Overview

Three Easy Pieces

- OS is a resource abstractor and resource allocator
  - resource abstractor
    defines set of logical resources that correspond to hardware resources and well defined operations on it
  - resource allocator
    decided who gets how much and when
- **Why do we want virtualization?**
  To make the computer easier to program and to provide each program the illusion that it is alone on the computer going through the Fetch Decode Execute cycle
- Concurrency
  juggling multiple things at once
- Persistence
  ability to store data that survives a program termination/ computer shutdown
  done by the file system

Kernel

- OS Kernel
  the core part of the software (OS is software) and is in charge of implementing resource abstraction and allocation
  A code and data that always resides in RAM
  - A challenging software development endeavors with multiple concerns
    not to use too much memory, speed, cannot use standard libraries
  - uses non portable compiler directives
  - Debugging is hard and a kernel bug in the file system makes generating a crash dump problematic

Process of Turning On a Computer

1. POST (Power-On Self-Tests) are performed by the BIOS in firmware/ROM (Read-Only Memory)
2. Booting: The BIOS runs a first program: the bootstrap program
3. The bootstrap program initializes the computer (register contents, device controller contents, ...)
4. Then it loads another program in RAM, the bootstrap loader, and runs it
5. The bootstrap loader loads (whole or part of) the kernel (i.e., some code and associated data) into RAM at a known/fixed address
6. At some point, a bootstrap loader creates and starts the first program (called init on Linux, launchd on OS X)
7. Once all this has been done... nothing happens until an event occurs

Booted OS

- kernel code and data reside in memory at a specified address, as loaded by the bootstrap program
- Kernel's footprint has to be small
- each program is loaded in RAM
- In RAM 2 kinds of code
  - User Code
    written by user and library developers
  - Kernel Code
    written by kernel devleopers
- process can run user code and by doing a system call it can run the kernel code

  - Memory Protection
    each process think its alone and processes never step on each other's toes in the RAM

Kernel: Event Handler

- event handler as in all entries into the kernel code occur as a result of an event
- kernel defines a handler for each event type
- 2 Kinds of OS Events
  Interrupts (Asynchronous): some device controller saying some hardware thing happened and generated in real time from the outside world
  Traps (Synchronous)
  caused by instruction executed by a running program and synchronous because generated as a part of fetch decode execute cycle from inside world
- System Call
  a trap by which user programs get the kernel to do some work on their behalf

Interfaces

- Batch: Historical Human Interface
- GUI
- CLI
  can start the system programs and user programs where the distinction between them is a matter of perspective
  system program is not in the OS but pre installed

System Call API

- "Lowest-level" interface to OS services and GUI, CLI are built on top of this API
- Every program uses it
- On Linux, the strace tool makes it possible to spy on how a program uses the System Call API
- On UNI-ish systems, the time tool makes it possible too time system calls
- each call is identified with a number stored internal table named the syscall table
- makes it possible to access virtualized hardware resource
- calls are expensive and should be done wisely

Main OS Services

- Process Management
  The OS is responsible for
  - Creating and deleting processes;
  - Suspending and resuming processes;
  - Providing mechanisms for process synchronization;
  - Providing mechanisms for process communication;
  - Providing mechanisms for deadlock handling.
- Memory Management
  determines what is in memory when the kernel is always in memory
  The OS is responsible for
  - Keeping track of which parts of memory are currently being used and by which process
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

  OS is not responsible for
  - for memory caching, cache coherency, etc. as they are managed by hardware

- Storage Management
  the OS provides uniform, logical view of information storage
  OS operates File System Management
  - Creating and deleting files and directories
  - Manipulating files and directories
  - Mapping files onto secondary storage
  - Backup files onto stable (non-volatile) storage media
  - Free-space management
  - Storage allocation
  - Disk scheduling
- I/O Management
  OS hides peculiarities of hardware devices from the user and the OS is responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), spooling (the overlapping of output of one job with input of other jobs), etc.
  - General device-driver interface
  - Drivers for specific hardware devices
- Protection and Security
  - Protection
    mechanisms for controlling access of processes to resources defined by the OS
  - Security
    defense of the system against internal and external attacks
    OS provides
    * Memory protection
    * Device protection
    * User IDs associated to processes and files
    * Group IDs for sets of users
    * Definition of privilege levels

Process

- OS abstraction to virtualize the CPU ; process is a program in execution
- Multiple process can be associated to the same program
- Multiple Process run on a system
  - OS / System Process
  - User Process
- the process is defined by
  - code (aka text)
  - program (static) data
  - program counter
  - content of all registers which include the Program Counter (PC)
  - heap
  - runtime stack
  - page table

Process Address Space

- defined by the code + data + heap + stack
- bounded by the OS configuration
- Heap
  - where the objects/structures/arrays are created
  - can be handled by memory manage but it is the OS that provides the dynamic memory allocation
- Runtime Stack
  - helps manage method/procedure/function calls and how to return from them
  - Items on the stack are pushed/popped in groups, or activation records, or frame
  - activation record contains all the bookkeeping necessary for placing and returning from function calls

Process Life Cycle

- a process can be in finite number of different states
- allowed transitions between some pairs of states
- transitions happen when some event occurs

Multi-Tasking (Multi Programming)

- multiple processes can co-exist in memory
- all processes have their own separate address space

CPU Virtualization

- context switching
  mechanism by which a running program is kicked off the CPU and another one which is done fast and really frequent

Process Control Block

- Process State
- Process ID (PID)
- User ID
- Saved Register Values
- CPU scheduling information
- Memory Management Information
- Accounting Information
- I/O Status Info

Process Table

- a list of PCB kept by the OS
- because the Kernel size is bounded so is the Process Table

Process API

Process Creation

- Process from a genealogy tree
- If process A creates process B implies that A is the parent of B and B is the child of A
- Process can have at most one parent
- Process can have many children
- Each Process has a PID $\in \mathbb{Z}$
  - picked by the OS and is increasing
  - PPID: PID of the parent of a process
- after creating the child the parent continues executing but at any point it can wait for the child's completion
- child can be
  - complete copy of the parent (have an exact copy of the address space)
  - be a new program

fork() System Call

- a system call that creates a new process
- child is almost exact copy of the parent except
  - PID
  - PPID
  - resource utilization
- after the call, the parent continues executing and the child begins executing
- fork() returns an integer value
  - fork() returns 0 to a child
  - fork() returns the child PID to the parent

Zombies

- a terminated child process remains as a zombie
- OS keeps Zombies for the following
  - Zombies do not use hardware resource
  - uses a slot in the Process Table
  - may fill up due to Zombies and cause fork() to fail
- A zombie lingers on until its parent has acknowledged its termination or parent dies
- Zombies is reaped by the OS
- frowned upon to leave zombies around unnecessarily

Process Termination

- process terminate itself with exit() which takes as argument an integer called the process exit/return/error/value/code
- process can terminate another process done using signals and kill() system call

Signals

- software interrupts
- OS defines a number of signals each with a name and a number
- signals happen for various reasons
  - invalid access to valid memory sends a SIGSEGV signal to the running process
  - SIGINT signal to the running program in the Shell
  - tying to access an invalid address sends a SIGBUS signal to the running process
  - can send SIGKILL to another process to kill it
- signals can be used for process synchronization

wait() and waitpid()

- wait()
  blocks until any child completes and returns the pid of completed child and child's exit code
- waitpid()
  blocks until a specific child completes; can be made nonblocking
- SIGCHILD
  when a child exits signal is sent to the parent
  - convenient way to avoid zombies
  - parent associates a handler to SIGCHILD
  - handler calls wait()
  - way all children terminations are acknowledged

Inter-Process Communication

Basics

- Independent Process
  each process runs code independently; both the parent and child are aware of each other but they do not interact
- Cooperate
  process needs to share information, speed up computation, and because it's convenient
- IPC
  means of communication between cooperating process
- All OSes provide system calls for Shared Memory
- All OSes provide system calls fro Message Passing

Remote Procedure Calls

- provides a procedure invocation abstraction across processes
- performed through a client stub with automatically generated code to
  - Marshal the param
  - Send the data over to the server
  - Wait for the server‘s answer
  - Unmarshal the returned values

Pipes

- powerful IPC abstraction provided by OSes
- an abstraction of an actual physical pipe with data that flows between two processes
- two ends: write and read end
- process can write a stream of bytes to the write end
- process can read from the read end
- process closes one of the ends the other will get some EOF notification

File Descriptors

- File Descriptor
  an integer associated with the open file
  file descriptor is the index in the array of process opened files
- Three Kind of File Descriptor
  - File descriptor 0: standard input (stdin)
  - File descriptor 1: standard output (stdout)
  - File descriptor 2: standard error (stderr)

OS Mechanisms

Limited Execution: Restricted Operations

- 2 kinds of instructions
  unprotected and protected

User Mode vs Kernel Mode

- User Mode
  protected instructions cannot be executed; where the user code is executed
- Kernel Mode
  where all instruction is executed and executes the kernel code
- Mode is indicated by status bit (mode bit) protected control register in the CPU
- Decode Stage
  If the instruction is protected and the mode bit is not set to "Kernel mode", abort and raise a trap (that the OS will answer by terminating the program saying something like "not allowed")
  else execute the instruction
- Protected Instructions
  - Updating the mode bit
  - halt the CPU
  - Update CPU control registers
  - Change the system clock
  - Read/Write I/O device control/status registers
  - general interact with hardware components

  all of these operations can happen in Kernel Mode and only the kernel code can use them

Trap Table

- stored in RAM
- CPU has a register that points to it
- For each event type that the CPU could receive, this table indicates the address in the kernel of the code that should be run to react to the event

Context Switching

- mechanism to kick a process off the CPU and give the CPU to another process
  - Save the context of the running process to the PCB in RAM
  - Make its state Ready
  - Restore from the PCB in RAM the context of a Ready process
  - Make its state Running
  - Restart its fetch-decode-execute cycle
- should be as fast as possible because it is pure overhead
- context switch is mechanism and deciding when to context switch is a policy which is called scheduling

Basics

Concurrent Computing

- several operations are performed during overlapping time periods
- a feature of a program that can do multiple things at the same time
- program is concurrent if it consists of units that can be executed independently
- **Purpose of Concurrency**
  - makes program run faster
    Running multiple activities at once can use the machine more effectively because there are multiple hardware components
  - make programs more responsive
    Structuring a program as concurrent activities can make it more responsive because while one activity blocks waiting for some event, another can do something

Concurrency with Process

- Processes run concurrently on the computer
- OS virtualizes memory processes don't share memory naturally
- This can make it difficult to program processes that have complicated cooperative behavior

Threads

- thread
  basic unit of CPU utilization within a process
- multi-threaded process
  concurrent execution of different parts of the same running program
- Thread comes with its own
  - Thread ID
  - Program Counter
  - Register Set
  - Stack
- shares with other threads
  - code/text section
  - data segment
  - list of open file descriptors
  - heap
  - signal behaviors
- Advantage of Threads vs. Process
  - Resource Sharing
    threads naturally share memory having concurrent activities in the same address
  - Economy
    Creating a thread is cheap
    Context-switching between threads is cheaper than between processes
    So if you can do with threads what you can do with processes, then you likely can do it a bit faster
- Drawback of Threads vs Process
  - one thread fails with an error/exception which is not managed
  - Threads may be more memory-constrained than processes
  - Threads do not benefit from memory protection

User Threads vs Kernel Threads

- User Threads
  - threads can be supported solely in User Space
  - main advantage: low overhead (e.g. no system calls)

  - Drawbacks
    If one thread blocks, all other threads block
    All threads run on the same core

- Kernel Threads
  - The kernel provides data structures and system calls to handle threads

- Thread Libraries
  provide users with ways to create threads in their own programs

Java Threads

- Thread Class
  Implementing a subclass that extends Thread

-

Java threads takeaway

- To launch or spawn a Thread/Runnable it is necessary to call the start() method (instead of run())

Basics

- Main Memory = Memory Unit
- array of byte/words contain its own address
- each incoming address is stored in the memory address register of the memory unit
- Called the "Main" memory by contrast with registers, caches, which are all managed 100% by the hardware
- Processes share the main memory, therefore the OS must manage the main memory
- CPU only works with registers but it can issue addresses that correspond to words of the main memory

Early System

- each process is allocated a contiguous zone of physical memory

Address Binding

def allocates a physical memory location to a logical pointer by associating a physical address to a logical address

Absolute Addressing

the binary executable contains physical addresses

Issue of Absolute Addressing

With absolute addressing a program must be loaded exactly at the same place into memory each time we run it

Therefore we may not be able to run a program because another program is running and encroaches on the address range!

Corollary

cannot run multiple instances of a single program

Relative Addressing

- Assume the address space starts at some BASE address, and computes all addresses as an offset from the BASE
- The code is now completely relocatable: Only the BASE needs to be determined before running it
- same program can be run anywhere in memory
- Multiple instances can run, each with a different BASE address

RAM Virtualization

- All addresses in the process address space are expressed as an offset relative to the base value

Memory Virtualization

- each program instance has the illusion that it's alone in RAM and that its address space starts at address 0
- Memory Protection
  - program doesn't need to know anything about other programs
  - good because writing code you don't know what other programs will be running anyway
- **Bottom Line** A program references a logical address space, which corresponds to a physical address space in the memory

Virtualizing The Process Address Space

- Some hardware component needs to translate virtual addresses into physical addresses
- Address translation happens very frequently, thus base address is accessed very frequently
- and offsets are added to the base address very frequently
- use a limit register that stores the largest possible logical address

Memory Management Unit

- specialized circuit between the CPU and the memory and integrated with the CPU nowdays

Segmentation

- having a single contiguous segment is wasteful
- Segmentation
  to avoid waste by breaking up the address space into pieces
- each piece has its own base/limit register
- logical address space is now a collection of segments
- compiler/language interpreter handles the segments and the logical addresses are built appropriately
- Segments used by C compiler
  text, data, heap, stacks, C library

Segment Table

- one entry per segment number is used to keep track of segments
- for each entry stores the base (starting address of the segment) and limit (length of segment)
- segment table is stored in memory
  - Segment Table Base Register
    points to the segment table address
  - segment table length register
    gives the length of the segment table; easy to detect an invalid segment offset
  - registers are saved/stored at each context switch

Swapping

def Moving processes back and forth between main memory and the disk
Concept
a process is swapped back in, it may be put into the same physical memory space or not
Swapping and DMA

- a process can be kicked out from RAM to disk by the OS at any time
- DMA controller may have no idea and happily continue to write data (into some other process' address space, which has replaced that of the one that was swapped out!)

Swapping Bad

- The disk is slow
- to cope with slow disks have been used
  - OS could swap in/out only processes with small address space
  - can dedicate disk/partition to swapping
- approach is to just not swap
  - Swapping should be an exceptional occurrence
  - Swapping is now often disabled

Memory Allocation

- Where should the processes be placed in memory?
  The kernel must keep a list of available memory regions or "holes"
  When a process arrives, before scheduling it, it is placed in a "I need memory" input queue
  Kernel must make the decision: pick a process from the input queue or pick a hole in which the process will be placed
- problem is known as the dynamic stroage allocation problem

Memory Allocation Strategies

1. Which process should be picked?
   - First Come First Serve
     easy, fast to compute, may delay small processes
   - allow smaller processes to jump ahead
     slower to compute favors small processes

- Limit the jumping ahead
- Look ahead

2. Which hole should be picked?
   - First Fit
     pick the first hole that is big enough
   - Best fit
     pick the smallest hold that is big enough
   - Worst fit
     pick the biggest hole
3. How should the picked process be placed in the picked hole?

Memory Allocation

- FCFS + First Fit + Top
- Jump Ahead + Worst Fit + Bottom
- above are heuristics that produce decent solutions
- same story to CPU scheduling

External Fragmentation

- Goal: hold as many processes as possible in memory
- External Fragmentation
  defined as number of holes
- For a given amount of available RAM, we're always happier with a single large hole than with several smaller holes
- processes terminate whenever they want to, we cannot avoid external fragmentation

Internal Fragmentation

- Do we want to keep track of tiny holes?
  no
- an OS would allocate slabs that are multiples of some "block size"
- Downside: a process may then not use the whole slab and some space is wasted

Smaller blocks
lower internal fragmentation, but more blocks to keep track of
Larger blocks
higher internal fragmentation, but fewer blocks to keep track of

Making Address Space Smaller

Smaller Address Space

- always a good idea as it is good for swapping
- don't swap as often (because if address spaces are small, then RAM looks bigger), not as slow to swap (because reading/writing a smaller address space from/to disk is faster)
- Three Common Place Techniques
    1. Dynamic Memory Allocation
       Ask programs to tell the OS exactly how much memory they need when they need it (malloc, new) so that we don't always allocate the maximum allowed RAM to each process
    2. Dynamic Loading
    3. Dynamic Linking

Dynamic Loading

- only load code/text when it's needed
- enacted by code written by the programmer for this purpose
- OS is not involved, although it provides tools to make dynamic loading possible

Dynamic/Static Linking

- Static Linking is the historical way of reusing code
- dynamic loading, but the OS loads the code automatically and different running programs can share the code
- code is shared in shared libraries:

Shared Libraries

- dynamic linking is enabled, the linker just puts a stub in the binary for each shared library routine reference
- a stub is a piece of that
    - checks whether the routine is loaded in memory