CS 111 – Operating Systems, OSTEP Reading

Spring 2024 – Professor Peter Reiher

WEEK 1

Operating Systems Principles

Complex, Bigger Software causes:

- More magnitudes of code
- More difficulty understanding
- More dependencies

Hierarchical:

- System is comprised of smaller components that can be understood individually
    - UC Campuses → UCLA → Schools → Engineering → Depts…

Modularity:

- Groups share a coherent purpose
- Functions can be performed by everyone in group
- Union of groups achieve system's purpose

Cohesion: Combine similar functionalities into smallest modules

Appropriate Abstraction: Interface which enables client to specify parameters most meaningful to them to easily get desired results

- Hides more complexity from client, makes interface more easy to learn
- Expands provider's ability to change interface in the future

Interface Contracts: Specifications which should work no matter the changes in the interface

OS's are best designed in an incremental fashion

3. Architectural Paradigms

Mechanism: Keeps track of resources, grants/revokes access to resources

Policy: Controls which clients get resources when, plug-in

Indirection: Implementations are accessed indirectly (implementation not stored on system locally)

Federation framework: Registers available implementation, enables clients to select desired implementation, automatically handles routing

Deferred Binding: Modules are only dynamically loaded when OS requests it

Dynamic Equilibrium: Resource allocation should be driven by opposing forces, systems dynamically and automatically adapt to wide range of conditions

Data structures a main determiner for OS performance

<u>OSTEP Ch 1-2</u>:

- Programs run on instructions:
    - Fetch from memory, decode, execute
- OS makes system <u>easy to use</u>

Virtualization:

- Converts physical resource into easy-to-use virtual form
    - OS creates an illusion (virtualizing CPU to perform tasks simoltaneously)
        - Policy: Declares what order programs should run
- System Call: Passing of process to OS
- OS is a system manager:
    - Computer hardware are resources
    - OS manages resources

Virtualizing memory:

- Physical memory is array of bytes. Memory is read using addresses

- Each process creates a virtual address space, OS maps to physical memory

Concurrency:

- CPU creates threads which work on same task simultaneously
    - Thread: Function running in same memory space as other functions

Persistence:

- DRAM → Volatile manner of storage → disappears when power is off
- SSD, Hard Drive → holds data persistently
- Standard Library → Uses device driver → Device driver connects OS to storage

Abstractions:

- Makes system convenient, easy to use
- Provides high performance, minimizes overheads of OS
- Provides protection between applications through isolations

OS must be:

- Reliable
- Secure
- Mobility

Software Interface Standards

Compatibility sparked by ISVs (Independent Software Vendors) and killer applications. Resulting in:

- High costs to build different versions of applications tailored to specific hardwares
- Computer sales driven by how many applications were supported, no ISV ports results in no customers

- Software Portability is main focus of Hardware manufacturer and Software Vendors

Because computers are a daily tool:

- New applciations must work on both old and new devices
- Software upgrades cannot break existing applications

Software Interface Standardization:

- Pros:
    - Countless contributors
        - Standard becomes platform-neutral
        - Standard is well considered, broad to encompass a wide range of applications
            - Clear and complete specifications
            - Well developed conformance testing
    - Gives technology suppliers freedom to explore alternative implementations
        - Any implementation that follows standard should work with all existing and future clients
- Cons:
    - Constrains range of possible implementations
    - Interface standards imposes constraints on consumers
        - If feature used outside of standard, things break
        - Stake-holders contribute to difficulties of evolving standards

Interface vs Implementations:

- Interface standards: specifies behavior in implementation-neutral way as possible
- Implementations: How the interface is actually coded

- If interface written after the fact:
    - Implementation may not comply with interface
    - Ambiguity: Behavior may be part of interface or bug from implementation

As time passes, a given interface can:

- Maintain strict compatibility with other interfaces, rejecting support for new applications
- Compromise, partially support new and old interfaces and technologies
- Develop new interface to embrace new technologies, throw out older interfaces

Proprietary Interface: developed and controlled by single organization

- Will lose to if competing open standards are better
- Results in competing standards, reduces adoption
- Full shouldering of costs of development and evangelization

Open Standards: Developed and controlled by many providers and consumers

- Reduced freedom to adjust interfaces
- Giving up competitive advantage from being only provider
- Re-engineering existing implementations to bring into compliance with interfaces

APIs (Application Programming Interfaces): Defines what subroutines do and how they're used

- API specifications are written at source programming level
    - Describes how source code should be written to use features
        - Allows developers to easily recompile and execute API on any platform which supports it

        - Platform suppliers who support APIs can easily port to their platform
- If API is written is defined in platform-independent manner:
    - Datatypes can be represented in different manner (int in x86-64 vs x86-32)
    - Datatypes can be stored in different manner (big/little endian)
    - Certain features may not be implementable

Application Binary Interfaces (ABI ABIs): Binding of an API to an ISA, describes the machine language instructions and conventions that are used to call routines

- Normally contains:
    - Binary representation of key data types
    - Instructions to call to and return from a subroutine
    - Stack-frame Structure, responsibilities of caller & callee
    - Register conventions
    - Similar conventions for system calls
    - Formats of load modules, shared objects, dynamically loaded libraries
- Normally used by:
    - Compiler (to generate coded)
    - Linkage editor (to create load modules)
    - Program loader (to read load modules into memory)
    - OS (to process system calls)

Interface Stability

Specifications mean any collection of parts can be assembled into a working whole, meaning

- Programmer work simplified
- Program is more portable to more systems

- Training time for new programmers is reduced

Better Interfaces → More Functional Interface Specifications → Better working systems

Interface Stability:

- Software interface specifications acts as a contract:
    - Describes an interface and associated functionality
    - Implementation providers agree that systems conform to the specification
    - Developers agree to limit their use of functionality to what is outlined in the interface specifications

How to change an interface:

- Upwards-compatible: Interfaces can add features in future updates
- Backwards-Compatible: Add new feature to work in old version
    - Interface Polymorphism: Use different versions of method to extend functionality of new interfaces
    - Versioned Interfaces: Don't change interfaces, but continue servicing older interface versions
- Implement in smaller updates
    - Alpha versions
    - Micro-releases
    - Support commitments (support version for 5 years)

Interface Stability: Interfaces should be designed to accommodate future design

- Rearrange distribution of functionality between components to create simpler interface
- Design features to be implemented in the future
- Introduce unnatural degrees of abstraction to leave enough slack for future changes

Week 2

OSTEP Ch 4: (Abstractions: The Process)

Process: A running program, programs are bunches of instructions sitting on the disk

Time sharing: OS runs one process, stops it and runs another

- Creates illusion that many virtual CPUs exist when only 1 physical CPU exists

Mechanisms: Low-level methods/protocols that implement functionality

Context Switch: Gives OS ability to stop one program and start running another on given CPU

Policies: Algorithms for making some decisions within the OS

Scheduling Policy: Uses historical information, workload knowledge, and performance metrics to decide which programs get more resources

4.1 The Abstraction: A Process

Machine State: What a program can read or update when it is running

- Comprised of:
  - Memory: Instructions lie in memory, stored in addresses
  - Registers: Many instructions read/update registers
    - Program Counter/Instruction Pointer: Tells which instruction program will execute next
    - Stack Pointer and frame pointer: Manage the stack for function parameters, local variables, return addresses
  - I/O Information

4.2 Process API

Consists of calls programs can make related to processes

- Create: OS creates a new process

- Destroy: Interface that destroys processes forcefully
- Wait: Wait for a process to stop running
- Misc Control
- Status: Get status information about process

4.3 Process Creation

How does the OS get a program to run?

- OS loads program code and any static data into memory (address space of the process)
    - Programs initially reside on disk in executable format
        - OS must read bytes from disk and place them into memory
    - Eager loading: Load all code at once before running program
    - Lazy loading: Load pieces of code/data only as they are needed during program execution (used by modern OS)
- OS allocates some memory for program's:
    - Runtime stack
        - Initializes parameters for main() function
    - Heap
        - Used for explicitly requested dynamically-allocated data (malloc())
- OS performs other initialization tasks (focusing on I/O)
- OS has set the stage for program execution, start the program at the entry point (main()) and transfer control of CPU to process

4.4 Process States

Different states which a process can be in:

- Running: process runs on a processor and executes instructions
- Ready: Process if ready to run, OS has not run process yet

- Blocked: Process performs operation that makes it not ready to run until some other event takes place

Process moves from ready to running: Scheduled

Process moves from running to ready: Descheduled

4.5 Data Structures

Process List: OS's list for all processes in system that are ready, tracks which processes are currently running

- Each entry is called a Process Control Block, which is a structure that contains information about a specific process

Register Context: Holds contents of registers of stopped process

Systems can have:

- Initial State: State when process is being created
- Final State: Process has been exited but not yet cleaned up
    - Allows other processes to examine return code to see if process executed successfully

OSTEP Ch5: (Interlude Process API)

What interfaces should the OS present for process creation and control?

5.1 fork() System Call

Fork() system call creates a new process

- OS creates (almost) exact copy of the calling/same process
    - OS thinks two copies of process are running and returning from fork() system call
- Child process does not start running at main(), but acts as if it had called fork() itself
    - On return from fork(), parent outputs PID, child outputs 0

    - Output is not deterministic
        - Creates problems for CPU scheduler in Multi-threaded programs

Process Identifier/PID: Names process (to refer to later)

5.2 wait() System Call

Wait() system call waits for another process to finish its tasks

- Can make the fork() call deterministic (we can shoose to begin working on either the child or the parent)

5.3 exec() System Call

Exec() system call runs programs that are different from the calling program

- Given name of executable:
- Load code from executable
- Overwrite current code segment
    - Heap, stack, memory space are re-initialized
- OS runs program
    - No new process is created, current process is simply transformed

5.4 Motivating the API

New Process Creation allows shell to run code after call to fork() but before call to exec()

Shell: Shows a prompt, takes command as input, figures out where in file system the executable resides, calls fork() to create command, calls wait() to complete process creation, calls exec() to run command, calls wait() to wait for process to be complete

- Allows for redirection of program outputs into files and standard output because UNIX systems start looking for free file descriptors at zero

Pipe() System Call: Output of one process becomes input to another process

5.5 Process Control and Users

Kill() System Call sends signals to processes (pause, die, etc)

- SIGINT (interrupt) → control-c
- SIGTSTP (stop) → control-z
  - Fg to start after pause

Signals subsystem allows delivery, receiving, and processing of external events to processes or process groups

- Signal() System Call catches various signals

OS defines users:

- Users only control their own processes
  - Users cannot send processes (ex: interrupt) to other users
- OS defines superuser who administers the system (controls all processes)

OSTEP Ch. 6 (Mechanism: Limited Direct Execution)

Time Sharing: Run one process for a little bit, then run another (to share the physical CPU)

6.1 Basic Technique: Limited Direct Execution

Limited Direct Execution: Run the program directly on the CPU for maximum speed

- Occurs in 2 phases
  - Kernel initializes trap table, CPU remembers trap table location for later use
  - Kernel sets up things and uses return-from-trap instruction to start execution of process (switching CPU to user mode)

- Issues:

- How do OS make sure that program doesn’t do anything we don’t want it to do?
    - Implement User Mode: Code is restricted
        - User uses system calls to perform privileged operations
            - System calls use traps to simultaneously jump into kernel and raise privilege level to kernel mode
                - Maintains registers using kernel stack
                - Uses trap table to refer process to trap handlers to work inside of the kernel
                - Traps are privileged code
            - Return-from-trap instructions returns into calling user program while reducing privilege level back into user mode
            - System-Call Number assigned to each system call to specify exact call
                - Acts as a protection so users cannot access exact kernel addresses
    - OS runs in Kernel Mode
- How do we stop a process from running and switch to another process?
    - Problem: If process runs on CPI, OS is not running
        - Cooperative Approach
            - OS trusts processes of system to behave reasonably (give up CPU periodically using system calls or yield system calls)
            - Applications transfer control to OS when they do illegal things (by generating a trap)
        - Non-Cooperative Approach (OS takes control)

            - Cooperative Approach only fixable with machine reboot
            - Use timer interrupt to halt process and run OS interrupt handler (a privileged instruction)
    - Scheduler: Decides whether to continue running current process or switch to different one after OS regains control
        - If switch, OS executes a context switch, which is storing a few register values and restore register values for new process
            - Swaps stack pointer to kernel stack of new process
        - Two Types of register saves/restores
            - User registers of running process are implicitly saved by hardware
            - Kernel registers are explicitly saved by software

6.4 Worried about Concurrency?

OS disables interrupts while processing interrupts

- Can lose interrupts

OS utilizes locking mechanisms to protect concurrent access to internal data structures

Linking and Libraries

The Software Generation Tool Chain

- Files
    - Source Modules
        - Text in programming languages not yet converted into machine language(human code)
    - Relocatable Object Modules

        - Compiled/Assembled instructions of source modules. Not yet a full program (ex: .o files)
    - Libraries
        - Collections of object modules, can fetch functions to be used in object modules
    - Load Modules
        - Complete programs, often combinations of Object Modules ready to be loaded into memory by OS and executed using CPU
- Software Tools
    - Compiler:
        - Reads source modules, parses input and generates lower level code
            - Generally Lower Level Code is in Assembly for portability, but some languages use their own low-level pseudo language
    - Assembler
        - Assembly translates code into machine language instructions
            - Allows for variables, macros, and regerences
            - Contains routines
                - User-Mode Code:
                    - Routines that implement calls into OS, Data Structure Manipulation
                - OS:
                    - CPI initialization, 1st level trap handling, synchronization operations

            - Non-Completeness derived from missing addresses and mappings from PM to VM
    - Linkage Editor
        - Reads specified set if object modules, places them consecutively into virtual address space, notes where each one is placed
        - Resolves unresolved external references by looking for object modules in libraries
    - Program Loader
        - Examines load modules, creates virtual address space, reads instructions and data values from load modules into virtual address space
- Object Modules
    - Program “fragments” which can be pieced together to form a program
        - Allows for code reusability
        - Modules can be incomplete (refer to code outside of module)
            - Module addresses are relative to the module
        - ISA-Specific
- Libraries
    - Collection of related object modules
    - Libraries are not always independent
        - Libraries can implement lower level libraries
        - A given function can have multiple implementations
            - Thus, order of library searching is important
- Linkage Editing

    - Linkage Editing: Process to turn object modules into a runnable program
        - Resolution: Search libraries to find object modules to solve unresolved external references
        - Loading: Place data segments in single virtual memory address
        - Relocation: Update loaded object module addresses with new address
- Load Modules:
    - Are complete
        - Requires no relocation
    - Can contain a symbol table to catch exceptions
    - Usage:
        - OS determines data segments and locations based on load module
        - OS allocates space in VM, loads load module into memory
            - Creates stack and stack pointer

Static vs. Shared Libraries

- Static Library: Everything is stored locally in the load module
    - Cons:
        - Many copies of same library, inefficient in terms of time and memory
        - Updating a library is gruesome
- Shared Libraries: Runtime Loadable, shared
    - Pros:
        - 1 Library shared across all programs
        - Because library is not locally stored:

            - New modules can be easily added to library
            - Library can be easily imported in different versions
        - With stub libraries, you can call another shared library from a current shared library
    - Limitations:
        - Shared libraries are read-only and thus cannot store data (needs to store things using the client, not the stack)
        - Shared libraries cannot reference global variables or static calls from client program
        - Entire library is loaded upon request (which uses lots of memory)

Dynamically Loaded Libraries: Libraries that are not loaded until they are needed

- Implicit implementation: Linkage editor automatically loads from DLL as needed (Application is unaware)
    - Reduces size of load modules
- Explicit implementation: Application requests OS to load entire library (is aware)

Stack Frames and Linkage Conventions

The Stack Model: Primarily used in storing local variables (procedure-local variables)

- Space allocation when functions are called
    - Space is only visible by code which called the allocation
    - Each function call has its own version of its variables
- Space is returned when function returns

Subroutine Linkage Conventions

- Consists of:
    - Parameter Passing (into routine)
        - Caller routine cleans parameters off stack
    - Subroutine Call: Save return address (on stack) to return to calling program
        - Callee must clean up callee allocated locals
    - Save register contents of *non-volatile* registers
        - Extra registers are saved
    - Allocating space for local variables of routine

Traps and Interrupts

- Traps: Informs software of execution faults
- Interrupts: Informs software that external event has happened
- Both traps and interrupts:
    - Require control transfer, saving of state, and restoration of state
    - Are initiated by hardware (linkage conventions are defined by hardware)
    - While procedure calls change program state after returning, traps and interrupts maintain program state upon return (as if nothing ever happened)

Week 3

OSTEP Ch 7 (Scheduling: Introduction)

Discipline: Series of Scheduling Policies

7.1 Workload Assumptions

- Workload: Processes running in the system

- Fully-Operational Scheduling Discipline
    - Each job (process) runs for same amount of time
    - All jobs arrive at same time
    - Jobs run to completion
    - All jobs use CPU
    - Run-time of each job is known

7.2 Scheduling Metrics

Scheduling Metric: Turnaround Time

- Turnaround Time: Time it takes to complete a job (from when it starts)
    - Based on performance
    - Fairness is measured using Jain's Fairness Index

7.3 First in, First out / First Come, First Served (FIFO, FCFS)

Runs the first processed job first

- Convoy Effect: Issue of FIFO scheduling where number short consumers are queued behind heavyweight resource consumers

7.4 Shortest Job First (SJF)

Runs the shortest job first

- Still runs into convoy problem assuming longer process is running while shorter processes are received
- Optimizes runtime, bad for response time

7.5 Shortest Time to Completion (STCF)

Preemptive, Optimizes runtime, bad for response time

- Preempting a process: Pause a process to run another one
- Non-Preemptive Scheduling: Jobs run to completion
- Preemptive Scheduling: Scheduler stops one process and begins another
    - Performs a context switch to stop one process and resume another

7.6 Metric: Response Time

Response Time: Time from when job arrives in system to when it is first scheduled

- Previous scheduling policies require jobs to run in entirety before scheduling new processes

7.7 Round Robin

Run jobs for a time slice (scheduling quantum), then switch to next process on queue

- Great for response time, worst for turnaround time
- Focuses on fairness (even distribution of CPU power across processes)

Amortization: Increasing time slice to reduce amount of time context switching

- Cost of time switching is not on OS, but rather the resetting of the state (CPU caches, TLBs, branch predictors, etc)

7.8 Incorporating I/O

I/O does not use CPU

- Overlap: CPU is blocked during I/O, thus we should run a process on CPU while processing I/O

OSTEP Ch. 8 (Scheduling: Multi-Level Feedback Queue)

A good scheduler should optimize turnaround time while minimizing response time

8.1 MLFQ: Basic Rules

MLFQ contains:

- Distinct queues with different priority levels
  - Processes placed on queues, queue priority decides which job runs
  - If queues have same priority, run Round Robin

- Priorities vary based on observed behavior (MLFQ learns about processes, stores history, and predicts future behavior)

8.2 Attempt #1: How to Change Priority

Allotment: Amount of time a job can spend at a given priority level before the scheduler reduces its priority

- Because scheduler does not know how long a job lasts, it initially assumes all jobs are short jobs
- Processes with high I/O will gain high priority due to constantly returning access to CPU

Issues:

- Starvation: Too many interactive jobs will consume the entire CPU, thus long-running jobs will never receive any CPI time
- Gaming the scheduler: Programs can exploit MLFQ rules to gain higher priority

8.3 Attempt #2: The Priority Boost

To address starvation, boost priority of all jobs periodically

- Period to boost is considered a voodoo constant, as there's no good way besides guessing to optimize
    - Ousterhout's Law: Place all voodoo constant in a file for easy modification

8.4 Attempt #3: Better Accounting

To address gaming the scheduler, we better account CPU time at each level of the MLFQ

- Scheduler keeps track of how much allotment process uses, demotes processes if completely use up allotment

8.5 Tuning MLFQ and other Issues

How do we parameterize a scheduler?

- High-priority queues get interactive jobs (and shorter time slices)
- Low-priority queues get long-running jobs (and longer time slices)

Real Time Scheduling

Priority Based Scheduling allows us to give better service to certain processes, it is a best effort approach

- Ex: Video Playing with Audio (Synchronization)

Real-Time Systems: System where correctness depends on timing as well as functionality

- Metrics
  - Timeliness: How closely does scheduler meet its timing requirements
  - Predictability: How much deviation is in delivered timeliness
- Concepts
  - Feasibility: Whether meeting the requirements for task is possible
  - Hard Real-Time: Strong requirements which result in system failure if not met on time
  - Soft Real-Time: Tasks where consequences of missing deadline are degraded performance or recoverable failures
- Parameters:
  - We may know the length of processes
  - Starvation is ok sometimes
  - Workload is fixed

Real Time Scheduling Algorithms

- Static Scheduling: Based on list of tasks and their expected completion times to run, create a fixed schedule to ensure timely execution of all tasks
- Dynamic Scheduling:
    - Different Implementations: How algorithm chooses next task to run, how algorithm handles overload (infeasible requirements)
- Preemption:
    - Pros:
        - Improves mean response time by breaking up long, intensive tasks
        - Prevents buggy programs from taking over CPU
    - Cons:
        - Preemption often means we will miss the completion deadline
        - Static Scheduling often already meets all the requirements, so preemption is not needed
        - Infinite loop bugs are rare due to thorough code testing

OSTEP Ch. 12 (A Dialogue on Memory Virtualization)

Every address generated by a user program is a virtual address

OSTEP Ch. 13 (The Abstraction: Address Spaces)

13.1 Early Systems

Virtual memory did not exist in older machines as older machines used physical memory with physical addresses

13.2 Multiprogramming and Time Sharing

Multiprogramming: OS switches between multiple processes which are ready to run at a given time

- Increases utilization and efficiency of CPU

Time Sharing: Sharing of computer resource across many processes

- Interactivity: Many users concurrently use a machine, each wants a timely response from their currently-executing tasks
- Protection: Each process should only be able to read, write, and access their own memory

13.3 The Address Space

Address Space: Abstraction of physical memory, running program's view of memory in the system

- Code is stored in memory
- Stack keeps track of function call chain, local variables, and passes parameters between values to and from routines
- Heap used for dynamically-allocated, user-managed memory

Virtual Memory: Running program is given illusion of being loaded in memory at an address with a large address space

- Virtual addresses need to be converted to physical addresses

13.4 Goals

- Transparency: Program doesn't know it is running virtual memory, and thus acts like it has its own private physical memory
- Efficiency: Virtualization should be as efficiant as possible in terms of time and space
- Protection: Processes should be protected from each other and the OS should be protected from processes
    - Isolation: Each process can only access its own address space

OSTEP Ch. 14 (Interlude: Memory API)

How do we allocate and manage memory?

14.1 Types of Memory

Stack/Automatic Memory: Allocations/Deallocations managed implicitly by compiler

Heap Memory: Long-Lived Memory, user explicitly handles allocations/deallocations

14.2 The malloc() Call

- Takes inputted bytes needed heap, returns pointer to heap space or NULL
    - Compile-time operator: number substituted in at compile time

14.3 The free() Call

- Frees heap memory at inputted pointer to heap address (not the memory)

14.4 Common Errors

- Proper Memory Management
    - Memory Management handled automatically in new languages
    - Garbage Collectors deletes memory that has no references to it
- Forgetting to Allocate Memory
    - Functions expect memory to be allocated already
        - Often results in segmentation faults
- Not Allocating Enough Memory
    - Called Buffer Overflow
        - Can overwrite important information on computer
- Forgetting to Initialize Allocated Memory
    - Call malloc() with no proper datatype/value
        - Causes uninitialized read, aka reading garbage from the heap
- Forgetting to Free Memory
    - Called a memory leak, occurs when memory is not freed
        - Causes system to run out of memory

    - Note: OS cleans up all leaked memory when process ends
- Freeing Memory before you are done with it
    - Called a dangling pointer
        - Can crash program or overwrite data
- Freeing Memory Repeatedly
    - Called double free
        - Results in undefined behavior
- Calling free() incorrectly
    - Called an invalid free, occurs when inputted pointer is not from malloc()
- Purify and valgrind assist with memory problems

14.5 Underlying OS Support

- Malloc(), free(), are <u>library calls</u>
    - Implemented using system calls
- System call brk: Changes program's break/location of end of heap
- Mmap(): creates anonymous memory region in swap space

<u>OSTEP Ch 15. (Mechanism: Address Translation)</u>

How do we efficiently and flexibly virtualize memory?

Virtualizing CPU → Limited Direct Execution

- Allows efficient virtualization with OS control (through interposing)

<u>Address Translation</u>: Change virtual address to physical address using hardware assistance

- OS manages memory: tracks what's free and what's used

15.1 Assumptions

- Suppose memory is contiguous, address space fits in physical memory, and address space is same size

15.2 An Example

Interposition: Allow OS to modify system calls before execution

15.3 Dynamic (Hardware-based) Relocation

- Address Relocation originally handled by loader software, referred to as static relocation

Dynamic Relocation/Base and Bounds:

- Hardware Registers:
    - Base register
        - OS sets base register to where in system memory process should be loaded
        - Transforms virtual addresses (generated by program) into physical addresses
    - Bounds/limit register
        - Ensures addresses are inside of address space
            - If outside space, raise exception and terminate process
- Address Translation:
    - Hardware takes virtual address and transforms it into physical address
        - Physical address = virtual address + base
        - Dynamic Relocation: OS can change addresses after process runs
    - Primarily handled by Memory Management Unit (MMU)

15.4 Hardware Support: A Summary

- Processor Status Word indicates whether CPU runs in privileged/kernel mode or user mode

- Free List: List of ranges of unused physical memory
- Hardware uses
    - MMU registers (base and bounds) to handle addresses
        - OS changes MMU registers using privileged instructions
    - CPU generates exceptions (illegal memory access)
        - Hands exceptions to OS exception handler

15.5 Operating System Issues

- OS uses free list to find slot of physical memory
- OS places memory back on free list when process terminates
- OS saves and restores base-and-bounds pairs during context switches
    - Saved in process control block
    - OS can move entire virtual address space while process is stopped
- OS possesses exception handler

15.6

Dynamic Reallocation results in:

- Internal Fragmentation: Space inside of allocated unit is not used/is wasted
    - Space between stack and heap are wasted

OSTEP Ch 16. (Segmentation)

How do we support a large address space?

16.1: Segmentation: Generalized Base/Bounds

Segmentation: A base-and-bounds pair is assigned to each logical segment of address space

- Allows segments to be placed in physical memory independently
- Only used memory is allocated space in physical memory
    - Addresses large amounts of unused address space

Segmentation Fault: Memory Access of illegal address

16.2 Which Segment are we referring to?

How does the hardware determine the offset to reach a segment and the contents of an address (to a segment)?

- Explicit Approach: Chop up address space into segments based on top few bits of virtual address
  - Top bits determine segment, bottom bits determine offset
  - Top bits waste space, segments are limited to maximum size
- Implicit Approach: Hardware determines segment location based on how it was created

16.3 What About the Stack?

The stack rows "downwards" towards smaller addresses

16.4 Support for Sharing

Code Sharing: Memory segments are shared to save memory

- Protection bits make a segment read-only and thus sharable

16.5 Fine-Grained vs. Coarse-Grained Segmentation

Coarse-Grained: Address Space split into large chunks

Fine-Grained: Address space spit into many smaller segments

Segment Table assists hardware in creating segments

16.6 OS Support

- OS stores registers for context switches
- OS resizes heap (or rejects resizing request)
- OS manages free space in physical memory

External Fragmentation: Physical memory has many small chunks of contiguous free space scattered around

OSTEP Ch 17. (Free-Space Management)

How should Free Space be Managed?

17.1 Assumptions

For the sake of this chapter, assume

- We are focused on external fragmentation (relating to hardware)
- Memory cannot be relocated (we cannot compact memory)

17.2 Low-level Mechanisms

Splitting and Coalescing

- If we request memory of size smaller than a free chunk,
    - Split → Find large enough free chunk and split it into two
        - First chunk goes to caller, second stays on list
- Coalescing: Merge existing free chunks into single, larger free chunk when freeing memory
    - Otherwise, we have lots of free small chunks which will return a failure if a larger chunk is requested

Tracking the Size of Allocated Regions

- Free() system call only takes pointer as parameter
    - Size of allocated region in memory is stored in header block
    - When N bytes of memory is requested, free chunk of N + header size is searched

Embedding a Free List

- Memory allocation is handled by a free list data structure

Growing the Heap

- If there is no more memory, return NULL or map data onto free physical pages

17.3 Basic Strategies

The ideal allocator is fast and minimizes fragmentation

- There is no best approach

Best Fit

- Return smallest chunk which can fit requested size
    - Pro: Reduces Wasted Space naively
    - Con: Exhaustive search to find correct free block

Worst Fit

- Find largest chunk, return requested amount
    - Cons:
        - Exhaustive search of free space is required
        - Leads to excess fragmentation while still having high overheads

First Fit

- Find first adequate block, return it
    - Pro: Speed (No Exhaustive Search)
    - Con:
        - Pollutes free list with small objects
        - Allocator management of free list causes issues
            - Solution: Address-Based Ordering: Order list by address of free space (to make coalescing easier)

Next Fit

Start looking through free list using extra pointer of last looked-at location

- Pro: Spreads searches more uniformly

17.4 Other Approaches

Segregated Lists: Keep separate list of popular-sized requests and forward them to more general memory allocator

- Pros: Fragmentation is less of concern, no complicated free list searching needed
- Cons: New complications, how big should the separate list be?
    - Solution: slab allocator which allocated object caches
        - Avoids costly initialization and destruction of data structures

Buddy Allocation: Focuses on making coalescing simpler, split free space until block big enough to accommodate request is found

- Pro: When freeing, if "buddy" is free, coalesce recursively
- Con: Internal Fragmentation still exists

Garbage Collection and Defragmentation

Garbage Collection: Seeking out no-longer-used resources and recycling them

Defragmentation: Reassigning and Relocatiing previously allo9cated resources

- Addresses External Fragmentation

Garbage Collection

- Original Implementation
    - Resources freed through explicit function calls, implicit program processes (returning)
        - Not practical due in multiple concurrent client situations, OS not aware of memory allocations
- Garbage Collection Implementation
    - Resources are allocated but never explicitly freed
    - When pool of available resources is small, garbage collect
        - Get list of all resources, scan which ones are reachable, remove reachable resources, free leftovers

    - Often results in huge “stop” in the middle of program running due to garbage collect

Defragmentation

Changes where resources are allocated

- Flash Management:
    - Idea: We can only flush memory in blocks at a time
        - Within a 4K block, copy used data into dense 64MB blocks
        - Once done copying, erase large block and move onto free list
- Disk Space Allocation
    - In recursive function calls (contiguous memory), creation and deletion of files may increase in magnitudes more quickly
        - Allocate a contiguous free space using coalescing
        - Copy files into contiguous space
        - Repeat until used space and free space is contiguous

WEEK 4

OSTEP Ch. 18 (Paging: Introduction)

Paging: Divide process’s address space into fixed-sized units called pages

- Thus, physical memory is an array of fixed-sized slots called page frames
    - Each page frame contains a single virtual-memory page

How do you virtualize memory with pages?

18.1 A Simple Example and Overview

Advantages of paging:

- Flexibility: System can support abstractions of an address space
- Simplicity: Access free data using free list
    - Page Table stores address translations for each virtual page

        - Page Tables are per-process, one exists for each process, each maps to a different physical page
        - Translation of Virtual Address requires:
            - Virtual page number: Defines how many pages to select
                - Used to index into physical frame number
            - Offset: Defines which byte of page we are interested in
                - While VPN changes to PFN, offset remains the same

18.2 Where are Page Tables Stored?

Page Tables can become large → Store Page Tables in Memory

18.3 What's Actually in the Page Table?

Linear Page Table: Page-Table Entries represented in an array indexed using the virtual page number to discover the physical frame number

Contents of PTE:

- Valid Bit: Indicates in translation is valid
    - Unused space marked as invalid, will trap in accessed
- Protection Bits: Indicates read, write, execute permissions of page
    - Invalid permission requests causes traps
- Present Bit: Indicates if page is in memory or on disk
- Dirty Bit: Indicates if page has been modified since brought into memory
- Reference Bit: Tracks whether page has been accessed (Useful in page replacement)

18.4 Paging: Also Too Slow

Page-Table Registers contain the physical address of beginning of the page table

- To do a memory reference, we must access memory twice
    - First is to fetch the translation from the page table

    - Second is the actual memory reference (ex: instruction fetch)

18.5 A Memory Trace

When a process runs:

- Each instruction fetch generates two memory references
    - One to page table (to find physical frame that holds the instruction)
    - One to instruction itself (fetch to CPU for processing)
- Additional explicit memory reference results in another page table access

OSTEP Ch. 19 (Paging: Faster Translations - TLBs)

How do we speed up address translation?

- Idea: Use hardware (part of the MMU specifically)
    - Translation-Lookaside Buffer: Hardware cache of popular virtual-to-physical address translations
        - If virtual translation in TLB, no need to consult page table

19.1 TLB Basic Algorithm

Check for physical address given a Virtual Page Number:

- TLB hit: TLB holds the translation
- TLB miss: Translation not in TLB, hardware accesses page table
    - Requires costly paging to find translation

19.2 Example: Accessing an Array

- A TLB miss causes a page to load → Spatial Locality means the next few TLB requests will hit
    - Hit Rate: Number of hits divided by the total number of accesses
    - Temporal Locality: If a data is accessed, the same variable will likely be re-accessed again

    - Spatial Locality: If a data x is accessed, the data near x will soon be accessed

19.3 Who Handles the TLB Miss?

- In old days, hardware handled TLB misses
    - Locates page table with page-table base register, walks through page table to find correct entry, extract translation and update TLB
        - Implemented using multi-level page table in Intel x86
- These days, software-managed TLB
    - Hardware raises an exception, pausing the instruction stream and raising the privilege level of the kernel to jump into the trap handler
        - Trap handler (software) looks up translation in page table and updates TLB with privileged instructions
    - Notes:
        - TLB miss-handling trap must return to instruction that *caused* the trap (vs after the trap)
        - OS should ensure that TLB misses don't infinitely repeat
        - Pro: Flexibility – Any data structure can be a page table
        - TLB Valid Bit is not the Page Table Valid Bit
            - Invalid Page Table Valid Bit: Page is not allocated by the process and should not be accessed
                - Results in a trap to the OS
            - Invalid TLB valid bit: no address translations are cached in TLB
                - Occurs when the process initially runs

19.4 TLB Contents: What's in there?

- TLBs are normally fully associative: Translations can be stored anywhere inside of the TLB, hardware must search through whole TLB to find it
- Valid bits show if entry has valid translation
- Protection bits determines permissions of a page

19.5 TLB Issue: Context Switches

- Idea: TLBs are per-process; are only valid for their own process. Thus in a context switch, OS can confuse which translation is for which process

How do we manage TLB Contents on a Context Switch?

- Approach: Flush the TLB on context switches (Set all valid bits to 0)
    - Either call explicit hardware instruction (Software) or change page-table register (Hardware)
    - Each time a process runs, TLB miss occurs → Lots of memory accessing
        - Address Space Identifier allows TLB to hold translations from different processes without confusion
- Approach: Share physical page across different VPNs
    - Reduces memory overhead by reducing number of physical pages

19.6 Issue: Replacement Policy

How to we decide which entry to replace in the TLB? (Cache replacement to minimize miss rate)

- Approach: Evict Least-Recently-Used/LRU entry
    - Maximizes locality
- Approach: Randomly evict a TLB entry
    - Simple, no corner-cases

19.7 A Real TLB Entry

Random Access Memory (RAM) means all parts of RAM are equally accessible

OSTEP Ch. 21 (Beyond Physical Memory: Mechanisms)

Hard Disk Drives are big, slow hard drives which have more capacity than memory

- Allows for multiprogramming capabilities (Pages need to be swapped)

21.1 Swap Space

Swap Space: Space on disk for moving pages back and forth between memory

- OS must remember the disk address of a given page
- Creates the illusion for processes that they have infinite space
- Code binaries are also stored in swap space (to be loaded into memory)

21.2 The Present Bit

Present Bits define if a page is present in physical memory, if page is not in memory, it is in disk

- Page Faults are accessing pages that are not in physical memory (are general illegal memory accessing)
  - Page Faults are dealt by the Page Fault Handler

21.3 The Page Fault

OS possesses page-fault handler, thus page faults are handled using software

- Software is used because hardware page faults are slow, and hardware cannot fully grasp swap space

Page Fault:

- If a page is not present and is in disk
  - Use page table to find disk address, update page table and record in-memory location of newly-fetched page
  - During I/O, process is in blocked state

21.4 What if Memory is Full?

Page-Replacement Policy: Decides which pages to replace/kick out

21.5 Page Fault Control Flow

In a request for a translation in a hardware implementation, there are three cases:

- Page is both present and valid: grab PFN from PTE, retry instruction
- Run the Page Fault Handler: Page not present in physical memory
- Access of Invalid Page: Hardware throws a trap, OS trap handler runs and likely terminates the process

If it is a software implementation:

- OS finds physical frame to hold newly-faulted page
    - If there is no such page, run replacement algorithm to create room
- Handler issues I/O request to read in page from swap space
- After reading completes, OS updates page table and retries instruction

21.6 When Replacements Really Occur

- Replacement is not a process that runs when almost out of memory
- Instead, a swap/page daemon runs in the background:
    - High watermark: Indicates the minimum amount of pages replacement should free
    - Low Watermark: Indicates the minimum threshold of free space needed to initiate replacement

OSTEP Ch. 22 (Beyond Physical Memory: Policies)

How do we decide which page to evict?

22.1 Cache Management

- Because memory is temporary storage, consider it a cache
    - We want to minimize the amount of cache misses/maximize the number of cache hits

        - Calculate Average Memory Access Time

22.2 The Optimal Replacement Policy

Optimal Replacement Policy: Throw out the page that will be accessed the latest in the future (to reduce cache misses)

- Compulsory/Cold-Start Miss: Initial cache miss when process begins
- Capacity Miss: Cache ran out of space and had to evict an item to make room
- Conflict Miss: Issue due to limitations on where an item can be placed in a hardware cache

22.3 A Simple Policy: FIFO

- Pages placed in a queue, tail of queue is evicted
- Con: FIFO cannot determine the importance of blocks

22.4 Another Simple Policy: Random

- Random page is evicted
- Con: Inconsistent, but generally a little bit better than FIFO

22.5 Using History: LRU

Issue: Replacement Policies evict important pages

- Solution: Principle of Locality: Observe page frequency (how many times its accessed) and recency (how recently it was accessed)
    - Spatial Locality: If page is accessed, pages around said page are likely to also be accessed
    - Temporal Locality: Pages that are recently used are likely to be used again

Least Frequently Used (LFU) Policy: Replaces least-frequently used page

Least Recently Used (LRU) Policy: Replaces least recently used page

22.6 Workload Examples

- No locality:
    - LRU, FIFO, Random all perform same (Hit rate determined by the size of the cache)
- 80% of references are made to 20% of the pages:
    - LRU performs better but not optimally
- References to 50 pages in sequence
    - Random performs best (due to having a non-zero hit rate)
        - This is worst case for FIFO, LRU

22.7 Implementing Historical Algorithms

- Without Hardware:
    - Page must be move to front of list using data structure after every page access
- Using hardware assistance:
    - Update memory field when page is accessed, use OS to determine least recently used page
        - Unsustainable for lots and lots of pages

22.8 Approximating LRU

Use Bit: Exists in every page, updates to 1 by hardware when page is referenced

Clock Algorithm: Clock hand points to some page. If Page needs to be replaced, if use bit is equal to 1 (not a good candidate as used recently), update bit to 0 and move to next page. Repeat until page is found with use bit of 0

- Less optimal than perfect LRU
- The most recently examined pages are behind the hand
- Pages examined longest ago are immediate in front of hand

22.9 Considering Dirty Pages

Modified Bit: Set any time a page is written, used to check cleanliness

- Clean pages have not been modified since being written from disk
    - Eviction is free (physical frame can be used for other purposes)
- Dirty Pages have been modified since being written from disk, and thus require a write to disk to be evicted

22.10 Other VM Policies

Page Selection Policy: When to bring a page into memory

- Demand Paging: OS brings page into memory when accessed "on demand"
    - OS prefetches data data that will be used ahead of time
- Clustering: Writing large groups of files into disk (rather than one at a time)

22.11 Thrashing

What should the OS do if it literally does not have enough physical memory?

- Old Approach:
    - Admission Control: Reduce working sets (pages being used) in memory by doing less work at a time
- New Approach:
    - Out-Of-Memory Killer: Daemon chooses and kills a memory-intensive process

Working Set Page Replacement

LRU does not work in Round-Robin approaches

- Destroys explicit temporal and spatial locality
    - Least recently used page will run next
    - Most recently used page will not run for a long time

Working Set Size: Amount of pages which will not change performance when increased but drastically slows performance when decreased

- Size is based on the behavior of the process (can be small or big)

Working Set Implementation

- Each page frame is associated with an owning process
- Each process has an accumulated CPU time
- Each page frame has a last referenced time
- Target Age Parameter: Memory goal for all pages

Notes:

- Pages only age when not referenced by owner
- If page is younger than target age, don't replace (to avoid thrashing)
- If page is older than target age, take away from owner, give tot new process

  Page Stealing Algorithm: Dynamic equilibrium mechanism which allocates available memory to the running processes in proportion to their working sizes

OSTEP Ch. 25 (CONCURRENCY)

OSTEP Ch. 26 (Concurrency: An Introduction)

Thread: Point of entry for a process

- Threads share the same address space and can access the same data
- Each thread has its own private registers
  - Context switches between threads use Thread Control Blocks (TCBs)
  - Each thread runs independently from each other
    - Each thread has a local stack → Thread-Local Storage

Multithreaded Program: Program with multiple points of entry

26.1 Why Use Threads?

- Parallelism: Split work between multiple CPUs
- Avoid blocking program progress due to slow I/O
  - Overlap of I/O with other activities within a single program (Taking advantage of sharing an address space using threads)

26.2 An Example: Thread Creation

- Threads can operate in different orderings
    - Main thread must wait for child threads to complete before continuing

26.3 Why it Gets Worse: Shared Data

Programs should generally produce deterministic results

Disassembler: Shows what assembly instructions make a program

26.4 The Heart Of the Problem: Uncontrolled Scheduling

- Race Condition: Results depend on the timing of the code's execution (Indeterminate), when two threads enter critical section and update data structure at the same time
    - Critical Section: Piece of code that accesses a shared variable but should not be concurrently executed by more than one thread
    - Mutual Exclusion: If one thread executes within critical section, others will be prevented from doing so
- Atomic Sections: "All or nothing"
    - Transaction: Grouping of actions into a single atomic section

26.5 The Wish For Atomicity

- Synchronization primitives: Hardware instructions to assist with outputting deterministic output

26.6 One More Problem: Waiting For Another

- Threads can wait for another thread to finish

OSTEP Ch. 27 (Interlude: Thread API)

How do we create and control threads?

27.1 Thread Creation

Threads in C take 4 Arguments:

- Thread: pointer to thread, used to interact with thread
- Attr: Specifies any attributes thread has
- Start_routine: Function pointer to function to be multithreaded
- Arg: Arguemnts to the function pointed in start_routine

27.2 Thread Completion

- Pthread_join: Waits for a thread to complete, 2 parameters
    - Thread: Specifies which thread to wait for
    - Value_ptr: Pointer to return value expected to get back
        - Routine changes the value of the passed in argument (why you pass in a pointer and not the value)
- Notes:
    - Avoids painful packing and unpacking of arguments
    - Passing in a single value does not require packaging it up as an argument
    - Never return a pointer that refers to something allocated on the thread's stack call
- Procedure Calls: Creates a thread

27.3 Locks

Lock: Synchronization primitive that prevents state from being modified or accessed by multiple threads of execution at once

Common Issues:

- Lack of Proper Initialization: All locks must be properly initialized in order to guarantee that they have the correct values to begin with
- Failure to check error codes when calling lock and unlock: If routine fails, then multiple threads could enter a critical section

27.4 Condition Variables

Condition Variable: Allows threads to wait for certain events to occur, commonly associated with a lock

Notes:

- We must hold a lock to send a signal (to avoid race conditions)
- Wait call takes in a lock as a second parameter (it puts thread to sleep and releases the lock)
- Waiting thread checks condition using a while loop because it is a safer implementation (compared to an if statement)

User-Mode Thread Implementation

Processes:

- Used for execution
- Bad for Parallelism
    - Processes are very expensive to create and dispatch
    - Each process operates in its own address space and cannot share

Threads

- Allow programs to be parallel
    - Independently schedulable unit of execution
- Shares same address space of a process
    - Has access to all resources available to its process
- Has registers, stack unique to each thread

General Thread Use (No OS implementation):

- Allocate memory for thread-private stack from the heap
    - Create new thread descriptor that contains identification, other stack stuff
    - Add thread to queue

- Usage of signals to allow for preemprive handling

Kernel Implemented Threads

- Generally not a good idea
    - If thread blocks, all threads stop, OS has no knowledge of threads because they were implemented in user-mode
    - Because OS is not aware process is multiple threads, it cannot assign the threads onto available cores

Parallel Implications

- Non-preemptive → More efficient than context switches
- Preemptive Scheduling → Cost of setting signals is greater than OS operating scheduling
- Threads running in parallel on multi-processor: Throughput is greater than the efficiency losses associated with context switches
- Signal disabling can be more expensive than using kernel implementations

Inter-Process Communication

Process Interactions:

- Coordination of Operations with other processes
    - Synchronization, exchanging signals, control operations
- Exchange of data between processes
    - Uni-directional data processing pipelines, Bi-directional interactions

Simple Uni-Directional Byte Streams (Literally pipes):

- Each program accepts a byte-stream input, produces a byte-stream output that is well defined
- Pipes are temporary files which recognize the difference between a file and an inter-process data stream

    - If reader exhausts all of the data in the pipe, the reader is blocked until more data becomes available
    - Flow Control: OS blocks the writer until reader catches up
    - If read and write file descriptors are both closed, the file is automatically deleted

Named Pipes and Mailboxes

- Named Pipe: Persistent pipe whose reader and writers can open it by name
    - Readers and writers cannot authenticate each other's identities
    - Writes from multiple writers may be interspersed (No clue which changes were from who)
    - All readers and writers must run on same onde
- Mailbox: Special memory location that one or more tasks can use to transfer data
    - Data is not a byte-stream but rather a message
    - Each write is paired with identification information about its sender
    - Unprocessed messages remain in mailbox after the death of a reader

Shared Memory: fastest way to move data between two processes

- Each process maps a file into its virtual address space
- Communicating processes agree on data structure in the shared segment
- Issues:
    - Processes must be on same memory bus
    - Bug in one process can destroy the other process
    - No authentication of which data is from what process

Week 5

OSTEP Ch. 28 (Locks)

Locks allow critical sections to operate as an atomic instruction

- Either available or acquired
- Called a mutex in POSIX

28.1 Locks: The Basic Idea

Lock(): Acquires the lock and enters critical section if no other thread holds the lock

Unlock()

Free lock: Either change lock state to free or transfer lock to next waiting thread

28.2 Pthread Locks

Coarse-Grained Locking Strategy: One big lock used any time any critical section is used

Fine-Grained Locking Approach: Use different locks to allow more threads to be in locked code at once (and protect different data and data structures)

28.3 Building a Lock

How do we build an efficient lock? → OS & Hardware

28.4 Evaluating Locks

Basic Functionality

- Mutual Exclusion: Prevents multiple threads from entering a critical section
- Fairness: Does any thread starve while waiting for the lock (never obtain it)
- Performance: Overhead of grabbing and releasing locks, performance of multiple waiting threads, performance with multiple CPUs

28.5 Controlling Interrupts

Idea: Disabling Interrupts to create mutual exclusion for single-processor systems

- Pro: Simple approach to create mutual exclusion
- Cons:
  - Requires thread to call privileged instructions

        - We must trust thread in design (No loops or greedy CPU use)
    - Does not work with multiprocessors (Multiple threads can access the same critical section)
    - Interrupts can become lost when interrupts are disabled for long periods of time

28.6 A Failed Attempt: Just Using Loads and Stores

Idea: Use variable flag and thus CPU hardware to create mutual exclusion

- Threads use set tests to hold lock, clear flag to stop holding lock
- Waiting threads will Spin-wait while waiting for flag to clear
    - Spin-waiting: Repeatedly asking if the lock is unlocked yet
- Cons:
    - Correction: Mutual Exclusion is not guaranteed (Two threads can access a critical section through interleaving)
    - Performance: Spin-waiting eats up processor time

28.7 Building Working Spin-Locks with Test-and-Set

Idea: Implement hardware support for locking (test-and-set instruction/atomic exchange)

- Test-and-Set: Simultaneously returns old value while updating said value (in memory) atomically
    - Used to implement spin locks on a preemptive scheduler for single processors

28.8 Evaluating Spin Locks

- Correctness: Yes, provides mutual exclusion (only single thread enters critical section at a time)
- Fairness: Not fair (A thread can spin forever)
- Performance:

    - Multiple threads waiting for thread: Waste of CPU cycles to iterate through all threads
    - Multiple CPUs: Checks for threads happen simultaneously

28.9 Compare-And-Swap

Compare-And-Swap: If the value of inputted pointer is same as expected, update memory of pointer to new value. (Otherwise do nothing)

28.10 Load-Linked and Store-Conditional

Load-Linked Instruction: Fetches a value from memory, places it in register

Store-Conditional Instruction: Only succeeds (in updating memory value at load-linked address) if no intervening store to the address has taken place

- Can work through interrupts (as one thread's store-conditional will success and the other will fail)

28.11 Fetch-And-Add

Fetch-And-Add: Atomically increments a value while returning the old value at a particular address

- Ticket Lock: Each waiting thread is assigned a ticket declaring the thread's "turn" unlocking is incrementing the ticket number
    - Ensures process for all threads as they are all scheduled

28.13 A Simple Approach: Just Yield, Baby

How to Avoid Spinning?

Hardware → Allows for working locks and fairness

- Does not account for context switches in critical section and infinite spin waiting

Yield(): Moves thread status from running to ready, effectively descheduling itself (Thread gives up the CPU and lets another thread run)

- Can result in many context switch calls in 100 threads case, inefficient, does not address starvation

28.14 Using Queues: Sleeping Instead of Spinning

Give OS a queue and more power to schedule threads (rather than solely the scheduler)

Solaris Implementation:

- Park(): Places a thread to sleep
  - Sleep when lock is held
- Unpark(): wakes up a thread based on its threadID
  - Wake up thread if lock is free
- Notes:
  - Flag does not update when thread wakes up: Lock is simply handed to next thread without flag update
  - Wakeup/Waiting race: A thread releases the lock while another thread is in the middle of sleeping to wait for the same lock

Priority Inversion: Spin lock correction issue, Higher priority thread deschedules lower priority thread but is unable to receive the lock (because the lower priority thread still holds it)

- Priority Inheritance: Temporarily boost lower-priority thread's priority
- Other solution: Make all threads have the same priority

28.15 Different OS, Different Support

Linux Futex: Similar to Solaris implementation, but more kernel functionality

28.16 Two-Phase Locks

Two-Phase Lock: Lock spins in phase 1 to wait for lock, if lock not acquired, enter second phase to place caller to sleep, wake up when lock becomes free

- Linux Lock only spins once

OSTEP Ch. 30 (Condition Variables)

How do we Wait for a Condition?

30.1 Definition and Routines

Condition Variable: Explicit queue storing threads waiting for a condition (as the condition is not desired as of now)

Wait(): Executed when Thread want to put itself to sleep

- Takes mutex parameter (which is locked when wait() is called)
- Hold the lock when calling wait (it is assumed)

Signal(): Executed when thread has changed something in the program and wants to wake sleeping threads on a condition

- Hold the lock when calling signal

Race Conditions:

- Child signals with no parent asleep on condition → No Thread ever wakes parent

OSTEP Ch. 29 (Lock-Based Concurrent Data Structures)

How do we add locks to data structures for correction? (And Concurrency)

- More concurrency does not mean better performance

29.1 Concurrent Counters

- Adds a single lock, acquired when calling a routine that manipulates the data structure and released from returning from the call
- Performs poorly
  - Perfect Scaling: Benchmark for performance, threads complete at same speed on multiple processors as single thread does on one

Scalable Counting

- Approximate Counter: Implemented using global counter and local physical counters (one per CPU core)

    - Lock on global counter and each local counter
    - Update each local thread without contention, update global counter as needed
        - Updates based on parameter S, higher S means more scalability because less updates

29.2 Concurrent Linked Lists

Malloc() errors (inability to allocated memory) breaks the concurrent counter

- Solution: Rearrange code to surround critical sections with lock and release, use common exit path in lookup code
    - Works because inserts do not need to be locked

Scaling Linked Lists

Hand-Over-Hand Locking: Implement locks for each node in the list (rather than one for the whole list), while traversing, grab next node's lock while releasing current node's lock

- While concurrent, more locks means slower performance

29.3 Concurrent Queues

Two locks: One on head of queue and one on tail of queue

- Concurrency occurs in enqueue (tail lock) and dequeue (head lock)

29.4 Concurrent Hash Table

Constraint: Assume Hash Table does not resize

- Implemented using concurrent lists, a lock handles a hash bucket

OSTEP Ch. 30 (Condition Variables, the rest of it)

30.2 The Producer/Consumer (Bounded Buffer) Problem

Producers generate data items that are placed into a buffer, Consumer grab items from buffer and consumes them

Put() assumes shared buffer is empty and inserts a value into a buffer and marks it full (updates the buffer)

Get() sets buffer to empty, reads from buffer

Problems:

- Suppose there are unequal amounts of producer and consumer threads, if the shared buffer updates in between one consumer thread's awakening but before another consumer threads running, we run out of buffer to call get() with
- Mesa Semantics: Signals only wake threads, cannot assume state is desired
- Hoare Semantics: Provides stronger guarantee that woken thread runs immediately upon waking up

Better but still broken: While, not if

- While loops are considered safer
- Can cause all threads to sleep in rare cases

The Single Buffer Producer/Consumer Solution

- Use two condition variables to properly signal which type of thread should wake up when the state of the system changes
  - Producer threads wait on condition empty, signals fill
  - Consumer threads wait on fill, signals empty

The Correct Producer/Consumer Solution

- Add more buffer slots: Multiple values can be produced and consumed before sleeping
- Producer only sleeps if all buggers are currently filled
- Consumers only sleep if all buffers are currently empty

30.3 Covering Conditions

How do we select which thread should be waken up (to use free memory)

- Covering Condition: Wake up all threads, check condition, go to sleep
  - Negative performance, as can wake up threads that should be awake

OSTEP Ch. 31 (Semaphores)

31.1 Semaphores: A Definition

Initial value determines behavior

- Second parameter = 0 → Semaphore is shared between threads in the same process
- Third parameter → Value of semaphore

Calls:

- Sem_wait(): returns right away if value is one or higher, suspends execution otherwise
- Sem_post(): Simply increments value of semaphore and wakes thread if needed
- If value of semaphore is negative, it is equal to the number of waiting threads

31.2 Binary Semaphores (Locks)

Holding lock: semwait() called but sem_post not yet called

31.3 Semaphores for Ordering (as an ordering primitive)

Parent calls sem_wait and child sem_post to wait for the condition of the child finishing its execution to become true

- Initial value of semaphore should be set to 0
  - Allows for parent to wait for child. To run
  - Accounts for if child runs to completion before parent calls sem_wait

31.4 The Producer/Consumer (Bounded Buffer) Problem

First Attempt:

- Two Semaphores: empty and full

- Issue: Can overwrite old data from producer

Solution: Adding Mutual Exclusion

- Lock the filling of buffer and incrementing of index into buffer using binary semaphores
    - Causes Deadlock:
        - Consumer holds mutex and waits for waits for someone to signal *full*
        - Producer could signal *full* but is waiting for the mutex

A Working Solution:

- Reduce the scope of the lock (move the mutex acquire and release outside of the critical section)

31.5 Reader-Writer Locks

Idea: There should be a more flexible lock that works for different kinds of data structures

- <u>Reader-Writer Lock</u>: Reads the data structure and allows lookups to proceed concurrently if there is no on-going insert
    - Acquiring a read lock
        - Reader acquires lock, readers variable is incremented (to count how many readers are in the structure)
            - Write lock must wait until all readers are finished
    - Cons:
        - Not fair
        - Starvation of writers
        - Adds overhead and does not speed up performance compared to locking primitives

31.6 The Dining Philosophers

Problem:

- 5 philosophers sit around a table
    - 1 fork in between each pair of philosophers
- In order to eat, a philosopher must grab two forks

Broken Solution:

- To acquire forks, we simply grab a "lock" on each one, once we finish eating, we release them
    - Each philosopher grabs the fork on their left, and then their right
- Issue: Deadlock
    - All forks can be acquired (each philosopher grabs 1 fork) but nobody can eat

Solution: Breaking the Dependency

- Change the behavior in which forks are required
- The last philosopher grabs the opposite: right then left

31.7 Thread Throttling

How can a programmer prevent too many threads from doing something at once?

- <u>Throttling</u>: A form of <u>admission control</u> where we create a threshold for too many, use a semaphore to limit the number of threads concurrently executing the code

31.8 How to Implement Semaphores

<u>Zemaphore</u>: Semaphores which do not follow the idea that a negative semaphore value reflects the number of waiting threads

- The zemaphore value will never be lower than 0

Post Midterm Material

Week 6

OSTEP Ch. 32 (Common Concurrency Problems)

How do we handle Common Concurrency Bugs?

32.1 What Types of Bugs Exist?

Lu et al Study: Analysis of Open-Source Applications (MySQL, Apache, Mozilla, MS Office Suite) to examine causes and fixes for concurrency bugs

32.2 Non-Deadlock Bugs

Atomicity and order violations account for 97% of Non-Deadlock Bugs

Atomicity Violation Bug: When desired serializability among multiple memory accesses is violated

- A code region is meant to be atomic, but the atomicity is not enforced during execution
- Solution: Add locks around shared variable references

Order Violation Bug: The desired order between two (groups of) memory accesses is flipped

- Solution: Enforce ordering, apply condition variables to synchronize code

32.3 Deadlock Bugs

Deadlock: One thread holds a lock 1 while waiting for lock 2, while another thread holds lock 2 but waits for lock 1

- Generally represented graphically as a cycle

Why do Deadlocks Occur?

- Complex dependencies arise between components of large code bases
- Encapsulation contributes to complications with applying locking (The idea is to hide the details of implementations)

Conditions for Deadlock

- Mutual Exclusion: Threads claim exclusive control of resources that they require
    - Thread grabs a lock
- Hold and Wait: Threads hold resources allocated to them while waiting for additional resources
    - Threads hold locks that they've already acquired while waiting for locks they wish to acquire
- No preemption: Resources cannot be forcibly removed from threads that are holding them
- Circular Wait: There exists a circular chain of threads such that each thread holds one or more resources that are being requested by the next thread in the chain

Prevention:

- Addressing Circular Wait
    - Write code that never induces a circular wait
        - Total Ordering on lock acquisition (strict ordering on all locks)
            - Use addresses for ordering of lock acquisition
        - Partial Ordering: Total Ordering which does not apply to all locks
- Addressing Hold-and-wait
    - Acquire all locks at once atomically
        - Require possession of global prevention lock
            - Does not work well with encapsulation
            - Decreases concurrency (all locks must be acquired early on at once versus when they are needed)
- Addressing No Preemption

    - Build a routine which when attempting to grab a lock, either returns success or returns an error (indicating the lock is held)
    - Livelock: Two processes attempt to simultaneously acquire the lock, but both repeatedly fail
        - Solution: Add random delay before looping back
            - Avoids hard parts of trylock approach
- Addressing Mutual Exclusion
    - Idea: Design data structures without locks at all (thus removing explicit locking)
        - Utilize compare-and-swap to repeatedly try to update value to new amount
- Deadlock Avoidance via Scheduling
    - Utilize global knowledge of which locks which locks threads may request, subsequently schedule threads to avoid an deadlock occurring
    - Sacrifices performance for reassurance deadlock cannot occur
- Detect And Recover
    - Allow some deadlocks to occur and apply recovery techniques as needed
        - OS freezes → Reboot system

Deadlock Avoidance

There are some cases where we cannot avoid mutual exclusion, hold and block, preemption, or resource dependency

- Thus we avoid these cases using Deadlock Avoidance

Reservations: Declining to grant requests that would put the system into an unsafe, resource-depleted state

Over-booking: It is relatively safe to grant more reservations thn we actually have the resources to fulfill

- Not all clients simultaneously request their maximum resource reservations
- More work is done with the same resources
- Sometimes there is demand which we cannot gracefully handle

Dealing with Rejection: Process attempted to maange situation in most graceful possible way

- Log an error, retry request, return error but serve new requests, attempt to reduce resource use, etc.

Health Monitoring and Recovery

Formal Deadlock Detection:

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

Health Monitoring: We detect if the system makes progress

- Detecting system progress
    - Internal monitoring agent watches message traffic
        - Might be able to monitor logs or statistics that determine service is running at reasonable rate, but if the internal monitoring agent fails, no more detections can occur
    - Clients submit failure reports to central monitoring service if server becomes unresponsive

        - Can determine whether or not monitored application responds to requests, but does not mean other requests have not been deadlocked
    - Each server sends periodic heart-beat messages
        - Heartbeat messages only tell if application is running, but not if application is serving requests
    - External health monitoring service sends periodic test requests to the service being monitored
        - Can determine whether or not monitored application responds to requests, but does not mean other requests have not been deadlocked
- Real Implementations are a combination of methods
    - First line of defense: Internal Monitoring Agent
    - Central Monitoring agent can detect failure of internal monitoring agent (assuming internal agent sends heartbeat signals to central)
    - External test service periodically generates test transactions to provide independent assessment

Managed Recovery: Software designed to kill itself/restart to different degrees

- Examples of restarts:
    - Warm-Start: Restore last saved state
    - Cold-Start: Ignore any saved state
    - Reset and Reboot: Reboot the entire system and then cold-start all of applications
- Restarts progress
    - Restart 1 process → Restart groups of processes → Restart node → Restart groups of nodes

False Reports

- We cannot assume that a process fails when a single heartbeat is skipped
    - Failing a process is expensive
    - Solution: Use two skipped heartbeats
        - Avoid Unnecessary disruptions
- Trade Offs:
    - Misdiagnosis makes the problems even worse
    - Waiting too long to initiate fail-overs prolongs service outage

Other Managed Restarts

- Non-Disruptive Rolling Upgrades: If system can run without some nodes, update software by taking some nodes down and upgrading each node one-by-one
- Prophylactic Reboots: Automatically restart system at regular intervals (to avoid slower software from running for long periods of time)

Week 7 (PART III: Persistence)

OSTEP Ch. 36 (I/O) Devices)

How ais I/O integrated into systems?

36.1 System Architecture

- Memory Bus
    - Attaches to CPU
        - CPU attaches to I/O chip via proprietary DMI (Direct Media Interface)
- I/O Bus
    - PCI (Peripheral Component Interconnect Express), graphics, higher-performance I/O

- Peripheral Bus
    - SCSI, SATA, USB
        - Storage interfaces: eSATA, ATA, SATA

The faster a bus, the shorter it should be (higher performance components are closer to the CPU)

36.2 A Canonical Device

Devices are composed of:

- Hardware Interface
- Internal Structure: Implementation specific, responsible for implementing abstraction device presents to the system
    - Implemented with hardware chips and firmware (software in a hardware device)

36.3 The Canonical Protocol

Device Interface is composed of:

- Status Register: Read to see current status of device
- Command Register: Tells device to perform a certain task
- Data Register: Passes Data to the device

Protocol Procedure:

- Polling: OS waits until device is ready to receive a command by repeatedly reading the status register
- OS sends data to data register
    - If main CPU is involved, it is referred to as programmed I/O (PIO)
- OS writes command to command register
    - Lets device know that data is present and that it should work on inputted command
- OS waits for device to finish by polling it in a loop

Inefficiencies with the Protocol:

- Polling is inefficient (wastes CPU time waiting rather than switching to another process)

36.4 Lowering CPU Overhead with Interrupts

How to Avoid the Costs of Polling?

- Interrupts: OS issues a request, places calling process to sleep, and context switches to another task
  - When device is finished, raise a hardware interrupt to make CPU jump into the OS at a predetermined interrupt service routine (ISR) / interrupt handler
  - Allows for overlap of computation and I/O
- Interrupt Handler: piece of OS code that finishes request, wakes waiting process (for I/O), then proceeds

Two-phased approach: Hybrid model that polls, then interrupts

Issues with Interrupts:

- Interrupts mainly work on slower devices
  - Can result in more expensive/slows down when tasks are completed very quickly
- Can cause OS to livelock in network environments
  - Livelock: OS only processes interrupts and never allows user-level process to run and complete requests

Coalescing: Device (which will call an interrupt) waits while delivering interrupt to CPU

- Waiting allows other requests to coalesce into single interrupt delivery
  - Reduces overhead of interrupt processing

36.5 More Efficient Data Movement with DMA

Using PIO to transfer large chunk of data to device → CPU is overburdened

How to lower PIO Overhead?

- Direct Memory Access (DMA): System device that organizes transfers between devices and main memory without much CPU intervention
    - OS programs DMA engine
        - Notes where data lives in memory, how much data to copy, which device to send it to
        - DMA controller raises an interrupt when complete

36.6 Methods of Device Interaction

How does the OS communicate with a device?

- Explicit I/O Instructions: Specifies way for OS to send data to specific device registers
    - Allows for construction of protocols
    - Privileged instruction
- Memory-Mapped I/O: Hardware makes device registers available as memory locations
    - OS issues a load or store to the address, hardware routes load/store to device instead of main memory
    - No new instructions are needed to support it

36.7 Fitting into the OS: The Device Driver

How to build a device-neutral OS:

- Approach: Abstraction
    - Device Driver: OS software which knows in detail how a device works
    - Raw Interface: Enables special applications to directly read/write blocks without file abstraction

36.8 Case Study: A Simple IDE Disk Driver

- Interface: (accessed through I/O addresses)
    - Control
    - Command Block
    - Status
    - Error
- Protocol:
    - Wait for drive to be ready
    - Write parameters to command registers
    - Start the I/O
    - Data Transfer
    - Handle Interrupts
    - Error Handling

<u>Device Drivers: Classes and Services</u>

Device drivers generalize and simplify abstractions

Updating device driver functionality:

- Deriving device-driver sub-classes for each of the major classes of the device
- Defining sub0class specific interfaces to be implemented by the drivers for all device in each of those classes
- Creating per-device implementations of those standard sub-class interfaces

Major Driver Classes

- <u>Block Devices</u>:
    - Implements a *request* method to enqueue asynchronous DMA requests
        - Large requests are broken into multiple single-block requests
    - Used within OS (by file systems to access disks)
- <u>Character Devices</u>:

    - Supports standard synchronous operations (read(), write(), seek())
        - Can be sequential access of byte-addressable
    - Used directly by applications
        - Large DMA transfers between user-space buggers and device might be more efficient that block device requests

Driver Sub-classes

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

<u>Device Driver Interface (DDI)</u>: Sub-class interfaces

- Pros
    - Sub-class drivers are easier to implement as functionality is implemented in higher level software
    - System behaves identically over range of different devices
    - Most functionality enhancements in higher level code work on all devices within sub-class
- Cons
    - If driver does not correctly implement standard interfaces for device sub-class, it will not work with higher level software
    - If driver implements additional functionality, those features will not be exploited by the standard higher level software

Device Driver Requriements:

- dynamic memory allocation

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

Dynamically Loadable Kernel Modules

Standard Design Patterns:

- All implementations have similar functionality based on a common interface
- Selection of particular implementation can be deferred until run time
- Decoupling the overarching service from the plug-in implementations makes it possible for a system to work with a potentially open-ended set of algorithms or adaptors

Dynamically Loadable Modules: Implementations selected and loaded at run time

- Ex: Browser plug-in
- Motivations
    - Number of I/0 devices is too much to build all at once
    - OS should automatically identify and load required device drivers to operate automatically
    - Many devices are hot-pluggable (we don't know what they are until they are plugged in)
    - Most device drivers are developed by hardware manufacturers and delivered independently from OS

Factory: Obtains an implementation

- Approaches to selection:

    - Usage of MIME-type associated with data to be handled (assumes data is tagged with a type)
    - Factory loads all known plug-ins, calls a *probe* method to see which plug-in fits the job
        - Can select wrong device
        - Touches registers in random devices

Loading new Modules:

- If modules requires external functions → Unresolved external references → Requires a run-time loader
    - References can only be from the dynamically loaded module into the program into which it is loaded

Initialization and Registration

- Dynamically Loadable Module returns a vector that contains pointer to initialization method
    - Initialization method: Allocates memory, allocates I/O resources, initializes driver data structures, registers all of device instances it supports

Using a Dynamically Loaded Module

- Each service maintains open references to underlying devices
    - OS forwards call to appropriate device driver entry point when communication with device is needed
- <u>Federation Framework</u>: System maintains table of all registered device instances and the associated device driver entry points for each standard operation

Unloading

- OS calls shut-down method

    - Unregisters itself as device driver
    - Shuts down devices it had managed
    - Returns all allocated memory and I/O resources back to OS
- Memory is freed, module is safely unloaded afterwards

The Criticality of Stable Interfaces

- Set of entry points for any device driver must be well defined and work with all relevant interfaces
- Set of functions in main program that dynamically loaded modules are allowed to call must be well defined and stable

Hot-Pluggable Devices and Drivers

- Hot-Plug Manager
    - Subscribers to hot-plug events
    - When new device is inserted, walk the configuration space to identify device, find and load the appropriate driver
    - When device is removed, find associated driver and call its removal method

OSTEP Ch. 39 (Interlude: Files and Directories)

Persistent Storage Device: Stores information permanently

- Ex: Hard disk drives, solid-state storage devices

39.1 Files and Directories

File: Linear array of bytes that can be read or written

- Inode number: low-level name of a file
- Type: a *convention* that informally denotes format of data
- Name

Directory: Contains a list of pairs (user-readable names, low-level names)

- Has an Inode number
- Directory tree: Directory with another directory inside of it
- Root Directory: Topmost directory on hierarchy
    - Sub-Directories: Accessed using separators
        - Absolute Pathname: Full path of file

39.3 Creating Files

Open(): Creates a file, returns a file descriptor

- File Descriptor: Private per process integer used to access files
    - Used in reading or writing to the file
    - Capability: Opaque handle that gives you power to perform certain operations
    - 0: Standard input
    - 1: Standard output
    - 2: standard Error
    - First free file descriptor is 3

39.4 Reading and Writing Files

Strace(): Traces every system call made by a program while it runs, dumps the trace on the screen

Open File Table: System-wide table tracking which underlying file the descriptor refers to, the current offset, and the permissions of a file

39.5 Reading and Writing, but not Sequentially

Lseek(): Allows for opening of file at a file offset (rather than from the beginning)

- Simply changes the variable in OS memory that tracks the offset its next read or write will store (NO ACCESS TO DISK)

File Offset:

- Implicitly updated using read() and write()

- Explicitly updated using lseek()

Within the Open File Table:

- New entries have offsets initialized to zero upon open
- If a process opens the same file twice, each open creates a different entry in the table, updated independently of each other
- We can reposition the current offset before reading

39.6 Shared File Table Entries: fork() and dup()

Mapping of file descriptor to an entry in open file table is generally one-to-one

Cases where the open file table is *shared*

- Fork(): Parent and child processes use same inode, thus reference count increases
    - Reference Count: Shared files are only removed from the open file table when its reference count is 0
- Dup(): allows process to create a new file descriptor that refers to same underlying open file as an existing descriptor

39.7 Writing Immediately with fsync()

Writes to persistent storages often have time buffer until actual writing to memory

- In rare cases, data is lost

Fsync(): Forces all dirty data to disk, returns once all writes are complete

39.8 Renaming Files

Memory Mapping: Creates a correspondence between byte offsets in a file (backing file) and virtual addresses in the calling process (in-memory image)

Rename(): Atomic call that updates a file name (either to entire old or new name)

39.9 Getting Information About Files

Metadata: Information about a file

- File size, inode, ownership information, accessed or modified dates

39.10 Removing Files

File deletion is implemented using unlink()

39.11 Making Directories

- Directories cannot be directly written to

Initialization

- . is current directory
- .. is parent directory

39.12 Reading Directories

Implemented using opendir(), readdir(), and closedir()

39.13 Deleting Directories

- Directory deletion requires an empty directory

39.14 Hard Links

Creating a new entry in the file system requires link()

- Link() creates another name in directory you are creating link to, refers to the same inode number of the original file
  - File is not copied, but two readable names refer to the same file

Creating a file:

- Creates inode for the file (that stores all relevant information about it)
- Links a name to file, places it into a directory
- Unlink() is used due to reference count issues (only when the table entry is 0 will it be removed)

39.15 Symbolic Links/Soft Links

You cannot hard link to files in other disk partitions (because inode numbers are unique within a file system)

- Symbollic links are files
  - Golds the pathname of linked-to file as data of link file

    - Can cause dangling references

29.16 Permission Bits and Access Control Lists

Permission Bits

- Owner
- Group
- Other/Anyone

Superuser can access all files regardless of privileges

AFS uses access control list (ACL) to represent exactly who can access a given resource

29.17 Making and Mounting a File System

Mounting takes an existing directory as a mount point and pastes a new file system onto the directory tree at that point

Time of Check to Time of Use (TOCTTOU): Validity check variables can be modified in the time interval between the validity-check and its associated operation

- Possible Solutions:
    - Reducing number of services that need root privileges to run
    - Transactional File System

OSTEP Ch. 40 (File System Manipulation)

How to Implement a Simple File System

40.1 The Way to Think

- Data structures: Types of on-disk structures that are used by the file system to organize its data and metadata
- Access Methods: How does the file system map the calls made by a process onto its structures

    - Which structures are read/written during execution of a system call

40.2 Overall Organization

Division of disk into blocks

- Simple file systems have 1 block size
- Data Region: Region of disk used for user data
- Inode table: Array of on-disk inodes
    - Amount of inodes stored is maximum number of files allowed on the file system

Allocation Structures: Tracks whether inodes or data blocks are free or allocated

- Free list: Points to first free block
- Bitmap: Each bit is used to indicate whether the corresponding object/block is free or in-use (one for data, another for inodes)

Superblock: Contains information about particular file system, includes magic number to identify the file system type

- When file system is mounted, superblock is read first

40.3 File Organization: The Inode

Inode: Structure that holds metadata for a given file

- Metadata: Type, size, blocks, protection information, time

Retrieving an inode

- Disks are not byte addressable, but rather consist of large numbers of addressable sectors
- To fetch the block of inodes, the file system issues a read to a sector to fetch the desired inode block

References to where data blocks are

- Direct Pointers inside of the inode: Each pointer refers to one disk block belonging to the file

- Extents: Disk pointer plus a length (in blocks)
    - Troubles finding contiguous chunk of on-disk free space when allocating a file

Multi-Level Index

- Indirect Pointer: Pointer to a block that contains more pointers, each of which point to user data
- Double Indirect Pointer: Pointer refers to a block that contains pointers to indirect blocks, which contains pointers to data blocks
- Triple Indirect Pointer

Motivations

- Most files are small: Small number of direct pointers is justified

40.4 Directory Organization

Vsfs: Directories contain a list of pairs (entry name, inode number)

- For each file and directory in directory:
    - String (and its length) and number of data blocks of the directory

Linked List Approach:

- Inode includes single pointer that pointers to first block of the file
    - Sometimes include another pointer to point to end of file for larger file

FAT File System: Utilizes a file allocation table (an in-memory table of link information indexed by the address of the data block)

B-tree: Compactly represents which chunks of the disk are free

40.5 Free Space Management

Free Space Management: Tracking of which inodes and data blocks are free and which aren't

Pre-Allocation Policy: Heuristic for allocating space

- Linux looks for sequence of blocks that are free → Guarantee that portion of file is contiguous → Improving performance

40.6 Access Paths: Reading and Writing

Access Path

Reading a File from Disk

- File system traverse pathname and locate inode beginning in root directory
  - Amount of I/O generated by the open is proportional to the length of the pathname
- Reads do not access allocation structures (Only accessed when allocation is needed)

Writing a File to Disk

- May allocate a block (unless it is being overwritten)
- Generation of 5 I/Os
  - Read the data bitmap
  - Write the bitmap
  - Read the inode
  - Write the inode
  - Write the actual block itself
- Creating an inode → Create an inode and allocate space → High I/O

How to reduce File system I/O Costs?

40.7 Caching and Buffering

Fixed-Size Cache: Holds popular blocks

Static Partitioning: Divides resource into fixed proportions once

- Pros:
  - More predictable
  - Easier to Implement

- Cons:
    - Unused pages in file cache cannot be repurposed for some other use (and thus go to waste)

Dynamic Partitioning: Gives differing amounts of resource over time

- Unified Page Cache: Integration of virtual memory pages and file system pages
- Pros:
    - Better utilization (resource-hungry resources consume idle resources)
- Cons:
    - Harder implementation
    - Potentially worse performance

Write Buffering:

- Delays writes → batches updates into smaller set of I/Os
- Buffering number of writes into memory → System schedules subsequent I/Os → Increases performance
- Delays → Some writes avoided entirely
    - Durability/Performance Trade-Off

Direct I/O interface: Works around the cache

Raw disk Interface: Avoids file system altogether

File Types and Attributes

Ordinary Files

- Text files: Byte stream, broken into lines, rendered as characters
- Archive: Single file that contains many others, alternating sequence of headers and data blobs
- Load Module: Different sections represent different parts of a program

- MPEG stream: Sequence of audio, video, frames, containing compressed program information

Datatypes and associated applications

- To interpret the meaning of a file's bytestream:
  - Require users to specifically invoke the correct command to process the data
  - Consult a registry that associates a program with a file type
    - File name suffixes (.c, .png, etc)
    - Magic number at the start of a file
    - Extended Attributes

File Structures and Operations

- Some files can be processed sequentially

Other types of files

- Directories:
  - Represent name spaces (association of names with blobs of data
  - Namespaces with different files owned by numerous users imposes different sharing/privacy constraints on the access to each file
- Inter-Process Communications Ports (IPC)
  - Inter-process communications port: Channel in which data is passed
    - Ex: a pipe
- I/O Devices
  - Connects a computer to the outside world

File Attributes

- System Attributes
  - Standard Set:

        - Type, ownership, protection, when the file was created, last updated, last accessed, size
    - All files have these attributes
    - OS depends on these attributes
    - OS maintains these attributes
- Extended Attributes (Other information that is important to correctly process files)
    - Metadata: Not part of file's contents but descriptive information that is necessary to properly process the file's contents
        - If file is encrypted, check-summed, internationalized, etc
    - Approaches to supporting extended attributes:
        - Associate a limited number/size of name-value attributes with each file
        - Pair each file with one or more shadow files that contain additional resources and information

Diversity of Semantics

An Introduction to DOS FAT Volume and File Structure

BIOS: The BASIC I/O Subsystem

- BIOS ROM provided run-time support for a BASIC interpreter

DOS FAT:

- Heavily used around the world
- Provides reasonable performance with simple implementation
    - Successful example of linked list space allocation

Structural Overview

- Bootstrap: Code to be loaded into memory and executed when the computer is powered on
- Volume Descriptors: Information describing the size, type, and layout of the file system (and how to find other key metadata descriptors)
- File Descriptors: Information that describes a file and pointers to where actual data Is stored on disk
- Free Space Descriptors: Lists of blocks of (currently) unused space that can be allocated to files
- File Name Descriptors: Data structures that user-chosen names with each file
- DOS FAT divides volume into fixed sized blocks, grouped into block clusters
- File Allocation Table: Used as a free list and keeps track of which blocks have been allocated to which files
    - First file is root directory

Boot Block BIOS Parameter Block and FDISK Table

- Boot record:
    - Begins with branch → Volume description → Real bootstrap code → Optimal Disk Partitioning → Signature
- BIOS Parameter block: Contains brief summary of the device and file system
    - Device Geometry
        - Number of bytes per sector, number of sectors per track, number of tracks per cylinder, total number of sectors on volume
    - Describes way file system is laid out on the volume

        - Number of sectors per cluster, number of reserved sectors, number of Alternate File Allocation Tables, number of entries in root directory
- FDISK Table: Partition table at the end of the boot strap block which partitions disk into logical sub-disks
    - FDISK has four entries (each describing one disk partition)
        - Partition Type
        - ACTIVE indication
        - Disk address where partition starts and ends
        - Number of sectors contained within that partition

File Descriptors (Directories)

- DOS combines file description and file naming into single file descriptor
- DOS directory is file that contains fixed sized directory entries
    - Entries include:
        - 11 byte name
        - A byte of attribute bits for the file
        - Times and dates of creation, last modification, and date of last access
        - Pointer to first logical block of the file
        - Length in the file

Links and Free Space (File Allocation Table)

- DOS File Attribution Table: Contains one entry for each logical block in the volume
    - If a block is free, this is indicated by the FAT entry
    - If block is allocated, FAT entry returns block number of next logical block in file

- Space is allocated in multi-block clusters determined when file system is created
    - Allocating space in larger chunks improves I/O performance but increases internal fragmentation
    - Maximum number of clusters relies on width of FAT entries
- Next Block Pointers
    - A file’s directory entry contains a pointer to the first cluster of that file
        - FAT entry for that cluster tells the number of *next* cluster in file
    - Must sequentially figure out next block number, however FAT is so small that entire FAT can be stored in memory
- Free Space
    - -1 → End of File
    - 0 → Cluster is free
- Garbage collection
    - No freeing of blocks, simple change of first byte of file name
        - Greatly reduced amount of I/O from file deletion
        - FAT systems regularly ran out of space
    - When out of space, garbage collect
        - Find every valid entry, follow chain of next block pointers, if cluster not found, than it is free and is updated on the FAT
            - Allows for recovering of deleted files for quite a while

Descendants of the DOS File System

- Long File Names
    - 32 byte directory entries didn’t have enough room to hold longer names

        - Changing format of DOS directories could break thousands of applications
        - Solution: Place extended file names in additional directory entries
            - Each file described by old-format entry, but supplementary directory entries could provide extensions to the file name
            - Tagged with unusual set of attributes
- Alternate/back-up FATs
    - If primary FAT fails, use the back-up FAT
- ISO 9660 (CDs)
    - Write files contiguously
        - No need for next block pointers
    - Directory entries can be variable length, entries begin with length field
    - Name field in entries are variable length
    - Variable length extended attributes section after file name

Week 8

OSTEP Ch. 41 (Locality and the Fast File System)

Old Unix File System: Super Block, inode region contains inodes, most of disk taken up by data blocks

41.1 The Problem: Poor Performance

Old Unix: Performance started off bad and got worse over time

- File system treated disk like random access memory: Data is spread all over the place

  - File system became fragmented due to lack of management
  - Original block size was too small to be efficient

How to organize on-disk data to improve performance?

41.2 FFS: Disk Awareness is the Solution

Fast File System (FFS): File system designed to be "disk-aware" by keeping interface but changing the implementation

41.3 Organizing Structure: The Cylinder Group

FFS divides disk into cylinder groups

- Cylinders are sets of tracks on different surfaces of a hard drive that are the same distance from the center of the drive
  - Often represented as a block group
- FFS keeps copy of super block in each group
  - If one copy corrupts, file system is still mountable
- FFS keeps copy of inode bitmap, data bitmap in each group

41.4 Policies: How to Allocate Files and Directories

How does FFS decide to place files and associated metadata on disk?

- Keep related stuff together, keep unrelated stuff far apart

Relation is defined using heuristics

- Placement of directories
  - Select cylinder group with low number of allocated directors and high number of free inodes,
- Placement of files
  - Makes sure to allocate the data blocks of a file in the same group as its inode, thus preventing long seeks between inode and data
  - Places all files in same directory in the cylinder group of the directory they're in

41.5 Measuring File Locality

- Distance measured by how far up directory tree to find common ancestor
    - 25% of file accesses have distance of 2

41.6 The Large-File Exception

- Without a different rule, a large file would fill up the block group it is placed in
    - Prevents subsequent files from being placed within the block group, damaging locality
- FFS does nothing for large files
    - Spreads files across the groups
    - <u>Amortization</u>: Reducing an overhead by doing more work per overhead paid

41.7 A Few Other Things about FFS

- Internal fragmentation could result in half of the disk being wasted for a typical file system
    - Solution: Store things using <u>sub-blocks</u> and not blocks (512b little blocks that the file system could allocate to files)
    - Avoids inefficiency by issuing writes in 4KB chunks to file system
- Disk layout is optimized for performance
    - FFS skips over every other block (as it has enough time to request the next block before it passed. The disk head)
    - <u>Parameterization</u>: How many blocks FFS should skip in doing layout to avoid extra rotations
- Allowed long file names, symbolic links

<u>OSTEP Ch. 42 (Crash Consistency: FSCK and Journaling)</u>

Crash-Consistency Problem: How to update persistent data structures despite the presence of a power loss or system crash

- On-disk structure will be left in an inconsistent state

How to update the disk despite crashes?

- Approaches:
    - FSCK/File system checker
    - Journalling/Write-ahead logging

42.1 A Detailed Example

Crash Scenarios

- Single write succeeds
    - Just the data block (Db) is written to disk
        - Data on disk, no inode or block allocation. As if write never occurred
    - Just the updated inode is written to disk
        - Inode points to disk address where data block was about the be written, but data block was not written there, aka garbage
        - File-System inconsistency: Disagreements between bitmap and inode alludes to inconsistencies in the data structures of the file system
    - Just the updated bitmap is written to disk
        - The bitmap indicates that block is allocated, but no inode points to it
            - Inconsistent file system, space leak
- Two writes succeed
    - The inode and bitmap are written to disk, but not data
        - File system metadata is consistent, but block 5 has garbage

    - The inode and the data block are written, but not the bitmap
        - Inconsistency between inode and bitmap, inoode points to correct data on disk
    - The bitmap and data block are written but no the inode
        - Inconsistency between inode and data bitmap
        - Block was written and bitmap indicates its usage, no inode points to file

Crash-Consistency Problem/Consistent-Update Problem: Disk can only commit one write at a time, and crashes or power loss can occur between these updates

- Optimally, we would be able to move file system from one consistent state to another atomically

42.2 Solution #1: The File System Checker

"Let the inconsistencies happen and fix them later"

fsck checks:

- Superblock → Free blocks → Inode state corruption → Inode link counts → Duplicate Pointer Check → Bad block pointer check → Directory Integrity Checks

Issue: fsck is slow on large file systems

42.3 Solution #2: Journaling (or Write-Ahead Logging)

"When updating the disk, before overwriting the structures in place, first create a log describing what you are about to do"

Data Journaling

- Physical logging/Logical Logging: Placing exact physical contents of update in the journal
    - Log transaction tells about updates, including information about pending updates to the file system and transaction identifiers

- Journal Write: Write the transaction, including a transaction-begin block, all pending data and metadata updates, and a transaction end block to the log. Wait for writes to complete
- Checkpointing: Overwrite old structures in file system only when transactions are safely on disk
    - Write the pending metadata and data updates to their final locations in the file system
- Transactional Write/Journal Commit: Occurs atomically
    - 1. Write all blocks except TxE block to journal
    - 2. When writes complete, file system issues write of TxE. Journal is in safe state

Maintaining order between two disk writes

- Disable write buffering
- Issue explicit write barriers
    - Write Barrier: Guarantees all writes issued before the barrier will reach disk before any writes issued after the barrier

Recovery

- Crash occurs before transaction is written safely to log
    - Pending update is skipped
- Crash happens after the transaction has been committed to the log, but before the checkpoint is complete
    - Redo Logging: File system recovers update
        - Scan of log looking for transactions that have been committed to the disk, replay them

Batching Log Updates

- File systems do not commit each updates to disk one at a time

    - File systems buffer all updates into a global transaction
        - Avoids excessive write traffic to disk in many cases

Making the Log Infinite

- Issues with logs:
    - Larger logs → Longer recovery time
    - Full log → no transactions can be committed to disk
- Solution: Circular Log: Treat log as a circular data structure
    - Once a transaction has been checkpointed, file system should free the space it was occupying in the journal to allow log space to be reused
        - Log oldest and newest non-checkpointed transactions in journal superblock

Metadata Journaling

- Data Journaling: Journals all user data
- Ordered Journaling / Metadata Journaling: User data is not written to journal
    - Data block is written to the file system proper (avoiding extra write)
- Ordering of data write does matter for metadata-only journaling

Tricky Case: Block Reuse

- If block is reused in metadata-journalling, only inode is committed to journal but not newly-written data
    - Results in overwrite of user data if crash occurs
- Solution: Revoke Record: Never reuse blocks until delete of blocks is checkpointed

42.4 Solution #3: Other Approaches

Copy-on-write (COW):

- Never overwrites files or directories in place
- Places new updates to previously unused locations on disk

- After certain amount of updates, COW file flips root structure of file system to include pointers to newly updated structures

Backpointer-Based Consistency (BBC)

- No ordering is enforced between writes
- Additional back pointer is added to every block in system
  - Used in consistency checks, as you check forward pointer to next's previous pointer

Optimistic Crash Consistency: Issues as many writes to disk as possible by using a generalized form of the transaction checksum

OSTEP Ch. 44 (Flash-based SSDs)

Solid-State Storage: Contains no mechanical or moving parts like hard drives, built from transistors

- Retains information despite power loss

Flash (NAND-based flash)

- Writing to chunk requires erasure of bigger chunk
- Writing too often causes page to wear out

How to build a flash-based SSD?

44.1 Storing a Single Bit

Single-Level Cell (SLC) Flash: Single bit stored in transistor

Multi-level cell (MLC): Two Bits are encoded into different levels of charge

Triple-Level Cell (TLC): Three bits per cell

44.2 From Bits to Banks/Planes

Banks:

- Blocks (erase Blocks)
  - Pages

Planes: Consists of number of cells

44.3 Basic Flash Operations

- Read (a page)
  - Flash chip accesses any location uniformly quickly → Random access device
- Erase (a block)
  - Before writing to a page, the entire block must be erased/copied to another location
  - Sets state of page
- Program (a page)
  - Changes 1s and 0s to desired contents of a page to the flash
  - Sets state to be readable
- Erase + Program → Write → Wear-out

44.4 Flash Performance and Reliability

- Latency (least to most) → Read → Program → Erase
- Reliability
  - Mechanical disks → <u>Head crash</u> → drive head makes contact with recording surface
  - <u>Wear out</u>: Flash blocks slowly accrue extra charge with erasures and programs (making it more difficult to differentiate between a 0 and 1), eventually making block unusable
  - <u>Disturbance</u>: When accessing a particular page in a flash, some bits get flipped in neighboring pages (<u>read disturbs</u> or <u>program disturbs</u>)

44.5 From Raw Flash to Flash-Based SSDs

How to turn flash chips into typical storage device?

- SSD Contains:

    - Flash chips
    - Volatile memory: used in caching, buffering and mapping data
    - Control logic: Organizes device operations by converting reads and writes to internal flash operations
        - Flash Translation Layer (FTL): Takes reads and writes on logical blocks and turns them into low-level read, erase, and program commands on underlying physical blocks and physical pages
            - Can be performed in parallel, reduces write amplification
- Wear leveling: Spreading writes across blocks of cache evenly to prevent wear out
    - Pages are programmed in order

44.6 FTL Organization: A Bad Approach

Direct Mapped:

- Read to logical page maps to read of physical page
- Write to logical page → Erasure of entire block, program of old pages + new one
    - Performance issues on write: write amplification → terrible write performance

Issues with Direct Mapped

- Wear out from repeated overwriting of metadata/user file data
    - Too much control to user over wear out to client workload

46.7 A Log-Structured FTL

Log Structured:

- Write to Logical Block → Device appends write to next free spot in current block (logging)

    - Subsequent reads uses mapping table to store physical address of each logical block in the system
        - Translation of logical block address to physical page number required to read data
    - Pages written in order to prevent program disturbance
        - Improves wear leveling
    - Downsides:
        - Overwrites of logical blocks creates garbage
            - Periodic garbage collection
- If device loses power:
    - Out-of-band (OOB) area: Used to reconstruct mapping table in memory

44.8 Garbage Collection

Process

- Find a block containing one or more garbage pages
    - Store information about which logical blocks are stored within each page to determine live r dead page
- Read in live log
- Reclaim entire block for use in writing

Cache Flush: Ensures writes have actually been persisted

Trim: Takes address and informs device that blocks at address have been deleted

Overprovision: Adding extra flash capability to push cleaning to the background

44.9 Mapping Table Size

Log-Structuring / page-level FTL can create extremely large mapping tables

Block-Based Mapping: Keep a pointer per block of the device

- Performance issues with small writes → Increases write amplification as entire block needs to be rewritten

Hybrid Mapping: FTL keeps few blocks (log blocks mapped per-page) eased and directs all writes to them

- 2 Mapping Tables: FTL first consults log table, then data table
  - Log Table: Small set of per-page mappings
  - Data Table: Larger set of per-block mappings
- Switch Merge: Make log block into storage location, use deleted block as a log block
- Partial Merge: Reunites other pages of physical block (to be referred to with single block pointer)
  - Performs extra I/O and increases write amplification
- Full Merge: FTL pulls together pages from many other blocks and cleans
  - Harms performance

Page Mapping Plus Caching: Cache only the active parts of FTL in memory

- If working set of translations fits in memory
  - Excellent performance without high memory cost
- If working set of translations cannot fit in memory
  - Requires extra flash read to bring in missing mapping
    - Possible eviction of old mapping, possible write if map is dirty

44.10 Wear Leveling

Log structuring allows for decent distribution of writes

- Case: Block can be filled with long-lived data that is not overwritten → Garbage collection will never recollect the data → never receives fair share of read/write

    - Solution: FTL reads all live data out of blocks and rewrites it somewhere else

44.11 SSD Performance and Cost

Performance

- SSD performance peaks at a couple MB/s
- Less difference with hard disk drive in sequential performance

Cost

- SSDs are expensive ☹

OSTEP Ch. 45 (Data Integrity and Protection)

Data Integrity/Data Protection: Ensuring data stored in system is same when storage returns data to user

45.1 Disk Failure Modes

Fil-stop model: Entire disk works or entire disk fails

Single Block Failures:

- Latent Sector Errors (LSE): Occurs when disk sector has been damaged in some way
    - Head Crash: Disk head touches surface for some reason
    - Cosmic rays
        - Solution: Error Correcting Codes (ECC): Determines whether on-disk bits in block are good
    - Fun Facts:
        - Costly drives with multiple LSEs are more likely to develop additional errors
        - Significant amount of spatial and temporal locality
- Block Corruption: Disk cannot detect block is corrupt

    - Buggy disk firmware writes to block in wrong direction
    - Faulty bus
    - Silent Faults: Disk gives no indication of problem when returning data
    - Fail-Partial Disk Failure Model: Disks can still fail in entirety but seemingly work
    - Fun Facts:
        - Chance of corruption varies greatly across different drive models
        - Workload and disk size l=have little impact on corruption
        - Corruption is not independent within a disk or across disks in RAID
        - There exists spatial locality and some temporal locality

45.2 Handling Latent Sector Errors

Detecting LSE:

- When storage system tries to access block and disk returns error
    - Storage system uses redundancy mechanism to return correct data
        - Block reconstruction, disk reconstruction
            - If disk reconstruction cannot successfully complete, utilize two parity disks as backup to restore missing blocks

45.3 Detecting Corruption: The Checksum

Detecting Corruption:

- Checksum: Result of a function that takes chunk of data as input, computes function to produce small summary of data contents

Common Checksum Functions

- More protection you get, more costlier it is

- Minimize collisions (multiple mappings to same checksum)
- Exclusive Or (XOR)
    - Computed by XOR'ing each chunk of data being checksummed, producing single value for entire block
    - If two bits in a checksummed unit change are in the same position, corruption is not detected
- Addition
    - Perform 2s-complement addition over each chunk of data, ignoring overflow
    - Fast
    - Does not work if data is shifted
- Fletcher checksum
    - Compute two check bytes
    - Detects all single-bit, double-bit, and many burst errors
- Cyclic Redundancy Check (CRC):
    - Treat data bloc kas a binary number, divide it by a constant k, remainder of division is output

Checksum layout

- Pack checksums onto single sector, followed by n data blocks
    - Works on all disks
    - Less efficient if file system wants to overwrite blocks

45.4 Using Checksums

Stored Checksum: Checksum of read data block from disk

Computed Checksum: Client computed-checksum over data block

If stored and computed checksums are the same → File is same

If file is corrupted:

- If system has redundant copy, return redundant copy
- Otherwise, return an error

45.5 A New Problem: Misdirected Writes

Misdirected Write: Data is written to disk correctly but in wrong location

Handling misdirected writes

- Physical Identifier (physical ID): Consists of disk number and sector offset, added onto checksum
    - Adds redundancy to disk

45.6 One Last Problem: Lost Writes

Lost Write: Device informs upper layer that write is completed but it is not persisted

Handling lost writes

- Write Verify or read-after write: Immediately read by data after a write
    - Slow, doubles number of I/Os needed for a write
- Add checksum elsewhere is system to detect lost bytes
    - Zettabyte File System: Includes checksum in each file system inode and indirect block for every file block

45.7 Scrubbing

Disk Scrubbing: Periodic reading of every block of the system, checking whether checksums are valid

- Disk system can reduce chances that all copies of a certain data item become corrupted

45.8 Overheads of Checksumming

Space Overhead

- On disk itself: Each stored checksum takes up room on disk (that cannot be used for user data)

- Memory of the system: In data access, memory must be allocated for checksums alongside data itself
    - If system checks checksum and later discards, this is short lived
        - This is often not the case though as checksum are used for redundancy

Time Overhead

- CPU must compute checksum over each block upon storage and access
    - Combine data copying and checksumming into one streamlined activity
        - Copy is needed anyway
- Some checksum approaches (that store checksums distinctly from data) induce extra I/O
    - Design fixes checksum distinction from data
    - Tuning fixes background scrubbing extra I/O

Week 9

OSTEP Appendix A (A Dialogue on Virtual Machine Monitors)

Hypervisor: Virtual machine monitor

OSTEP Appendix B (Virtual Machine Monitors)

B.1 Introduction

Virtual Machine Monitor (VMM): Sits between one or more OS's and hardware gives illusion that OS runs the whole machine

How to Virtualize the Machine Underneath the OS

B.2 Motivation: Why VMMs?

- Administrator can consolidate multiple OSes into fewer hardware platforms

  - Lower costs and easier administration
- Improvement in functionality
- Testing and debugging

B.3 Virtualizing the CPU

Limited Direct Execution: CPU Virtualization technique

- Booting an OS means jumping to address of first instruction and letting OS run

Machine Switch: VMM saves entire machine state of one OS, restores the machine state of to-be-run OS, and jumps to PC of the to-be-run OS

- VM may be in the OS itself

Trapped Instructions on VMM:

- Use of trap table, processor switches to kernel mode
  - VMM knows where the trap handler is on the OS

System Calls on VMM:

- Execution of a trap instruction in kernel mode, OS handles privileged instruction and returns from trap, VMM performs return-from-trap and returns to user mode

OS runs in Supervisor mode: No privileged instructions, but greater access to memory

B.4 Virtualizing Memory

Machine Memory: VMM’s perspective of the physical memory of the system

- OS maps virtual-to-physical via per-process page tables
- VMM maps physical mappings to machine addresses via per-OS page tables

Address Translation: Process generates a virtual address, OS converts it to a physical address and fetches desired contents from memory

TLB Misses in VMM

- Process
    - Virtual memory misses in TLB
    - VMM TLB handles miss as true privileged owner
    - VMM TLB miss handler jumps into OS TLB miss handler
    - OS TLB miss handler runs page table lookup for Virtual Page Number
    - Calls trap into VMM to establish VPN-to-PFN mapping
    - System returns to user level, TLB hit is retried
- Notes:
    - More expensive
        - Software TLB: records every virtual-to-physical mapping

Para-Visualization: Running of a modified OS on a VMM

B.5 The Information Gap

Information Gap: VMM does not understand what the OS wants or is doing

- Examples
    - OS runs an idle loop while waiting for an interrupt
    - VMM zeroes a page twice (once by VMM, once by OS)
- Solutions
    - Inference: Utilization of implicit information

OSTEP Ch. 53 (Introduction to Operating System Security) PR :p

53.1 Introduction

Cracks in the OS:

- OSes are large and complex
- OS supports multiple processes simultaneously
- Different files and processes have different permissions
- All software relies on proper behavior of underlying hardware

53.2 What Are We Protecting?

Trusted Platform Module: Protects the boot process of the OS

Security Enclaves: General hardware elements which try to control what can be done on the machine

- Supports cloud computing

53.3 Security Goals and Policies

Goals

- Confidentiality: Information should be hidden from others as needed
  - Process memory space cannot be arbitrarily read by another process
- Integrity: If a piece of data is supposed to be in a state, other things should not be able to change it
  - If a user writes a record to a particular file, another user who should not be able to write that file can't change the record
- Availability: If information is supposed to be available for your own or others' use, an attacker should not be able to prevent its use
  - One process running on the system cannot hog the CPU and prevent other processes from getting their share of the CPU
- Controlled Sharing: Information is shown to some, but not others

Non-Repudiation: Accountability, if someone claims something, they cannot deny it later

Security Policies: Outlines specifics of how security goals are applied

OS has mechanisms necessary to implement a desired security policy with a high degree of assurance in its proper application

- Exceptions:
  - Maintaining a process' address space private unless specifically directed otherwise

53.4 Designing Secure Systems

- Economy of Mechanism: Keep system as small and simple as possible
- Fail-safe defaults: Default to security
- Complete mediation: Check if actions to be performed meets security policies
- Open design: Assume adversary knows every detail of your design
- Separation of privilege: Require separate parties or credentials to perform critical actions
- Least Privilege: Give a user or a process the minimum privileges required to perform the actions you wish to follow
- Least common mechanism: For different users or processes, use separate data structures or mechanisms to handle them
- Acceptability: People need to be able to use your OS

53.5 The Basics of OS Security

- Process Handling
    - Clean separation of processes that are only broken with OS help
        - Neither shared hardware nor OS services can divert from security goals
- System Calls
    - Processes access system services by making an explicit call
        - Access control mechanisms: determines if identified process is authorized to perform the requested action

OSTEP Ch. 54 (Authentication) PR :p

Principal: Security-meaningful entities that can request access to resources

Agent: Process or other active computing entity performing the request on behalf of a principal

Who is requesting an OS service?

- System calls made by particular processes
    - Trap from user code into OS
        - OS takes control and performs service in response to system call
            - OS data structure that describes the process
                - Determining identity: OS must make policy-based decision on whether to perform operation

Object: Particular resource whose access is being requested

- If OS determines that process has access:
    - OS remembers decision by using per-process data structure (like PCB)
        - Page tables
        - Credentials: Data created and manged by OS that keeps track of access decisions
- If OS has no credential, determine if request should be granted:
    - User identity: All processes run by a particular person will have same identity
        - Group of Users
        - Program that the process is running: Certain privileges are granted to particular programs

Once principal has been authenticated, systems will always rely on that authentication decision for at least lifetime of process

How to securely identify processes?

54.2 Attaching Identities to Processes

Processes are created by other processes

Approaches to Attaching identity

- Copy identity of process that created it
    - Primal process: Special system identity not assign to any human user assigned upon OS booting
    - Issue: No way to differentiate privileges between processes
- Assign a user ID for new processes
    - Issue: How does shell/window manager get your identity attached to itself?
        - When user interacts with system, OS sets new process's ownership to user
            - User identified by logging in

54.3 How to Authenticate Users?

Authenticating identity of human beings:

- Authentication based on what you know
- Authentication based on what you have
- Authentication based on what you are

54.4 Authentication By What You Know

Passwords: Secret known only to the party to be authenticated, used to prove identity

- Assumes other people do not know party's password
    - Vulnerabilities
        - User should not write the password on a piece of paper
        - System's copy of password may leak out
    - Encryption
        - Don't store things → Store in hash using cryptographic hash → Encrypt with strong cipher
            - Avoid storing passwords in:

        - Temporary editors/files
        - Distributed Executables
        - Heaps of executing programs
  - System does not need to know what the password is
    - Authenticate using the hash of the password
      - Hash functions cannot be reversed → Stored passwords cannot return a password
    - Cryptographic hashes: Make it infeasible to use the hash to figure out what the password is
      - Hard to design
- Assumes other people cannot guess the password
  - Length of password determines how resistant it is to guessing
  - Vulnerabilities:
    - Passwords are restricted to letters of alphabet
      - Dictionary Attack: Guessing based on lists of names and words
        - Solution: Limit how many wrong guesses can be made
    - Password file can be stolen
      - Solution: Concatenate a large random number (salt) to password before hashing
        - Hacker can no longer create one translation of passwords in the dictionary to their hashes but rather one per salt
  - Multi-factor Authentication / Two-Factor Authentication: Not secure for today's environments

    - Password Vault / Keychain: Encrypted file kept on computer that stores passwords
        - Encrypted with password of its own
- Party must remember their password

54.5 Authentication by What You Have

Linux Login Procedures

OS can determine whether user trying to log in has proper device or not

- Dongles: Security tokens designed to determine proper devices
- Frequent changes of whatever information the device passes into to OS
    - Otherwise, anyone who can learn the static information can pose as a user without the device
- What if you don’t have the authentication item?
    - You now don’t have the magic item needed to authenticate yourself to OS
    - Someone else has the authentication item and can pretend to be you

54.6 Authentication by What You Are

Computer programs measuring a human characteristic and determining if it belongs to a particular person

- Facial Recognition for phone authentication
    - Issues:
        - Camera assumer person phone holder is owner
        - Assumes owner’s face is visible
        - Assumes good lighting
        - Assumes owner faces the camera
    - Based on analysis across photos to factor in light

    - False Negative: Intolerance of all but the closest matches, sometimes the owner can't access their phone
    - False Positive: total tolerance of differences in measured versus stored data
- Many machines do not have the hardware more biometric data collection/techniques
- Physical gaps between where biometric quantity is measured and where it is checked

54.7 Authenticating Non-Humans

Approaches

- Passwords
    - Assign password to web server users, use as needed
    - Provide mechanism that allows a process to change its ownership
- Sometimes no authentication is required at all, as other users are given the right to assign new identities to processes they create
- Identifying groups of users who share common characteristics
    - Group users together. Check to see if non-human is member of group
        - Associate group membership with each process
        - Associate process's individual identity information as an index into a list of groups that people belong to

OSTEP Ch. 55 (Access Control) PR :p

55.1 Introduction

Turning information into software

- Access Control: Figuring out if the requests fits the security policy
- Authorization: If it does, perform the operation. If not, make sure it isn't done

55.2 Important Aspects of the Access Control Problem

- Subject: Entity that wants to perform the access
- Object: Thing that the subject wants to access (e.g. files, devices)
- Access: Particular mode of dealing with the object

Reference Monitor: Algorithm code that determines when access control decisions will be made

Low cost without compromising security

- Give subjects objects that belong only to them
    - Virtualization
    - Peripheral Devices

Approaches to determining if Subject can read and write object

- Access Control Lists: Access Control List determines if subject can right
- Capabilities: Subject determines if subject can write

55.3 Using ACLs For Access Control

- OS relies on ACLs for access control
    - Each file has its own access control list
    - ACL should be stored close to the read: inodes, data block, directory entry
- How big is the list?
    - Reserve a big per-file list that is sometimes full sometimes empty
        - Every time we want to check access, we must search
    - Variable-sized access control lists: Only allocates space required for each list
    - Read Write Execute permissions
        - Solved large storage and large searching issues
- Pros:

    - Easily discern who is allowed access to resource
    - Easily change set of subjects who can access an object
    - Easily access all of file's permission information from ACL
- Cons:
    - ACLs require storage of access control information near file, ACLs may require large amounts of searching
    - Determining entire set of resources a principal is permitted to access requires checking of all ACLs in system
    - Requires a common view of identity across all machines for ACLs to work in distributed systems
        - <u>Name Space</u>: If name chosen for new thing is already in use, don't allow it to be assigned
            - Ignore it
            - Require authority to approve name selection

55.4 Using Capabilities for Access Control

A running process has some set of capabilities that specify its access permissions

- Capabilities are bits specific to a resource
    - It is possible for anyone to create any capability they want
    - One someone has a working capability, they can copy it however many times they want to
        - Processes never gets its hands on any capability
        - OS manages capabilities using the PCB
        - Capabilities can be cryptographically protected
            - Cannot be guessed in a practical way
            - Works for a distributed system security
- Pros

    - Easy to determine which system resources a given principal can access
    - Revoking access merely requires removing the capability from the list
    - If stored in memory, capability list can be cheap to check
    - Easily creates processes with limited privileges
- Cons:
    - Determining the entire set of principals who can access a resource is expensive
        - Any principal can have a capability for the resource and all principal's capability lists must be checked
        - Systems must be able to create, store, and retrieve capabilities in a way that overcomes forgery
    - It is hard to create a process with subset of parent's privileges

55.5 Mandatory And Discretionary Access Control

Discretionary Access Control: Almost anyone or almost no one is given access to a resource is at the discretion of the owning user

Mandatory Access Control: Some elements of the access control decisions ins such systems are mandated by an authority, who can override the desires of the owner of the information

55.6 Practicalities of Access Control Mechanisms

Android Access Control Model

Role-Based Access Control (RBAC): "assigning permissions to users based on their role within an organization"

- Allows for a particular user to take on multiple disjoint roles

Type Enforcement: A particular role may have access to read or write to a file but not another thing outside of the role

- Security Context: Detailed access rules of particular objects

Privilege Escalation: Allows particular program to carefully extend privileges beyond those of the user who invokes them

OSTEP Ch. 56 (Protecting Information With Cryptography) PR :p

Cryptography: A Set of Techniques to convert data from one form to another in controlled ways with expected outcomes

How do we protect information outside of the OS's domain?

56.2 Cryptography

Cipher: Algorithm that uses a key to convert data into a different form

- Deterministic
- The cryptography's benefit relies entirely on the secrecy of the key

Cipher-text: Output of cryptographic algorithm

Symmetric Cryptography: The same key is used to encrypt and decrypt the data

56.3 Public Key Cryptography

Public Key Cryptography: One of the two keys is widely known to the entire public (public key). Only owner knows the private key

- Avoids securely distributing a secret key (as the owner keeps their key)
- Much more expensive than traditional cryptography

56.4 Cryptographic Hashes

Hash the data and encrypt the hash

- Chances to cause hash collisions

Cryptographic Hash:

- Properties
  - It is computationally infeasible to find two inputs that produce the same output

    - Any change tot an input will cause an unpredictable change to the resulting hash value
    - It is computationally infeasible to infer any properties of the input based only on the hash value
- Used to determine if a stored file has been altered

56.5 Cracking Cryptography

Mainly relies in software cracks

- Distribution of secret keys in software shared with people
- Incorrectly transmitting plaintext versions of keys across networks
- Choosing keys from a seriously reduced set of possible choices
- Heartbleed attack

Brute Force: Guess every key

Perfect Forward Secrecy: "gives assurances that session keys will not be compromised even if long-term secrets used in the session key exchange are compromised"

Gathering Entropy: Examining processes that are random in nature

56.6 Cryptography and Operating Systems

Two Approaches:

- Trust the OS (Thus, cryptography is not needed)
- Don't Trust the OS
    - Keep decryption at minimum
    - Dispose of plaintext immediately

At-Rest Encryption: Data is encrypted, key is not even on hardware

56.7 At Rest Encryption

- Store key as encrypted
    - Cannot be used in computations

    - Preserves a safe copy of data
- Full Disk Encryption: Entire contents of storage device are encrypted
    - Data remains encrypted as long as it is stored anywhere in the machine's memory (including shared buffers and user address spaces)
- Protection:
    - No extra against users trying to access data they should not be allowed to see
    - Does not protect against flaws in applications that divulges data
    - Does not protect against dishonest privileged users on the system (such as the system admin)
    - Does not protect against security flaws in the OS itself
- Pros:
    - Protections against physical theft
- Cases to protect selected data:
    - Archiving data that might need to be copies and must be preserved but not used
    - Storing sensitive data in a cloud computing facility
    - User-level encryption performed through an application
    - Password Vaults / Key Rings: User asks for a password when system is first accessed, but is not required to re-authenticate again until logging out
        - Permits an unattended terminal to be used by unauthorized parties to use someone else's access permissions

56.8 Cryptographic Capabilities

Symmetric Cryptography: Both the creator of the capability and the system checking it shares the same key

- Most feasible when both are on the same system

Public Key Cryptography: Creator and resource controller need not to be co-located and the trust relationships do not need to be strong

- Each hold one key

Week 10

Distributed Systems Goals and Challenges

OSTEP Ch. 48 Distributed Systems

Client/Server: Client interacts with large collection of machines

Commons Issues:

- Failure: A machine in system fails
- Performance: Reduction of work and efficiency of communication
- Security: Validation of identity, no third party monitoring

48.1 Communication Basics

Packet loss due to lack of buffering within a network switch, router or endpoint

- Router may drop one or more packets if cannot accommodate all in memory

48.2 Unreliable Communication Layers

Unreliable layers handle packet loss

UDP/IP: Form of end-to-end arguments

- Process accesses a socket's API to create a communication endpoint
- Processes on other machines send UDP datagrams (fixed sized messages)
- Use of checksums to detect corruption

48.3 Reliable Communication Layers

How to build a reliable layer that handles packet loss?

- Acknowledgement (ack): Sends sends a message to receiver, receiver sends short message back to *acknowledge* the receipt

    - Timeout: If no acknowledgement received within time, then it's lost
        - Sender simply retries send
        - Exponential Back-off: Double timeout if server is overloaded
- Aim for exactly one copy of message received by receiver
    - If duplicate received, we don't pass it onto next recipient
    - Sequence Counter: Localized counter which counts amount of times a packet passes by the receiver
    - TCP/IP

48.4 Communication Abstractions

Distributed Shared Memory (DSM): Enables processes on different machines to share large virtual space

- If virtual page exists on local machine, then fetch is quick
- If virtual page is on other machine, page fault handler sends message to other machine and page is installed
    - If machine fails, address space is gone
    - Page faults and fetches are expensive

48.5 Remote Procedure Call (RPC)

Remote Procedure Call: Makes process of executing code on remote machine simple through calling a function

- Client makes procedure call, results are later returned
    - Stub Generator/Protocol Compiler: Automizes packing of function arguments into messages (client stubs)
        - Avoids human error
        - Complex types → Handled by well-known types or annotation of data structure
        - Organization of concurrent servers → thread pools: Finite amount of worker threads handles RPC calls

            - Requires extra locks and synchronization primitives
- Runtime Library:
    - Locating a remote service:
        - Naming:
        - Client shows a hostname or IP address
        - Protocol suite handles packet routing to particular address
    - Building RPC on a reliable layer is inefficient
        - Current approach is unreliable where RPC handles reliability
- Other Issues
    - Long-Running Remote may appear as failure → Server sends message to client that it works properly
    - Sender-side fragmentation: Larger packets into smaller packets
    - Receiver-side Reassembly: Smaller Parts into one larger logical whole
    - Byte Ordering: Apply XDR as endianness message standard
    - Expose asynchronous nature of communication to clients
        - Client requests from RPC layer