfy those references, and places them in the evolving virtual address space as well.

    - it finds every reference to a relocatable or external symbol and updates it to reflect the address where the desired code/data was actually placed.
- **Program loader:**
    - It examines the information in a load module, creates an appropriate virtual address space, and reads the instructions and initialized data values from the load module into that virtual address space.
    - If the load module includes references to additional (shared) libraries, the program loader finds them and maps them into appropriate places in the virtual address space as well.

Describe the process of linkage editing.

- Filling in all of the linkages (connections) between the loaded modules.
- Linkage editor would search the specified libraries for a module that could satisfy the external reference,
- The linkage editor would add it to the virtual address space it was accumulating. Then, with all unresolved external references satisfied, and all relocatable addresses fixed, the linkage editor would go back and perform all of the relocations called out in the object modules.

Difference between static and shared libraries.

- Static linking involves permanence but has the following disadvantages
- Thousands of identical copies of the same code increase the required down-load time, disk space start-up time (to read them into memory) and memory (while they are executing).
- Better if all pgms used popular library to share a single copy.
- newer version of a library is probably better than an older version. But with static linking, each program is built with a frozen version of each library, as it was at the time the program was linkage edited. Better to load latest library.

Implement a shared library

- Reserve an address for each shared library. This is possible in 32-bit architectures, and easy in 64-bit architectures.
- Linkage edit each shared library into a read-only code segment, loaded at the address reserved for that library.
- Assign a number (0-n) to each routine
- Create a stub library, that defines symbols for every entry point in the shared library, but implements each as a branch through the appropriate entry in the redirection table.
- Linkage edit the client program with the stub library.
- When the operating system loads the program into memory, it will notice that the program requires shared libraries, and will open the associated (sharable, read-only) code segments and map them into the new program's address space at the appropriate location.

Why shared library over static library?

- Version of library (and therefore interface) is determined at program load time instead of linkage editing time. This means that we can easily load in the latest version of the library.
- Still need to know the library name at linkage editing time though (usually through a variable path or something). DLLs fix this.

Why are dynamically loaded libraries better than shared libraries?

Shared libaries have the following issues:

- There may be very large/expensive libraries that are seldom used; Loading/mapping such libraries into the process' address space at program load time unnecessarily slows down program start-up and increases the memory footprint.
- While loading is delayed until program load time, he name of the library to be loaded must be known at linkage editing time.

General model of implementing a dynamically loaded library (DLL):

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

What are the basic elements of subroutine linkage?

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

Describe the trap mechanism.

- there is a number (0, 1, 2 ...) associated with every possible external interrupt or execution exception.
- there is a table, initialized by the OS software, that associates a Program Counter and Processor Status word (PC/PS pair) with each possible external interrupt or execution exception.

- when an event happens that would trigger an interrupt or trap:
    1. the CPU uses the associated interrupt/trap number to index into the appropriate interrupt/trap vector table.
    2. the CPU loads a new program counter and processor status word from the appropriate interrupt/trap vector.
    3. the CPU pushes the program counter and processor status word associated with the interrupted computation onto the CPU stack (associated with the new processor status word).
    4. execution continues at the address specified by the new program counter.
    5. the selected code (usually written in assembler, and called a *first level handler*:
        - saves all of the general registers on the stack
        - gathers information from the hardware on the cause of the interrupt/trap
        - chooses the appropriate second level handler
        - makes a normal procedure call to the second level handler, which actually deals with the interrupt or exception.
- after the second level handler has dealt with the event, it returns to the first level handler, after which ...
    1. the 1st level handler restores all of the saved registers (from the interrupted computation).
    2. the 1st level handler executes a privileged return from interrupt or return from trap instruction.
    3. the CPU re-loads the program counter and processor status word from the values saved at the time of interrupt/trap.
    4. execution resumes at the point of interruption.

Difference between procedure call and interrupt.

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

Remember that interrupt = some external event occurred, while trap = some exception/fault occurred.

How are processes created?

- The first thing that the OS must do to run a program is to load its code and any static data (e.g., initialized variables) into memory, into the address space of the process
- Some memory must be allocated for the program's run-time stack (or just stack) and heap.

- The OS will also do some other initialization tasks, particularly as related to input/output (I/O). For example, in UNIX systems, each process by default has three open file descriptors, for standard input, output, and error;

Describe process states and transitions.

- Running: In the running state, a process is running on a processor. This means it is executing instructions.
- Ready: In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- Blocked: In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place.
- Being moved from ready to running means the process has been scheduled; being moved from running to ready means the process has been descheduled. Once a process has become blocked (e.g., by initiating an I/O operation), the OS will keep it as such until some event occurs (e.g., I/O completion); at that point, the process moves to the ready state again (and potentially immediately to running again, if the OS so decides).

Differences between child and parent in fork system call.

- Child isn't an exact copy.
- has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of fork() is different.
- Specifically, while the parent receives the PID of the newly-created child, the child receives a return code of zero.

Describe the process through fork(), wait(), and exec()

- The parent process calls wait() to delay its execution until the child finishes executing. When the child is done, wait() returns to the parent.

Why is the separation of fork() an exec() essential to building a UNIX shell?

- It lets the shell run code after the call to fork() but before the call to exec()
- this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

Describe direct execution and what happens when it attempts to perform restricted operations.

- Run the program directly on the CPU. OS creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry point (i.e., the main() routine or something similar), jumps to it, and starts running the user's code. If process tries to perform restricted operation, introduce user mode.
- To execute a system call, a program must execute a special trap instruction.
- This instruction jumps into the kernel and raises the privilege level to kernel mode;
- System performs necessary operations

- When finished, the OS calls a special return-from-trap instruction, returns into the calling user program while simultaneously reducing the privilege level back to user mode.
- All LDE is is that we let processes run directly on the CPU, but in a limited fashion. That means the OS has privileged ways of stopping a process from running on CPU so that it can regain control:
    - interrupts
    - traps
    - syscalls

How does setting up a trap table at boot time help the kernel control?

- When the machine boots up, it does so in privileged (kernel) mode, and thus is free to configure machine hardware as need be. OS tells hardware what code to run when certain exceptional events occur. The OS informs the hardware of the locations of these trap handlers, usually with some kind of special instruction.
- Trap table defines what handler a specific trap number should use. These handlers and this mapping is privileged, so it is defined by the OS at boot time and at boot time, the hardware remembers this.
    - So now when we trap into the OS the hardware knows what to do given a trap number, it can just get the handler from the table (which it has remembered since boot time)

Describe the advantages and disadvantages of switching between processes using the "wait for system calls" approach and the "OS takes control" approach.

- Waiting for system calls: trust processes to behave. Most actually do frequently make system calls. Apps transfer control even when illegal operation. Problem with infinite loop.
    - Advantages of “wait for system call”: potentially less issues with scheduling and concurrency. We know that we can only run other processes when one yields, so we don’t have to worry about locking stuff up and randomly getting preempted during shitty times (critical sections)
- OS takes control: timer interrupt which raises an interrupt handler
    - With the OS takes control the key advantage is that a single process cannot take over the machine. Now other processes won’t starve.
    - disadvantage is that there will be concurrency and scheduling issues that we will solve

Describe the process of a context switch.

- OS has to do is save a few register values for the currently-executing process (onto its kernel stack, for example) and restore a few for the soon-to-be-executing process (from its kernel stack).
- OS will save the general purpose registers, PC, as well as the kernel stack pointer of the currently-running process, and then restore said registers, PC, and switch to the kernel stack for the soon-to-be-executing process. By switching stacks, the kernel enters the call to the switch

code in the context of one process and returns in the context of another. When OS excutes return-from-trap instruction, the soon-to-be-executing process becomes the currently-running process. And thus the context switch is complete.

- Full pattern:
    1. Process A is running but a timer interrupt happens
    2. Hardware saves regs(A) onto kernel-stack of A
    3. Raise privilege to kernel mode
    4. Jump to trap handler
    5. Trap handler calls switch
    6. In switch, saves regs(A) to proc-struct of A
    7. Restores regs(B) from proc-struct(B)
    8. Executres a return from trap instruction, returning in the context of B (initially tarpped in A's context!)
    9. Hardware restores regs(B) to kernel-stack of B
    10. Go back to user mode privilege
    11. Jump to PC of B

What happens when during a system call, a timer interrupt occurs?

- Several ways to handle this - it's a concurrency issue. If we give timer interrupts a higher precedence than a system call, then leave the system call and execute the timer interrupt code (probably switching to and running another process).
    - This may be complicated since when we switch back to the process that was running the system call, we would need to make sure **not** to return to its user-mode context and PC, but to the contxt and PC of the system call
        - should be able to be done with saving/popping tho
- Or, when we're in kernel mode, we can just disable interrupts and not deal with them

Lecture 4

- What are the execution states of a process
    - Running : In memory and running
    - Ready: In memory and ready to be run
    - Blocked: Cannot be run, might be swapped out of memory
    - Swapped Out: moved out of memory, stored on disk
    - Swap wait: ready to be moved back into memory
- Name and describe 4 useful metrics in comparing scheduling policies
    - (Mean) Turnaround time (Completion Time) : how long it takes for a job to complete
    - (Mean) Response time: how long it takes for a job to be scheduled for the first time
    - Fairness : How well does the policy evenly divide the CPU among active processes

    - Throughput: Operations per second for a particular activity or job mix
- What are three different kinds of systems with different scheduling goals?
    - Time Sharing
        - Fast response time to interactive programs
        - Each user gets an equal share of the CPU
        - Execution favors higher priority processes
    - Batch
        - Maximize total system throughput
        - Delays of individual processes are unimportant
    - Real-time
        - Critical operations must happen on time
        - Non-critical operations may not happen at all
- Describe 5 different scheduling policies / algorithms
    - First in First out (Non-Preemptive)
        - First job to arrive gets scheduled first
        - Advantages:
            - very simple to implement
            - seems intuitively fair
            - all process will eventually be served
        - Problems:
            - highly variable response time (delays)
            - a long task can force many others to wait (convoy)
    - Shortest Job First (Non-Preemptive)
        - First job to arrive still gets scheduled first, but of those that arrive at the same time, the shortest is scheduled first
        - Advantages:
            - Likely to yield a fast response time
        - Disadvantages:
            - Some processes may face unbounded wait times (starvation)
            - still suffers from the convoy effect if a large job arrives first
    - Priority (Non-preemptive)
        - All the processes are given a priority, and the highest priority job is run until it blocks or yields
        - Advantages:
            - User controls priorities
            - Can optimize per-customer "goodness" function
        - Disadvantages:
            - Still subject to starvation
    - Shortest Time-to-Completion First (Preemptive - can switch jobs)
        - First job to arrive gets scheduled, but upon the arrival of a new job, the scheduler runs whichever of the remaining jobs has the least time left to completion.
        - Can starve larger jobs

        - This is optimal for turnaround time..if the OS can predict the future.
    - Round Robin (Preemptive)
        - Runs each job for a ‘time slice’ and then switches to the next job in the run queue.
        - Time slice must be multiple timer interrupt period
        - Shorter time slice means better response-time but the cost of context switching will dominate the overall performance. Need to make the time slice long enough to amortize the cost of context switch without making it so long that you increase the response time.
        - Highest response time and high fairness but one of the worst policies for turnaround time
- Compare Non-Preemptive Scheduling to Preemptive Scheduling
    - Non-Preemptive Scheduling
        - Scheduled process runs until it yields CPU
        - Works well for simple systems
        - Depends on each process to voluntarily yield
            - Easily leads to starvation
    - Preemptive scheduling
        - A process can be forced to yield at any time
        - Advantages:
            - Enables enforced “fair share” scheduling
        - Disadvantages:
            - Introduces gratuitous context switches
            - Creates potential resource sharing problems
- Describe the cost of a context-switch
    - Entering the OS : taking interrupt, saving registers, calling scheduler
    - Cycles to choose who to run
    - Moving OS context to the new process: switch process descriptor, kernel stack
    - Switching process address spaces: map out old process, map-in new process
    - Losing L1 and L2 cache contents
- What is dynamic equilibrium
    - Result of competing processes and negative feedback
    - Processes are self-calibrating and auto adapt to changing circumstances
- Why is it hard to optimize both response time and turnaround time
    - if you are willing to be unfair, you can run shorter jobs to completion, but at the cost of response time; if you instead value fairness, response time is lowered, but at the cost of turnaround time
- What are the rules of a Multi-Level Feedback Queue
    - Rule 1: If Priority(A) > Priority(B), A runs (B doesn’t).
    - Rule 2: If Priority(A) = Priority(B), A & B run in Round Robin
    - Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).

  - Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
    - This rule is important: makes sure that there's no gaming of the CPU. Even if a process tries to take advantage of time slice ends by giving up CPU right before its time slice end (and thus retaining its priority or moving up, and starving other processes), it will eventually be lowered priority.
  - Rule 5: After some time period S, move all the jobs in the system to the topmost queue (priority boosting to avoid starvation)
- How to control priorities for MLFQ?
  - As discussed, all of them enter at highest priority (so they definitely get SOME time to run) and eventually run again (when their priority is boosted). This is to ensure good response times and no starvation. But else? If we find that a process is continually yielding the CPU before its time slice end is up, we could move it up to a higher priority list/queue
    - In higher priority lists, the time slice ends are shorter. It's better for interactive processes
  - If we have a process that is consistently using its entire time slice without giving up the CPU, it may be performing batch/long running computations
    - So it is not very interactive. Give it a lower priority that has longer time slices but also less priority
- What is an ideal time slice end for a process
  - It's expected value
  - What is a real-time system? What metrics can one use to check the performance of a real-time system?
  - A real-time system is one whose correctness depends on timing as well as functionality. The metrics that can be used are timeliness: how closely the system meets its timing requirements and predictability: how much deviation there is in delivered timeliness
- What characteristics of real time systems make scheduling easier?
  - We may actually know how long each task will take to run. This enables much more intelligent scheduling.
  - Starvation (of low priority tasks) may be acceptable. Understanding the relative criticality of each task gives us the freedom to intelligently shed less critical work in times of high demand.
  - The work-load may be relatively fixed.
- What are some real-time scheduling algorithms?
  - Static scheduling - Don't use a scheduler and just have things in a fixed order since everything is known (list of tasks, time for completion, etc.)
  - Dynamic Scheduling
    - How to choose the next job
      - shortest job first

            - static priority ... highest priority ready task
            - soonest start-time deadline first (ASAP)
            - soonest completion-time deadline first (slack time)
        - How they handle overload
            - best effort
            - periodicity adjustments ... run lower priority tasks less often.
            - work shedding ... stop running lower priority tasks entirely.
- Compare Hard Real Time Schedulers to Soft Real Time schedulers
    - Hard real time:
        - System absolutely must meet its deadlines
        - System fails if deadline is not met
    - Soft real time
        - Highly desirable to meet deadlines
        - Goal of scheduler is to avoid missing deadlines: “best effort”
        - May have different classes of deadlines
        - May have more dynamic/variable traffic
- Should you use preemptive scheduling in real-time systems?
    - NO:
        - preempting a running task will almost surely cause it to miss its completion deadline.
        - since we so often know what the expected execution time for a task will be, we can schedule accordingly and should have little need for preemption.
        - embedded and real-time systems run fewer and simpler tasks than general purpose time systems, and the code is often much better tested ... so infinite loop bugs are extremely rare.
- What is the process of un-dispatching a running process and re-dispatching a process
    - Enter the operating system somehow (e.g via a system call or clock interrupt)
    - Preserve the state of the process
    - Yield CPU
    - Select the next process to be run: get a pointer to its process descriptor
    - Locate and restore its saved state: restore code,data,stack segments, saved registers, PS and the PC
    - We are now executing a new process
- What is the process of blocking / unblocking a process
    - Process needs an unavailable resource like data that has not yet been read in from the desk
    - Process is blocked until the resource becomes available
    - Process is unblocked when the resource becomes available
- Why do we swap processes between primary and secondary storage
    - Make the best use of limited memory:
        - a process can only execute if it is in memory.
        - If it isn’t ready it doesn’t need to be in memory

    - Improve CPU utilization
        - When there are no READY processes, CPU is idle
        - Idle CPU time is wasted, reduced throughput
        - We need READY processes in memory
- Describe the processes of swapping out and swapping back in
        - Swapping out:
        - Process' state is in main memory: code, data
        - Copy out state to secondary storage
        - Update resident process descriptor: process no longer in memory
        - Freed memory now available for other processes
    - Swapping Back In:
        - Re-Allocate memory to contain process: code/data segments, non-resident process descriptor
        - Read data back from secondary storage
        - Change process state back to Ready
        - Involves a lot of time and I/O
- What is graceful degredation (in real time systems)?
    - System overloads will happen: random fluctuations in traffic, load bursts from unanticpicated events,additional work associated with errors
    - We have 3 options of 'graceful degredation' to deal with overloads
        - Offer slower service to all clients
        - Allow deadliens to get later and later
        - Offer on-time service to fewer clients

Lecture 5:

Q1: What are the three goals of memory abstraction?
A1:
Transparency: The program running should not know its address space is being virtualized.

Efficiency: The OS should be efficient in its virtualization. Ie translations must be fast and shouldn't take up much space. The OS does this through the use of TLBs

Protection: The OS should make sure that it protects one process' address space from interference from another process. It should also make sure that a process cannot access memory it isn't supposed to.

Q2: How does the free() call know how large the allocated memory is when the only parameter we pass to it is the virtual address of the beginning of the allocated memory?

A2: When allocating memory, the pointer returned actually allocates slightly more meory than asked for. It creates a head struct that holds information about the allocated memory like the size. This allows the free() call to know how much memory the pointer points to.

Q3: What are some common errors that can occur while dynamically allocating/freeing memory from the heap?

A:
i) Forgetting to allocate memory: referencing a pointer that doesn't point to malloc()'d memory/
ii) Not allocating enough memory: aka buffer overflow. Memory immediately after the allocated memory block can get corrupted (Or alternatively raise a segfault, depending on how malloc() was implemented)
iii) Forgetting to initialize malloc()'s memory: Accessing allocated memory that hasn't been assigned yet can give unexpected values. Easily prevented by using calloc()
iv) Forgetting to free memory: Some languages require explicit freeing of memory before the pointer goes out of scope. This can cause segments of memory to remain unusable even after the program that allocated it is done running. This problem once started can only be fixed by restarting the machine.
v) Freeing a pointer repeatedly: Self explanatory

Q4: Explain when a brk or sbrk syscall is made:

A4: When a process is initialized, the OS gives it some extra space for its heap to grow during execution. If the allocated space within the heap runs out, the program must had over control to the OS, that makes the brk syscall, which finds free pages to map them to the virtual address space of the process that initialized the brk syscall.

Q5: Describe the structure of a free list.

A5: A free list works like a linked list data structure. Each node contains a 'head' struct that contains information about the chunk of free memory right after it; information like its size etc. It can also contain a so called 'magic' number that can be used to do a sanity check to see whether the free() call is referring to the right block of memory.

Q6: Name and briefly explain the different mechanisms implemented to assign memory to a malloc call from the free list:

A6:
Best Fit: Goes through the entire free list, flags memory chunks just as big or bigger than the chunk requested, and returns the smallest of the flagged chunks. This method is very slow as it

needs to traverse the entire free list to find the best fit. Fragmentation is minimal in the beginning, but gets progressively worse with time.

Worst Fit: Does the opposite of best fit. Returns the largest chunk available to reduce fragmentation. This approach is just as slow, but the fragmentation is much worse, and gets worse with time.

First Fit: Returns the first chunk it finds that is bigger than the chunk requested. This can cause random fragmentation as the free list entry that is big enough that appears first decides the amount of fragmentation.

Next Fit: The free list must be sorted for this to work. Holds a pointer that points to the chunk in the free list that was last allocated. This way the iterator doesn't need to start from the beginning when it starts searching for the next requested chunk. Works on the assumption that the size of subsequent requests will be roughly of the same size. Performs slightly better than the other algorithms.

Q7: What are segregated lists?
A7: They are similar in structure to regular free lists except all their chunks are of similar sizes. This way if there is a 'popular' sized chunk, it'll be readily available for the MMU to hand over. Requests of other sizes are forwarded to a more general memory allocator.

What is buddy allocation?

An allocation algorithm that takes a large chunk of size 2^N and recursively splits it in 2 even parts until we reach a portion that's just big enough for our request (IE, we can't split anymore because then it'll be too small). We then allocate one of these chunks. Prone to internal fragmentation, if we requested 9 bytes then we'd get 2^4 bytes.

But it's really easy to find the buddy and check if the buddy is also free - only differs by 1 bit

Explain, in copious detail, slab allocation.

Slab allocation is an allocation algorithm that aims to be efficient at allocating memory for objects. It relies on the idea that initializing and destroying objects is very expensive. So it maintains a slab for one specific (and popular) object that is being requested, and caches an initialized state of the object so that we don't have to destroy and recreate everytime.

When it runs out of memory it asks main memory for more slabs

Q8: Go through the steps of Garbage Collection.
A8:

- resources are allocated, and never explicitly freed
- when the pool of available resources becomes dangerously small, the system initiates garbage collection:

1. begin with a list of all of the resources that originally existed.
2. scan the process to find all resources that are still reachable.
3. each time a reachable resource is found, remove it from the original resource list.
4. at the end of the scan, anything that is still in the list of original resources, is no longer referenced by the process, and can be freed.
5. after freeing the unused resources, normal program operation can resume.

What is defragmentation?

At attempt to reduce external fragmentation in main memory. TO reduce this, we organize the contents in memory so that they are as close as possible together, so that they fit into the smallest possible region

Q9: How does the CPU know which mode it is running in?
A9: There is a bit stored in some kind of processor status word that indicates which mode the CPU is currently running in.

Q10: What is the hardware's role in memory management?
A10:

a) It provides support for the base and bound register pairs
b) Memory operations are often privileged, and hence need hardware support to run protected instructions
c) In cases of exceptions like Segfaults, the program traps the syscall, and the OS needs to execute the error handling code with the help of the hardware.
d) The CPU/hardware also needs to raise an exception when a user mode program tries to access any other privileged instructions.
e) The OS must be able to tell the hardware what it must run when an exception occurs.

Q11 What are the advantages of Segmentation?
A11:

a) Having a base bound pair per segment instead of per program give us better control over where we can store the segments in physical memory and hence reducing fragmentation.
b) It allows for the sharing of read only code. If there's a special 'protection' bit to indicate that this segment is read-only, we can give multiple programs permission to enter the region and read and execute that code. (don't pages do this too?)
c) Each program accessing the shared code will think that the shared block is in their private address space due to virtualization.
d) It allows a more efficient mechanism for popular request sizes or popular request objects. If we were using paging it wouldn't be any more efficient, but with segmentation we can have things like free lists

Q12 What are the differences between Coarse grained and fine grained segmentation?

A:
Fine grained:

a) Smaller segments, more numerous, give OS more options over where to store individual segments and hence reduces fragmentation.
b) Slower access due to the large number of segments. Needs to maintain a segment table data structure to keep track of all t segments and improve access time by implementing some caching.

Coarse Grained:

a) Larger segments, few in number, OS has limited options over where to store individual segments in physical memory, and hence has higher fragmentation.
b) Faster access, no need to maintain a segment table, register pairs are usually stored in the process' address space.

NOTE: When the OS does a context switch, all the base-bound register pairs get flushed and saved to disk. They are then restored again when the process is back in the running state.

Lecture 6

Q1: How does paging work?
A1:

a) Memory is divided into 'pages' of fixed size
b) The physical addresses of pages are abstracted and the virtual addresses have bits that translate to physical page numbers.
c) To keep track of each virtual page, the OS maintains a per-process structure called a 'page table'
d) The page table stores address translations.
e) The virtual address contains bits that translate to a physical page number, and the other bits are the offset of bits within that page.
f) The OS and hardware work together for this translation
g) Translating can often be slow, hence some translations are cached in Translational Lookaside Buffers (TLBs) aka page table cache(not a real name).

Q2: Define the structure of a page table
A2 A page table can be structured in many ways. The simplest way is as an array:

a) The Virtual Page Number (VPN) act as the index of that array.
b) Each index location points to a Page Table Entry (PTE)
c) The contents a PTE are: The Physical Frame Number (PFN), a valid bit to check if the translation is valid (ie if there's actually anything stored in that physical address)
d) The PTE also has some more bits that define some more behaviour.

Q3: What is the need for a TLB?
A3: Translating a virtual address to a physical address from a page table that is stored in memory can get incredibly slow. For this reason we introduce TLBs.

TLBs make address translation much faster as they cache frequently requested page numbers. They make use of temporal and spatial locality to speed up translations.

Q4 What happens when there's a TLB miss?

A:When there is a TLB miss, the software raises an exception that causes the current stream to pause, raises the privilege level to kernel mode, and jumps to the trap handler.
Then, the OS uses special privileged instructions to update the TLB, and return from the trap. Usually, when the OS returns from the trap the instruction that caused the trap get skipped and execution of the rest of the code begins, but with TLBs, the instruction that caused the TLB miss is re run. This time there will be a TLB hit.
• Extract VPN => Check if in TLB => TLB Miss => Use PTBR to find page table => Look up PTE from VPN => If page in memory, extract PFN from PTE => install into TLB => retry instruction => TLB Hit = *EH*
• Extract VPN => Check if in TLB => TLB Miss => Use PBTR to find page table => Look up PTE from VPN => Page not in memory (determined by present bit) => Page Fault => Page Fault Handler swaps the page into memory from disk => update page table as present, PFN field of PTE => TLB miss ... => TLB hit = Fuck that was long

Q5: What are common replacement policies when there's no space to bring more entries to the TLB?

A5: There are a variety of different policies the TLB can have:

a) LRU: Least Recently Used: Evicts the least recently used entry, ie the entry with the lowest temporal locality. Works well in most cases except when say a program loops over n+1 pages with a TLB of size n pages. In such a case, there will be continuous TLB misses.
b) Random: It's random, so its performance is also random. However, we can say with some certainty that in the aforementioned case when a program loops over n+1 pages, the random policy works much better than LRU.

c) Why does the LRU appear to work well in most cases? What about the other cases?
   i) It works well bc programs exhibit some degree of temporal and spatial locality. Pgms are likely to come back to same data, or reference code within the same page
   ii) When doing round-robin time sharing with multiple processes, they rarely access the same pages. Most recently used pages in memory will not be run for a long time and least recently used pages belong to process that is about to be run.

d) What is trashing and what causes it?
   i) Dramatic degradation of system performance bc lack of memory to run all of the ready processes
   ii) Improved performance from reducing page faults will make up for the delays processes experience while being swapped.

e) How to implement a working set?
   i) Each page frame is associated with owning process, and a process has an accumulated CPU time.
   ii) Each frame has a referenced time.
   iii) We also maintain a target age parameter
   iv) Age decisions are determined by accumulated CPU time in owning the process
   v) If page is younger than target age, do not replace
   vi) If page is older than target age, take it away from curr. Owner and give it to a needy process.
   vii) No pages older than target age?
      1) full scan and replace the oldest page in memory (expensive)
      2) If target age well chosen, reduce number of processes in memory

f) Describe the page stealing algorithm (dynamic equilibrium).
   i) Every process is continuously losing pages that it has not recently referenced .
   ii) Every process is continuously stealing pages from other processes.
   iii) Processes that reference more pages more often, will accumulate larger working sets.
   iv) Processes that reference fewer pages less often will find their working sets reduced.
   v) When programs change their behavior, their allocated working sets adjust promptly and automatically.
   vi) The continuously opposing processes of stealing will allocate available memory in proportion to working set sizes. Manages memory to avoid thrashing.

Lecture 7

Reading: Introduction to IPC

Describe the difference between pipes and regular pipes.

- A named-pipe (*fifo(7)*) is a baby-step towards explicit connections.

    - persistent pipe, whose reader and writer(s) can open it by name, rather than inheriting it from a *pipe(2)* system call. A normal pipe is custom-plumbed to interconnect processes started by a single user.
    - A named pipe can be used as a rendezvous point for unrelated processes.
- For pipes:
    - If the reader exhausts all of the data in the pipe, but there is still an open write file descriptor, the reader does not get an *End of File*(EOF). Rather the reader is blocked until more data becomes available, or the write side is closed.
    - If the writer gets too far ahead of the reader, the operating system may block the writer until the reader catches up. This is called *flow control*.

- For named pipes, Readers and writers have no way of authenticating one-another's identities.
- For named pipes, writes from multiple writers may be interspersed, with no indications of which bytes came from whom.
- Named pipes do not enable clean fail-overs from a failed reader to its successor.
- For name pipe, all readers and writers must be running on the same node.
- Named pipes are persistent and remains in the file system for later use after done, while regular pipes are automatically deleted after use.
- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use

What is a mailbox (inter-process comunication)

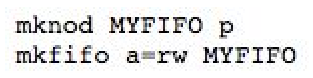
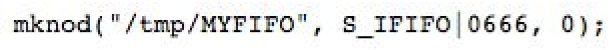
- Data is not a byte-stream. Rather each write is stored and delivered as a distinct message.
- Each write is accompanied by authenticated identification information about its sender.
- Unprocessed messages remain in the mailbox after the death of a reader and can be retrieved by the next reader.
- subject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

Describe how shared memory is the most efficient way to move data between processes

- create a file for communication.
- each process maps that file into its virtual address space.
- the shared segment might be locked-down, so that it is never paged out.
- the communicating processes agree on a set of data structures (e.g. polled lock-free circular buffers) in the shared segment.
- anything written into the shared memory segment will be immediately visible to all of the processes that have it mapped into their address spaces.
- Comes at a price:
    - Only works for processes on the same memory bus
    - No authentication of which data came from which process

Reading: Named Pipes

Describe the several ways of creating a named pipe.

- Directly from shell with the following commands:
  - 
```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```
- In C
  - Make use of mknod() system call.
  - 
```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```
  - Parameters are file requested permissions.
  - If creating device file, the third parameter is useful.

Describe the process of blocking in FIFO

- FIFO is opened for reading
- process will "block" until some other process opens it for writing.
- This action works vice-versa as well.
- If undesirable, the O_NONBLOCK flag can be used in an open() call to disable the default blocking action.

What are the goals of IPC?
- Simplicity
- Convenicence
- Generality
- Efficiency
- security/privacy
- robustness/reliability

Compare messages to streams (in IPC)
- Streams
  - Continuous stream of bytes
  - Read or write few or many bytes at a time
  - Write and read buffer sizes are unrelated
  - Stream may contain app-specific record delimiters
- Messages (ask datagrams)
  - A sequence of distinct messages
  - Each message has its own length
  - Message is typically read/written as a unit

    - Delivery of a message is typically all or nothing

How do sockets work?

- Connections between addresses/ports
- Many data options like TCP(reliable) and UDP(best effort)
- Complex flow control / error handling
- trust/privacy/security/integrity
- Both stream and message semantics
- Socket destroyed when creator dies

Compare in-band to out-of-band messages

- In-band
    - Messages delivered in same order as sent
    - Message n+1 won't be seen til after message n
- Out of band
    - Messages that leap ahead of queued traffic
    - Use priority to cut ahead in the queue
    - Deliver them over a separate channel

When would you use a thread vs process?

- Processes
    - Running multiple distinct programs
    - creation/destruction are rare events
    - Running agents with distinct privileges
    - Limited interactions and shared resources
    - Prevent interference between processes
    - Firewall one from failures of other
- Threads
    - Parallel activities in a single program
    - Frequent creation/destruction
    - All can run with same privileges
    - They need to share resources
    - They exchange many messages/signals
    - No need to protect from each other

What are the benefits of parallelism

- Improved throughput
- Improved modularity
- Improved robustness
- A better fit to emerging paradigms: e.g client server computing/ web based services

What is a race condition? When is it caused?

- A race condition occurs when the results of a multithreaded program are dependent on the timing execution of the code. It is when multiple threads access a critical section at once. A critical section is a piece of code that accesses a shared resource.

Reading: User-Mode Thread Implementation

What are problems with implementing threads in a user mode library?

- When system call blocks:
  - All threads stop executing. Since in user mode, OS doesn't know which threads are still runnable
- Exploiting multiple processes
  - OS can process each to run in parallel. OS must be aware there are multiple threads, or will fail.

What does atomic/atomically mean?

- It means "as a unit", as in, the atomic instructions happen all at once. It can't be interrupted mid instruction, either it runs fully or it doesn't run at all.

What is an indeterminate program?

- An indeterminate program consists of one or more race conditions. The output of the program varies from run to run, depending on which threads ran when. The outcome is not deterministic.

Compare kernel mode threads to user mode threads

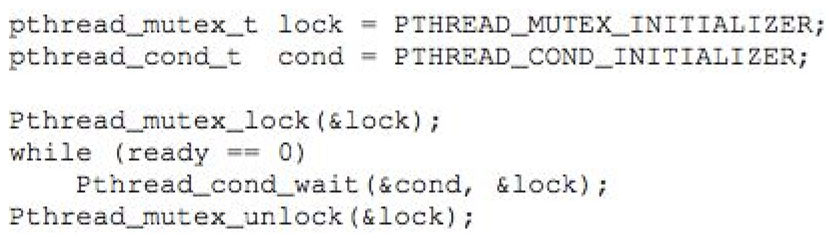
- Kernel mode
  - Multiple threads can truly run in parallel
  - One thread blocking does not block others
  - OS can enforce priorities and preemption
  - OS can provide atomic sleep/wakeup/signals
- User-Mode
  - Fewer system calls
  - Faster context switches
  - Ability to tailor semantics to application needs

What are the performance implications of non-preemptive scheduling and preemptive scheduling?

- Non-preemptive
  - user-mode threads operating in with a *sleep/yield* model are much more efficient than doing context switches through the operating system. There are, today, *light weight thread* implementations to reap these benefits.
- Preemptive
  - costs of setting alarms and servicing the signals may well be greater than the cost of simply allowing the operating system to do the scheduling.

Lecture 8

Describe the process of a mutex lock using condition variables in the following process:



```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

- In this code, after initialization of the relevant lock and condition3 , a thread checks to see if the variable ready has yet been set to something other than zero. If not, the thread simply calls the ait routine in order to sleep until some other thread wakes it.

Why should you use a simple globe teen two thread?

Describe coarse-grained locking strategy versus fine-grained approach.

- Instead of one big lock that is used any time any critical section is accessed (a coarse-grained locking strategy),
- one will often protect different data and data structures with different locks, thus allowing more threads to be in locked code at once (a more fine-grained approach).

What are the problems with using mutual exclusion to disable interrupts for critical sections?

- First, this approach requires us to allow any calling thread to perform a privileged operation (turning interrupts on and off), and thus trust that this facility is not abused.
- Trouble manifests in numerous ways:

    - a greedy program could call lock() at the beginning of its execution and thus monopolize the processor
    - an errant or malicious program could call lock() and go into an endless loop. OS never regains control of the system, and there is only one recourse: restart the system.
- Second, the approach does not work on multiprocessors. If multiple threads are running on different CPUs, threads will be able to run on other processors (regardless of interrupt disables) and enter the critical section.
- Third, turning off interrupts for extended periods of time can lead to interrupts becoming lost
- Finally, and probably least important, this approach can be inefficient.

Compare the Test-and-set and CompareAndSwap Locks.

- Test-and-set:
    - It returns the old value pointed to by the ptr, and simultaneously updates said value to new.
    - This sequence of operations is performed atomically. The reason it is called "test and set" is that it enables you to "test" the old value (which is what is returned) while simultaneously "setting" the memory location to a new value
- Compareandswap:
    - Test whether the value at the address specified by ptr is equal to expected; if so, update the memory location pointed to by ptr with the new value.
    - If not, do nothing.
    - Both cases return the actual value at that memory location, thus allowing the code calling compare-and-swap to know whether it succeeded or not.

Describe why test-and-set and compareandswap are atomic exchanges.

Describe the usage fetch-and-add in building a ticket lock.

- Fetch-and-add instruction, atomically increments a value while returning the old value at a particular address.
- The basic operation is pretty simple: when a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value; that value is now considered this thread's "turn" (myturn).
- The globally shared lock->turn is then used to determine which thread's turn it is; when (myturn == turn) for a given thread, it is that thread's turn to enter the critical section.
- Unlock is accomplished simply by incrementing the turn such that the next waiting thread (if there is one) can now enter the critical section.
- Ensures progress for all threads.

Explain the usage of queues for sleeping instead of spinning.

- park() to put a calling thread to sleep, and unpark(threadID) to wake a particular thread as designated by threadID.
- These two routines can be used in tandem to build a lock that puts a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free.

In the case of parent waiting for a child using condition variable, what are the two cases to consider?

- First case: Parent creates the child thread but continues running itself, calling pthread_join waiting for child to complete.
- Second case: child runs immediately when created, sets done to 1, calls signal to wake a sleeping thread, done.

Lecture 9

What is the producer/consumer (bounded buffer) problem?

- Consider producer and consumer threads. Producers generate data and place in the buffer for the consumers to grab.
- Bounded buffer is a shared resource, so we need shared buffer.
- Use a condition with lock mutex.
- Producer want to fill butter, waits for it to be empty. Consumer uses same logic and waits for fullness.
- Consumer runs, acquires lock, checks if any buffers ready for consumption, waits (releasing lock)
- Producer runs, acquires lock, checks if buffer is full, goes and fills buffer bc empty. Producer signals that buffer filled. Moves the first consumer from sleeping form the ready queue to run.
- Another consumer sneaks in and consumes buffer. Tc1 acquires lock but no buffers to consume.
- State may change by thee time the thread wakes up

What is the solution to the producer/consumer problem?

- Enable more buffer slots.
- Reduces context switches with multiple producers or consumers.
- Producer only sleeps if all buffers are currently filled. Consumer only sleeps if all buffers are currently empty.

If there are multiple threads attempting to free memory, how do we decide which thread to wake up?

- Replace the pthread cond signal() call in the code above with a call to pthread cond broadcast(), which wakes up all waiting threads. By doing so, we guarantee that any threads that should be woken are.

Describe the process of a sloppy counter.

- Representing a single logical counter via numerous local physical counters, one per CPU core, as well as a single global counter.
- When a thread running on a given core wishes to increment the counter, it increments its local counter; access to this local counter is synchronized via the corresponding local lock.
- threads across CPUs can update local counters without contention, and thus counter updates are scalable.

Describe where and why you would place locks in a concurrent linked lists, concurrent queues, and concurrent hash tables?

- Concurrent linked lists: around every node bc of hand-over-hand locking. Enables high degree of concurrency in list operations.
- Concurrent queues: around the tail and the head to enable concurrency of enqueue and dequeue operations.
- Around the buckets, enables many concurrent operations.

What are semaphores and its basic routines?

- A semaphore is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are sem wait() and sem post()1

Follow the process of binary semaphores in the figure below.

- Initial value should be 1.
- First thread will dec val of sempahore to 0. Will wait as long as val is less than 0. Since val is 0, will return. Thread 0 enters critical section.
- Calls sem_post() and restores value of semaphore to 1.
- Another thread tries to enter while holding the lock which is at 0. Dec to -1, and thus wait. Thread 0 calls sem_post(), back to 0. Thread 1 takes lock and finishes, restoring value back to 1.

Why should a semaphore be set t0 in the process where a parent is waiting of a child?

How to solve the producer/consumer (bounded buffer) problem with semaphores, mutual exclusion?

- Reduce the scope of the lock. Simply move the mutex acquire and release to be just around the critical section; the full and empty wait and signal code is left outside. The result is a simple and working bounded buffer, a commonly-used pattern in multi-threaded programs

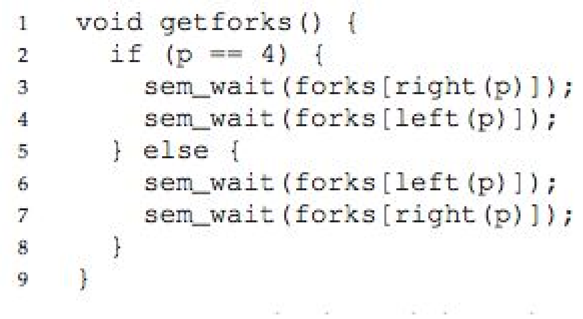
Describe the reader-writer lock

- If thread wants to update data structure, it calls rwlock_acquire_writelock() and rwlock_release_writelock().

- Any thread that wishes to acquire the write lock will have to wait until all readers are finished.
- The last one that exits calls sem_post() on writeLock, enabling a waiting ng writer to acquire the lock
- Add more overhead and do not speed up performance.

Describe the dining philosopher's problem and how semaphores can be used to solve it.

- There five philosophers sitting in a circle. Between each pair of philosophers is a single fork. The philosophers each have times where they think, and don't need any forks, and times where they eat. In order to eat, a philosopher needs two forks, both the one on their left and the one on their right.
- Change how forks are acquired by at least one of the philosophers. Specifically, let's assume that philosopher 4 (the highest numbered one) acquires the forks in a different order.
- No situation where each philosopher grabs one fork and is stuck waiting for another



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

-