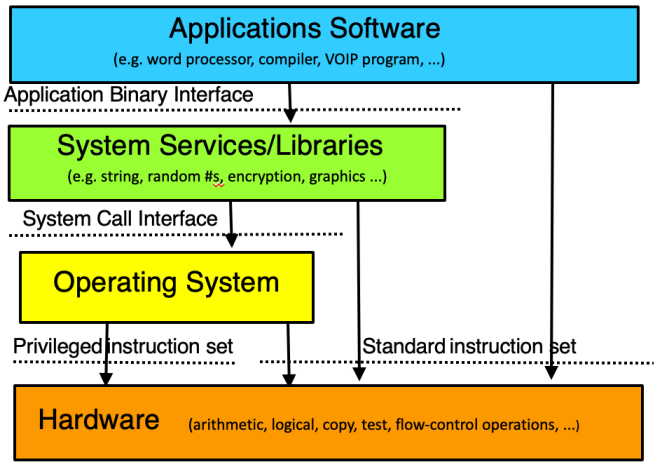
CS111 Introduction to Operating Systems

OS Introduction

- What is OS
  - Def: low level system software which provides better, more usable abstractions of the hardware below it, allowing safer, easier, fairer use and hsaring of resources
    - right on top of hardware to provide support to high level applications, hiding low level hardware details
    - System software intended to provide support for higher level applications
      - Primarily for user processes, some higher level system sw applications
    - Hides nasty details of hw, sw, common tasks
  - Virtualization: despite the fact that there's only one piece of hw, illusion that different applications can still use a piece of the hw and act as if it has a lot of hw
  - OS also crashes
    - Writing invalid programs
    - Security
- Why we have OS:
  - When we do NOT need OS:
    - System with one piece of hw, one piece of application that only do with this piece of hw.
    - --Anything beyond this need OS
  - MOTIVATION: Assembly programming is difficult; so when u write applications noneed to deal w/ low level details
    - Help perform complex operations while hiding complexity; make sure nothing is in the way of anything
      - Using various hardware and various bits of software
  - Foundation of all applications
    - Every smartphone, IOT, even more tiny devices, has an OS
    - Hardware: Memory chip, CPU, IO, monitors, hard-drive -> OS (bridge) -> Interface of word docx, video games, etc.
  - Everyone has an OS
    - All computing devices you will ever use has OS
      - Servers, laptops, desktop machines, tablets, smart phones, game consoles, set-up boxes
    - Tthings you don't think of as computer have CPU inside(usually w/ OS)
  - How to work with OS
    - Configure: setups like time zone, user name, etc
    - Use their features when writing programs
    - Rely on services they offer:
      - Memory mgmt.
      - persistent storage
      - scheduling and synchronization
      - interprocess communications
      - security
  - Many hard problems have been tackled in the context of OS
    - Coordinate separate computations, manage shared resources, virtualize hw & sw, organize communications, protect computing resources
    - OS solution often applicable to programs and systems you write
- What OS do for us
  - Manage hardware for programs
    - Allocate hw and manage its use
    - Enforce controlled sharing and privacy

        - Oversees execution and handles problems
    - Abstracts the hw
        - Makes it easier to use and improves sw portability
        - Optimize performance
- What does OS look like
    - A set of mgmt. & abstraction services
        - Invisible, happen behind the scenes
    - Applications see objects and their services
        - CPU support data-type and operations
            - Bytes, shorts, longs, floats, pointers
            - Add, subtract, copy, compare, write, signal
    - OS extends a computer
        - Create much richer virtual computing platform
            - Support richer objects, more powerful operations



- Distribution
    - Binary is the derivative of source
        - OS written in source
        - BUT: source MUST be compiled
        - A binary distribution is ready to run
    - OS usually distributed in binary, one binary distribution per ISA
        - Different ISA, different distribution format
    - Binary model for platform support:
        - Device drivers can be added after-market
            - Can be written and distributed by 3rd party
            - Same driver works with many versions of OS

- Interface stability
    - People want new releases of OS
        - New features, bug fixes, enhancement
    - People also fear new release of OS
        - OS changes can break old applications
        - Prevent:
            - Define well-specified Application Interfaces
                - Application Programming Interface (API)
                    - The interface between applications. Nothing to do with OS. Like services provided by function calls

        - Application Binary Interface (ABI)
            - Talk to OS. Compiled against ABI.
        - C program: OS independence program.
          Compiling C: depend on OS, because of different ABI.
    - Applications only use committed interfaces
    - OS vendors preserve upeards-compatibility
- OS and abstractions
    - Major function of OS: offer abstract versions of resources NOT actual physical!
    - OS implements the abstract resources using the physical resources
        - Example: processes (abstraction) are implemented using CPU and RAM (physical resources)
        - Example: files (abstraction) are implemented using disks (physical resources)
    - WHY abstractions
        - Simpler and better suited for programmers and users
            - Easier to use
                - Ex: no need to worry about keeping track of disk interrupts
            - Compartmentalize/encapsulate complexity
                - Ex: need not concern what other executing code is doing and how to stay out of their way
            - Eliminate behavior that is irrelevant to user
                - Ex: hide the slow erase cycle of flash memory
            - Create more convenient behavior
                - Ex: make it look like you have the network interface entirely for your own use
    - Common types of OS resources
        - Serially reusable resources
            - used by multiple clients, but only one at a time
            - Ex: printer, toilets, trash-can
            - require access control
            - require graceful transition
                - graceful transition: a switch that totally hides the fact that the resource used to belong to someone else.
                - Do not let second user to access the resource until the first user is finished
                    - No incomplete operations that finish after the transition
                - Ensure that each subsequent user finds the resource in. the “like-new” condition
                    - No trace of data or state left over from the first user
        - Partitionable resources
            - Divide into disjoint pieces for multiple clients
                - Spatial multiplexing
            - Example: memory (divided into pages), disk, hotel rooms
            - Need access control:
                - Containment: cannot access resources outside of your partition
                - Privacy: NOBODY else can access resources in your partition
            - Need graceful transition
                - When change “owner”; no need when it is “yours”
                - Most partitionable resources aren’t allocated permanently
                    - RAM
        - Sharable resources
            - usable by multiple concurrent clients

        - clients don't wait for access to resource
        - clients don't own a particular subset of the resource
    - May involve limitless resources
        - Example: CPU, air in classroom, copy of OS shared by processes
    - NO need for graceful period
        - Usually does not change state or isn't reused
        - Never have to clean up: it does not get dirty

- General OS Trend
    - C and UNIX system: 1960s
        - Unix never put into use in reality, but pioneer the foundations
        - Mac OS: Unix like system, instead of linux
    - Linux: follower of Unix
    - OS is changing:
        - Grow larger and more sophisticated
        - Role changed:
            - From shepherding the use of hw to:
                - Shielding applications from hw
                - Providing application computing platform
                - Become sophisticated "traffic cop"
        - STILL: Bridge between applications and hw
        - Understood thru services they provide:
            - Capabilities they add
            - Application they enable
            - Problems they eliminate
        - Reason:
            - User's need
            - OS must provide core services to applications
                - Applications have become more complex
                    - More complex internal behavior
                    - More complex interfaces
                    - More interactions with other software
                - OS needs to help with all that complexity
        - OS convergence:
            - 3 main OS: MacOS, windows, linux
            - Other special uses OS: realtime and embedded system OS
            - OS in the same family is used for different purposes
            - Most OS based on old models
            - Why converge:
                - Expensive to build and maintain OS
                    - Converge, more mature
                    - Hard to get-into and stay in such business
                - They only if succeed if users choose them over other OS options
                    - Should support all apps
                    - Require others parties to do a lot of work
                - Need to have clear advantage over alternatives

Instruction Set Architectures (ISAs)

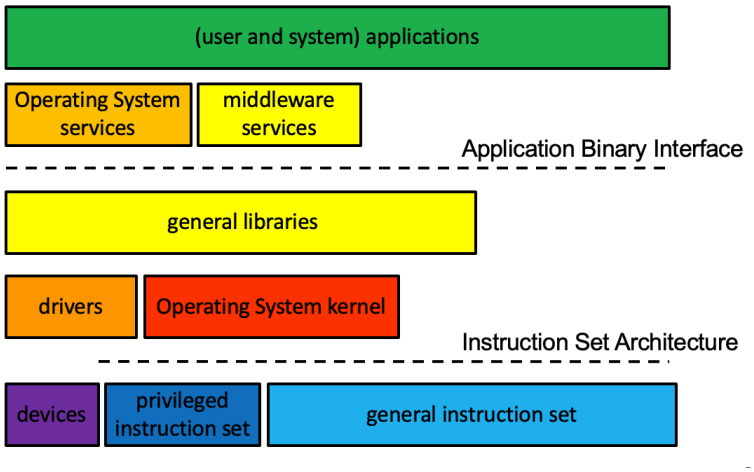
- Def: The set of instructions supported by a computer

    - Bit patterns correspond to what operations
- OS has nothing to do with ISA. ISA is OS independent.
- There are many different ISAs
    - Different word/bus widths (8, 16, 32, 64 bits)
    - Different features (low power, DSPs floating point)
    - Different design philosophies: RISC vs. CISC
    - Competitive reasons
- ISA usually come in families
    - Newer models add features
        - MUST remain upward-compatible with older models
            - a program written for an ISA will ONLY run on any complaint CPU
- Privileged vs. General instructions
    - Most modern ISAs divide the instruction set into privileged vs. general
        - General Mode:
            - Any code running on the machine can execute general instructions
            - can be executed by any program on the top
        - Privileged Mode:
            - Processor must be put into a special mode to execute privileged instructions: kernel mode
                - Usually only in that mode when OS is running
                - Privileged instructions do things that are “dangerous”
            - can only be used by OS. OS first enter kernel mode. NOTE user mode CANNOT access privileged. For safety (interruption) and security.
- Probability to Multiple ISAs
    - Successful OS run on MANY ISA => OS will abstract ISA!
        - Some consumers cannot choose their ISA
            - If you don’t support, can’t sell
    - Minimal assumptions about specific hardware
        - General frameworks are hardware independent
            - File system, protocols, process
        - Hardware assumptions isolated to specific modules
            - Context switch, I/O, mem mgmt.
        - Careful use of types
            - Word length, sign extension, byte order, alignment

OS Services

- OS Services
    - OS offers important services to other programs
        - Generally: as abstractions
        - Customer of OS service is applications
            - the motivation for OS
    - Basic categories:
        - CPU/Memory abstractions
            - Process, thread, virtual machines
            - Virtual address spaces, shared segments
        - Persistent storage abstractions
            - Files and file system
        - Other I/O abstractions
            - Virtual terminal sessions, windows
            - Sockets, pipe, VPNs, signals (as interrupts)

- Services: high level abstractions
  - Cooperating parallel processes
    - Locks, condition variables
    - Distributed transactions, leases
  - Security
    - User authentication
    - Secure sessions, encryption
  - User interface
    - GUI widgets, desktop and windows mgmt.
    - Multi-data
- Invisible services:
  - Error handling
  - Enclosure mgmt.
    - Hot-plug, power, fans
  - Sw updates and config registry
  - Dynamic resource allocation and scheduling
    - CPU, memory, bus, disk, network
  - Networks, protocols and domain services
    - USB, BlueTooth
    - TCP/IP, DHCP, LDAP, SNMP
    - iSCSI, CIFS, NFS



- How OS deliver services
  - Possible ways:
    - Applications call subroutines
    - Applications make system calls
      - Subroutine vs. system call
        - subroutine: like malloc; a long process that does a lot of things, may involve system call
        - system call: sbrk()
    - Application send message to software that perform the service
  - Each option works at a different layer of the stack of software
  - OS layering: NO privilege! JUST CODE!!
    - Modern OS offer services via layers of software and hardware
    - High level abstract services offered at high software layers
    - Lower level abstract services offered deeper in OS
    - Ultimately: everything mapped down to relatively simple hardware
  - Service Delivery via Subroutines

- push para, jump to subroutine, return values in registers on stack
- typically at high layers
- +:
    - Fast: no context switch (nano-seconds)
    - Run-time implementations binding possible
- -:
    - All services implemented in same address space
        - Address space: virtual memory
    - Limited ability to combine different language
    - Can't usually use privileged instructions (only OS can)

- Service delivery via library: JUST CODE! NO PRIVILEGE!
    - One way of service delivery via SUBROUTINE!!
    - Programmers need NOT write all code for programs
        - Std utility functions can be found in libraries
    - Libraries:
        - Def: a collection of object modules
            - Single file that contains many files (like zip/tar)
            - These modules can be used directly, W/O recompilation
            - Users can build their own lib: function often needed
        - +:
            - Reusable code
            - Single well written/maintained copy
            - Encapsulates complexity -> better building blocks
        - Compiling vs. linking vs. loading vs. execution
            - Compile: source code into binary format
            - Link: solve the reference for names not included in current source file
            - Load:
            - Execute: OS create a process
        - Multiple bind time options (how lib are linked)
            - Static: include in load module in link time
                - Added to a program's load module
                    - Each load module has its own copy of each library
                        - Size of each process ++
                    - Program MUST be relinked to incorporate new lib
                        - Existing doesn't benefit from bug-fixed
            - Shared: map into address space at exe time
                - NEED to know where the lib is statically; take pointer instead of copy to the program
                - One in mem; Shared by all process
                - Keep the lib separate from load modules
                - OS loads lib along w/ the program
                - +:
                    - Reduced mem consumption
                    - Faster program start-ups
                        - If already in Mem, no need to load again
                    - Simplified updates:
                        - Lib modules are not included in program load module
                        - Lib can be updated easily

        - Programs auto get the newest version when they restarted
    - Limitations:
        - Not all modules will work in shared lib
            - Cannot define/include global data storage (globally global issue)
        - Added into program Mem
            - Whether actually needed/not (if-else)
        - Called routines MUST be known at compile-time
            - fetch code delayed til runtime
            - Symbols known at compile time, bound at link time
        - Dynamically loadable libraries are most general
- Dynamic: choose and load at rum-time
    - At compile time, no idea where the lib is; or even which lib to call
    - The decision of where and how to load the lib happens at runtime (dlopen())

- Service Delivery via System Calls
    - Force an entry into the OS
        - Parameter/returns similar to subroutine
        - Implementation is in shared/trusted KERNEL!
    - Privilege, speed (get thru trap functions, thru HW)
    - +:
        - Able to allocate/use new/privileged resources
        - Able to share/communicate w/ other processes
    - -: avoid whenever u can
        - All implemented on the local node
        - 100x-1000x slower than subroutine calls
    - Providing services via Kernel
        - Primarily functions that require privilege
            - Primarily instructions (I/O, interrupts)
            - Alloc of physical resource (Mem)
            - Ensuring process privacy and containment
            - Ensuring integrity of critical resources
        - Some operations may be out-sourced
            - Sys daemons, server process
        - Some plug-ins may be less trusted
            - Device drivers, file sys, network protocols

    - System Services outside the kernel
        - NOT all trusted code must be in the kernel
            - May not need to access kernel data structures
            - May not need to execute privileged instructions
        - SOME: somewhat privileged processes
            - Login can create/set user credentials
            - Some can directly execute I/O operations
        - SOME: merely trusted
            - Sendmail is trusted to properly label massages

        - NFS server is trusted to honor access control data
    - Service Delivery via Messages
        - Exchange messages w/ server (via system calls)
            - Parameters in request, returns in response
        - +:
            - Server can be anywhere on earth, or local
            - Service can be highly scalable and available
            - Service can be implemented in user-mode code
        - -:
            - 1000x – 100 000x slower than subroutines
            - Limited ability to operate on process resources
    - Service delivery via Middleware
        - Middleware: Software that is a key part of the application service but NOT part of the OS
            - Database, pub/sub messaging system
            - Apache, Nginx
            - Hadoop, Zookeeper, Beowulf, OpenStack
            - Cassandra, RAMCloud, Ceph, Gluster
            - Machine learning, big data
        - Kernel code is very expensive and dangerous
            - User-mode code is easier to build, test and debug
            - User-mode code is much more portable
            - User-mode code can crash and be restarted
- Service interfaces and standards
    - OS interfaces
        - No one buys a computer to run the OS
        - OS is meant to support other programs via abstract services
            - Usually general: support different programs
        - Interfaces are required between the OS and other programs to offer general services
    - Application Program Interface (API)
        - Source level interface, specifying:
            - Include files, data types, constants
            - Macros, routine and their parameters
        - Basis for software portability
            - Recompile program for the desired architecture
            - Linkage edit with OS specific libraries
            - Resulting binary runs on THAT architecture and OS
        - An API compliant program will compile and run on ANY compliant system
        - API are for PROGRAMMERS!
    - Application Binary Interface (ABI)
        - A binary interface, specifying:
            - Dynamically loadable libraries
            - Data formats, calling sequences, linkage conventions
        - A basis for binary compatibility
            - One binary serves ALL customers for that hardware
        - An ABI compliant program will run (unmodified) on ANY complaint system
        - AbIs are for USERS
    - Libraries and Interfaces
        - Normal libraries (shared and otherwise): accessed thru API
            - Source level definitions of HOW to access the lib

- Readily portable between different machines
- Dynamically loadable libraries ALSO called thru API:
    - BUT: loading mechanism is ABI specific!
    - Issues of work length, stack format, linkages, etc.

- Interfaces and Interoperability
    - Strong, stable interfaces are KEY to:
        - allowing programs operate together
        - OS evolution
            - Don't want OS upgrade to break your existing programs
            - => interface between OS and programs better NOT change
    - Interoperability requires stability
        - NO program is an island
            - Program use system calls
            - Program call library routines
            - Programs operate on external files
            - Programs exchange messages with other software
            - If interface change, programs fail
        - API requirements are frozen at compile time
            - Execution platform MUST support those interfaces
            - All partners/services MUST support those protocols
            - All future upgrades MUST support older interfaces
    - Interoperability requires compliance
        - Complete interoperability testing is impossible
            - Cannot test all applications on all platforms
            - Cannot test interoperability of all implementations
            - New apps and platforms are added continuously
        - Instead, focus on interfaces:
            - Interface are completely and rigorously specified
            - Standards bodies manage the interface definitions
            - Compliance suites validate the implementations
        - And, hope that sampled testing will suffice
    - Side effects
        - All side effects occur when an action on one object has non-obvious consequences
            - Effects not specified by interfaces
            - Perhaps even to other objects
        - Often due to shared state between seemingly independent modules and functions
        - Lead to unexpected behaviors
            - Resulting bugs can be hard to find
    - Standards
        - Different than interfaces
            - Interfaces can differ from OS to OS, but standards are global
        - If you don't follow standards, others cannot work with you
        - Roles of standards:
            - There are many software standards
                - Subroutines, protocols, data formats
                - Portability & interoperability
                - Some are general, some domain specific
            - Key standards are widely required
                - Non-compliance reduces application capture

        - Non-compliance raises price to customers
    - Bottom line: if you don't meet the standard, your system won't be used
- Service and interface abstractions
    - Interface is a definition of contract, about specifications. Interface is implementation neutral.
    - Abstractions
        - Ex: address
        - Motivation: many things an OS handles are complex (varieties of hardware, software, configurations), life is easier for application programmers and users if they work w/ a simple abstraction
        - OS creates, manages ,and exports such abstractions
        - Simplifying abstraction
            - Ex: the number on watch is ~ of time
            - Hardware is fast, BUT complex and limited
                - Using it correctly is complex
                - May not support the desired functionality
                - NOT a solution, just a building block
            - Why abstraction:
                - Encapsulate implementation details
                    - Error handling, performance optimization
                    - Eliminate behavior that is irrelevant to the user
                - Provide more convenient or powerful behavior
                    - Operations better suited to user needs
    - Critical OS abstractions
        - OS provides some core abstractions that outr computational model relies on, and build others on top of that
        - Memory abstraction
            - Example:
                - Stack variables/heap object is the abs of memory
                - OS perspective: sequence bytes is abs of memory
            - Many resources used by programs and people relate to data storage
                - variables, chunks of allocated memory, files, database records, messages to be sent and received
            - These have similar properties:
                - YOU read and write them
                - There are complications
            - Complicating factors:
                - Persistent vs. transient memory
                - Size of memory operations
                    - Size user/application wants to work with
                    - Size the physical device actually works with
                - Coherence and atomicity
                    - single operation, no interfere
                - Latency
                - Same abstraction might be implemented with many different physical devices
            - Source of complications:
                - OS does NOT have abstract devices with arbitrary properties; OS has particular physical devices
                - Core OS abstraction problem:

        - Creating the abstract device with the desirable properties from the physical device that lacks them
    - Example: a typical file
        - We can real/write the file (arbitrary amt of data)
        - When writing: expect our next read to reflect the results of write => coherence
        - Expect the entire read/write to occur => atomicity
        - If there are several rea/write to the file, we expect them to occur in SOME order
    - Implementing a file
        - Often on a hard disk device
        - Disk drives have peculiar characteristics => OS need to smooth out
            - Long, worse variable access latencies
            - Accesses performed in chunks of fixed size
                - Atomicity ONLY for accesses of THAT size
            - Highly variable performance depending on exactly what gets put where
            - Unpleasant failure modes
        - Consequence:
            - Great effort by file system component of OS to put things in the right place on disk
            - Recording of disk operations to improve performance => complicates providing atomicity
            - Optimizations based on caching and read-ahead => complicates maintaining consistency
            - Sophisticated organizations to handle failures
- Processor abstraction
    - Abstraction of interpreters
        - Interpreter: something that performs commands; executer of program; one way of executing program
        - Motivation: The physical level of the processor is NOT easy to use, so the OS provides us with higher level interpreter abstractions
        - Basic interpreter components
            - Instruction reference
                - Tells interpreter which instruction to do next
            - Repertoire
                - Set of things the interpreter CAN do
            - Environment reference
                - Describes the current state on which the next instruction should be performed
            - Interrupts
                - Situations in which the instruction reference pointer is overridden
        - Example: a process
            - OS maintains a program counter for the process => instruction reference
            - Its source code specifies its repertoire
            - Its tack, heap, register contents are its environment
                - OS maintains pointers to ALL of them

            - No other interpreters should be able to mess up the process's resources
    - Implementing process abstractions in OS
        - Difficulties:
            - Easy if there's only 1 process, BUT always multiple
            - OS has limited physical memory to hold environment info
            - Usually ONLY 1 set of registers, OR one per core
            - Processes share CPU/core
        - Solving:
            - Schedulers share the CPU among various processes
            - Memory management hardware and software
                - To multiplex memory use among the processes
                - Giving each the illusion of full exclusive use of memory
            - Access control mechanisms for other memory abstractions
                - So other processes cant't fiddle with my file
- Communication abstraction
    - Communication link allows one interpreter to talk to another
        - On same or different machines
    - Ex:
        - At physical level, memory and cables
        - At abstract level, network and interprocess communication mechanisms
    - Communication links distinct from memory
        - Highly variable performance
        - Often asynchronous
            - Usually issues with synchronizing the parties
        - Receiver may only perform the operation because the send occurred: unlike the typical
        - Additional complications when working with a remote machine
    - Implementing the communications link in OS
        - Easy if both ends are on the same machine, complicated when remote
        - On same machine, use memory for transfer:
            - Copy message from sender's memory to receiver's
            - OR: transfer control of memory containing the message from sender to receiver
                - Shared memory: fast access, less reliable
        - Consequence:
            - Need to optimize cost of copying
            - Tricky memory mgmt.
            - Inclusion of complex network protocols in the OS itself
            - Worries about message loss, retransmission, etc.
            - New security concerns that OS might need to address
- Generalizing abstractions
    - Make many different things appear the same
        - Applications can all deal with a single class
        - Often lowest common denominator + sub-class
    - Requires a common/unifying model
        - Portable document format (PDF) for printed output
        - SCSI/SATA/SAS standard for disks, CDs, SSDs
    - Usually involved a federation framework

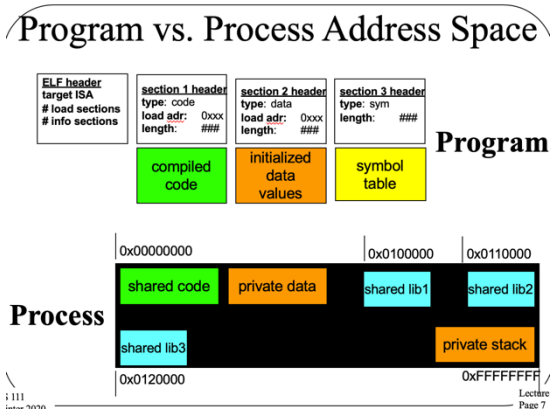
        - Federation frameworks: A structure that allows but somewhat different things to be treated uniformly
            - Method: create one interface that all must meet, then plug in implementations for the particular things you have
            - Ex: make all hard disk drives accept the same commands, even have 1000 models installed
            - Common models does NOT have to be "lowest common denominator"
                - The model can include optional features
                    - Which, if present, implemented in a standard way
                    - BUT may not always be present and can be tested for
                - Many device have features that cannot be exploited thru common model
    - Abstraction and layering
        - It is common to create increasingly complex services by layering abstractions
            - Ex: a generic file system layers on a particular file system, which layers on abstract disk, which layers on real disk
            - +: Layering allows good modularity
                - Easy to build multiple services on lower layer
                    - Ex: multiple file system on one disk
                - Easy to multiple underlying services support a higher layer
                    - Ex: file system can have either a single disk or RAID below it
            - -:
                - Typically add performance penalties
                - Often expensive to go from one layer to the next
                    - Frequently requires changing data
                    - Involved extra instructions
                - Lower layer may limit what the upper layer can do
                    - Ex: abstract disk prevents operation reorderings to maximize performance
    - Other OS abstractions:
        - Often provide different ways of achieving similar goal
        - OS MUST do work to provide each abstraction
            - Higher the level, the more work
        - Programmers and users have to choose the right abstraction to work with

Process

- What is process
    - What is a PROCESS:
        - A type of interpreter
        - An executing instance of a program
        - A virtual private computer
        - A process is an object which is:
            - Characterized by its properties
            - Characterized by its operations
            - NOT all OS objects are processes
                - BUT process us central and vital OS object
        - A virtual private computer

        - Each process is a separate “computer”, in some sense;
        - Own private share of resource (CPU, memory, disk)
    - What is STATE:
        - Dictionary definition: a mode or condition of being
            - An object MAY have a wide range of possible states
        - All persistent objects have state
            - Distinguishing them from other objects
            - Characterizing objects’ current condition
        - Condition of state depends on object
            - Complex operations often means complex state
            - Can serve/store the aggregate/total state
            - Can talk of a subset (Ex: scheduling state)
    - OS object State
        - Ex:
            - Scheduling priority of a process
            - Current pointer into a file
            - Completion condition of an I/O operation
            - List of memory pages allocated to a process
            - OS objects’ state is mostly managed by the OS itself
                - NOT by user mode
                - MUST ask OS to access/alter state of OS objects

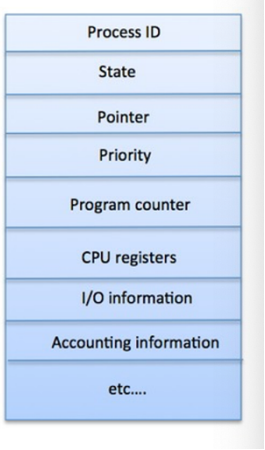
- OS handling process
    - Process Address Space (virtual memory)
        - Def: Each process has some memory addresses reserved for its use
        - A process’s address space is made all memory locations that the CAN address, otherwise CANNOT access
        - Modern OS pretend that EVERY process’s address space can include memory => which is NOT true



    - Process address space layout
        - ALL required memory elements for a process MUST be put somewhere in its address space
        - Different types of memory elements have different requirements
            - Ex: code is not writable but executable
            - Ex: stack is readable & writable but not executable
        - Each OS has some strategy for where to put these process memory segments
        - Unix Process in Memory
            - Code segments: statically sized
            - Data segments: grows up
            - Stack segments: grows down
            - NOT ALLOWED TO MEET!!

- Code Segments in address space
  - Start with load module
    - The output of a linkage editor
    - All external references have been resolved
    - All modules combined into a few segments
    - Includes multiple segments (code, data, etc)
  - Code MUST be loaded into memory
    - Instructions can only be run from RAM
    - A code segment must be created
    - Code must be read in from the load module
    - Map segment into process's address space
  - Code segments are read/execute ONLY and SHARABLE
    - Process CAN use the same code segments
- Data Segments in address space
  - Data MUST be initialized in address space
    - Process data segments must be created and mapped into the process's address space
    - Initial contents must be copied from the load module
  - Properties:
    - Read/write only; Process PRIVATE
    - Program can grow/shrink it (system call: sbrk)
- Stack Segment in address space
  - Size of stack depends on program activities
    - Ex: amt of local storage used by each routine
    - Grows larger as calls nest more DEEPLY
    - After call returns, stack frames can re recycled
  - OS manages the process's stack segment
    - Stack segment created at the same time as data segment
    - Some OS allocate fixed sized stack to program load time
    - Some dynamically extend stack as program needs it
  - Read/write only; process PRIVATE
  - Process and Stack Frame
    - Modern programming languages are stack based
    - Each procedure call allocates a new stack frame
      - Storage for procedure local (vs. global) variables
      - Storage for invocation parameters
      - Save restore registers
        - Popped off stack when call returns
    - Modern CPU also has stack support
      - Stack MUST be preserved as part of process state
    - Stack overflow: never-terminating recursion, touch the data segment, and crash the program
- Libraries in address space
  - Static libraries are added to load module
    - EACH load module has its own copy of EACH library
    - Program MUST be relinked to get new version

        - Shared libraries use less space
            - One in-memory copy, shared by all processes
            - Keep the library separate from the load module
            - Operating system loads library along with the program
            - +: reduced memory use, faster loads, easier lib upgrade
    - Other process states:
        - Registers
            - General registers
            - Program counter, processor status, stack pointer, frame ptr
        - Process's own OS resources
            - Open files, current working directory, locks
        - OS needs some data structure to keep track of ALL these info
            - Solution: Process Descriptors
    - Process Descriptors



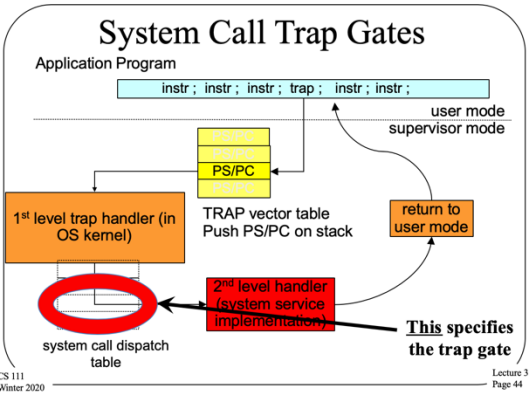
        - Def: Basic OS structure for dealing with processes
        - Function: store all into relevant to the process
            - State to restore when process is dispatched
            - References to allocated resources
            - Info to support process operations
        - Managed by: OS
        - Uses: scheduling, security decision, allocation issues
        - Ex: keep track of:
            - Unique Process ID(pid)
            - State of the process (e.g. running)
            - Address space info
            - Various other things
        - NOT all process state is stored directly in the process descriptor
            - Ex: application execution state: on stack, in registers
            - Ex: Linux process have supervisor-mode stack
                - To retain the state of in-progress system calls
                - To save the state of an interrupt preempted process

- Handling Process
    - Creating process
        - Created by OS
            - Using some method to initialize their state
            - In particular, set up a particular program to run
        - At request of other processes
            - Which specify the program to run
            - Other aspects of their initial state
        - Parent process
            - The process that created your process
        - Child process
            - The processes your process created
        - Creating a Process Descriptor
            - A new process NEEDs a PD
                - OS put PD into a process table
                    - Process table: The data structure the OS uses to organize all currently active process

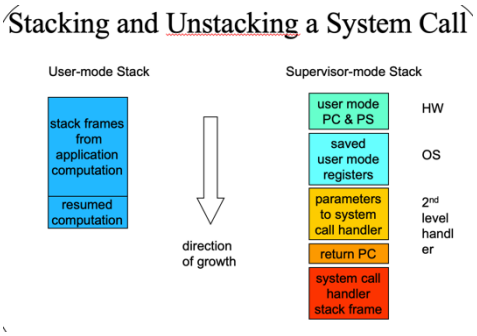
        - Contain one entry(a PCB)for each process in system
    - A new process NEEDs an address space t hold all the segments it will need, and allocate memory for code, data, stack
        - OS then loads program code and data into new segments
        - Initialize a stack segments
        - Set up initial registers (PC, PS, SP)
- Choices for process creation
    - Start with blank process
        - No initial state or resources
        - Have some way of filling in the vital stuff: code, PC, ect.
        - Basic Windows approach
            - CreateProcess() system call
            - Flexible (many parameters w/ many possible value)
            - Generally the system cak includes the name of the program to run
            - Different parameters fill out other critical information for the new process: environment info, prioriteis
        - The system call that creates it need to provide:
            - Everything needed to set up the process properly
            - At minimum, what code is to be run, etc.
    - Use the calling process as template
        - Give new process the same stuff as the old one: code, PC, etc.
        - Basic Unix/Linux approach
            - fork()
            - clones the existing parent process
            - on assumption that new child process is a lot like the old one: parallel programming
                - not true for typical user computing
            - after fork: 2 process w/ differ pid, same code & PC
    - Forking and data segments:
        - Forked child share the parent code
        - Forked child has PRIVATE stack
            - Initialized to match the parent's, like second process has run to the same point
        - Child should have its OWN data segment
    - Copy-on-Write
        - If the parent had a big data area, setting up a separate copy for the child is expensive; fork is supposed to be cheap
        - If neither parent nor child write the parent's data area, no copy needed => solution: copy-on-write
        - Copy-on-write: if one of them writes it, then make a copy and let the process write the copy, other process keep ori
    - Remake a process (Unix): exec() (call after fork())
        - Change the code associated with a process, reset much of the rest of its state (e.g., open files)
        - Different stack AND PC
        - Different set of resources
        - OS handling exec():
            - Get rid of old code, stack, data (☺ if COW)
            - Load a brand new set of code for that process

        - Initialize child's stack, PC, and other relevant control structure => start a fresh program
- Destroying process
    - Terminating:
        - All process terminate when machine down; but most exit before
        - Some process killed by OS or other processes
        - When process terminates: OS clean it up (resources)
    - OS role in terminating process:
        - Reclaim any resources it may be holding: mem, lock, hw access
        - Inform any other process that needs to know:
            - Those waiting for interprocess communications
            - Parent/child process
        - Remove process Descriptor from the process table
- Running process
    - Process must execute code to do their job => OS must give access to processor core.
    - There are more process than cores => process share cores, cannot all execute instructions at once
    - Loading a process
        - To run process on core, core hw must be initialized
        - Load the core's registers, initialize stack, set %rsp
        - Set up memory control structure, set PC
    - How process run on OS
        - Process use execution model: limited direct execution
            - CPU directly executes most application code
                - Punctuated by occasional traps (for system calls)
                - w/ occasional timer interrupt (for time sharing)
            - Maximizing direct execution is always the goal
                - For Linux and Windows user mode processes
                - For OS emulation (e.g., Windows on Linux)
                - For virtual machines
            - Enter OS as seldom as possible, back to app asap
        - Most instructions are executed directly by the process on core
            - w/o OS intervention: otherwise slow++
        - some instructions instead cause a trap to the OS:
            - privileged instructions that can only execute in supervisor mode
            - OS takes care of things from there
    - Exceptions
        - Def: the technical term for what happens when the process can't run an instruction
        - Ex: interrupt (generated from hardware), signal (OS send ~ to process, like kill(), process should handle the signal)
        - Some exceptions are routine: EOF, arith overflow, conversion err
            - Should check for these after each operation
        - Some exceptions occur unpredictably
            - Seg fault, ^C, hang-up, power-failure => asynchronousExpt
        - Asynchronous Exceptions: unpredictable
            - Programs CANNOT check for them (random happennings)
            - Some languages support try/catch operations
            - Hardware & OS support traps: catch expt, ctrl to OS
            - OS use these for system calls

- Using traps for System Calls (PS carry error code)
  - Ways of using OS services: library (ALL in user mode), system calls, remote
  - Trap: hardware instruction come from processor
    - Look up in trap table done by hw
  - parameter: system call ID,
  - Made possible at processor design time, not OS design time
  - Reserve one privileged instruction for system
  - Define system call linkage conventions
    - Call: r0 = system call number, r1 points to arg
    - Return: r0 = return code, condition code: suc/fail
  - Prepare arguments for the desired system call instruction
  - Exe designed sys call instruction(cause expt that trap OS)
  - OS recognize and perform requested operation:
    - Entering the OS thru a point called a gate
  - Return to instruction after the system call



- Trap handling
  - Partially hardware, partially software
  - Hardware portion of trap handling
    - Trap cause an index into trap vector table for PC/PS
    - Load new processor status word, switch to supervisor mode
    - Push PC/PS of program that caused trap onto stack
    - Load PC (w/ address of 1st level handler)
  - Software portion of trap handling
    - 1st level handler pushes all other registers, gather info, select 2nd level handler
    - 2nd level handler actually deals with the problem
      - Handle event, kill process, return
- Trap and Stack
  - The code to handle trap is JUST code, though run in privileged mode
  - The user stack and kernel stack is separated, which is one of the underlying reason why trap exists
  - This protection allows you execute in the privileged mode that user mode has no way to touch



- Returning to User-Mode
  - 2nd level handler => 1st level handler == restore all resgister from stack => use privileged return instruction to restore PC/PS => resume user mode exe at next instr
  - Saved registers can be changed before return:
    - Change stacked r0 to reflect return code
    - Change stacked PS to reflect success/failure

- Asynchronous Events
  - Some things are worth waiting(read()wait for data, latency milisec)
  - Other time waiting doesn't make sense:
    - Do sth. Else while waiting, multiple operations outstanding, some events demand very prompt attention
    - Need event completion call-backs
      - This is a common programming paradigm
      - Computes support interrupts
      - Commonly associated w/ I/O devices & timers

        - User mode signal handling
            - OS defines types of signals: excpt, operator action, comm
            - Process can control their handling
                - Ignore, handler, default (kill/coredump)
            - Analogous to hardware traps/interrupt
                - But implemented by OS, delivered to user mode process
    - Managing process state: a shared responsibility
        - Process itself take care of its own stack & contents of memory
        - OS keep track of resources that have been allocated to the process:
            - Which memory segments, open files & devices, supervisor stack..
    - Blocked processes: processes NOT ready to run are blocked
        - One important process state element is whether a process is ready to run
        - Why a process is not ready:
            - Waiting for I/O
            - For some resource request to be satisfied
            - OS keep track of whether the process is blocked
        - Why block process:
            - Blocked/unblocked are notes to scheduler
            - Scheduler knows not to choose them, so other parts of OS know if they need to be unblocked later
        - ANY part of OS can set/remove blocks; process can ask to block itself thru system calls
        - Who handles blocking:
            - Usually happens in resource manager
                - When process need unavailable resources:
                    - Change scheduling state to "blocked"
                    - Call the scheduler and yield the CPU
                - When required resource becomes available:
                    - Change the process' scheduling state to "ready"
                    - Notify scheduler that a change has occurred

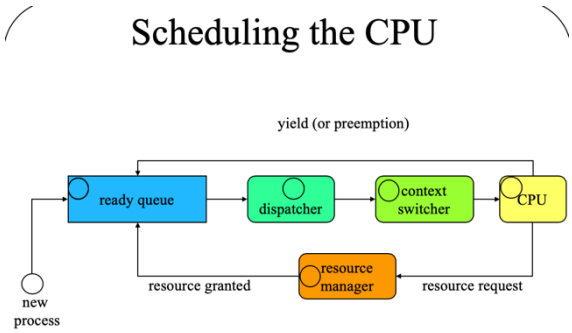
Scheduling

- What is scheduling
    - Def: Making decisions of what to do next, in particular:
        - One-client-at-a-time resource
        - Multiple potential clients
        - Who use resource next, how long?
    - Changing scheduling algorithms can drastically change system behavior
    - Process queue
        - The OS typically keeps a queue of processes that are ready to run
            - Ordered by whichever one should run next
            - Depend on the scheduling algorithms used
        - To schedule to new process, grab from queue
        - Un-ready process: not in queue OR at the end
    - Potential scheduling goals:
        - Maximize throughput: get as much work done as possible
            - When you want get a lot of process down quickly
        - Minimize avg waiting time: avoid delay too many for too long
            - Time sensitive jobs, e.g., interactive program in smartphone
        - Ensure some-degree fairness: min worst case waiting time
        - Meet explicit priority goals: scheduled items tagged with relative priority

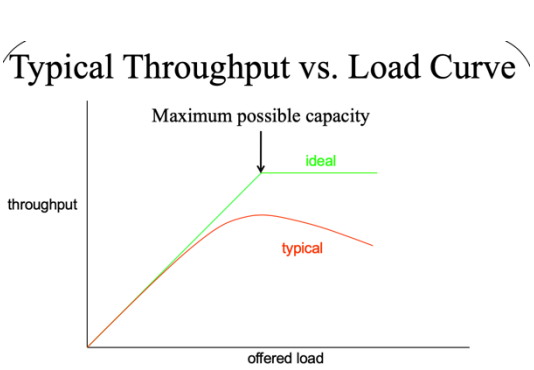
        - Real time scheduling: scheduled items tagged w/ ddl to be met
    - Different system, different scheduling goals
        - Time sharing:
            - Fast response to interactive programs
            - Each user gets an equal share of the CPU
        - Batch
            - maximize total system throughput
            - delays of individual processes are unimportant
        - Real-time
            - Critical operations must happen on time
            - Non-critical operations may not happen at all

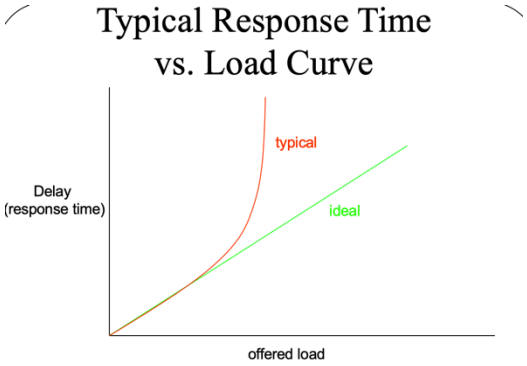
==========

- Scheduling policy and mechanisms
    - Dispatching: scheduler moving jobs into and out of processor
        - How dispatching is done should not depend on the policy used to decide who to dispatch
        - Desirable to deparate the choice of who runs (policy) from the dispatching mech
        - Desirable if OS process queue truscture NOT policy-dependent
    - Policy: high level guidance on which program should be switched into, logically
    - Mechanism: context switch, etc., mechanism
    - Principle: changes introduced should be as small as possible, change policy not mechanisms
    - Scheduling and performance
        - Performance: Depending on the goal, throughput, run time/latency, fair
            - May not optimize ALL
        - Scheduling performance has very different characteristic under light vs. heavy load
        - General comments on Performance:
            - Performance goals should be quantitative & measurable
                - Cannot optimize what can't measure
            - Metrics: the way & units in which we measure
                - Choose a characteristic to be measured
                    - Must correlate well with goodness/badness of servic
                - Find a unit to quantify that characteristic
                    - MUST be a unit that can be actually measured
                - Define a process for measuring the characteristic
        - Quantifying scheduler performance
            - Candidate metric: throughput (processes/second)
                - BUT: different process need different runtime
                - Process completion time not controlled by scheduler
            - Candidate metric: delay (milliseconds)
                - BUT: specifically what delay should measure
                    - Time to finish a job (turnaround time)
                    - Time to get response
                - Some delays are not scheduler's fault:
                    - Time to complete a service request
                    - Time to wait for a busy resource
            - Mean time to completion (seconds)



        - For a particular job mix, i.e., benchmark
    - Throughput (operation/second)
        - For a particular activity or job mix (benchmark)
    - Mean response time (milliseconds)
        - Time spent on ready queue
    - Overall "goodness"
        - Require customer specific weighting function
        - ⇔ Service Level Agreement (SLA)
- Example: Measuring CPU scheduling
    - Process execution can be divided into phases:
        - Time spent running: process control how long
        - Time spent waiting for resources or completions: resource managers control how long
        - Time spent waiting to be run: controlled by scheduler
    - Proposed metric: time ready process spend waiting for CPU





    - Why cannot achieve ideal throughput
        - Scheduling is not free
            - It takes time to dispatch a process -> overhead
            - More dispatches means more overhead -> lost time
            - Less time(/second) is available to run process
        - To minimize performance gap:
            - Reduce overhead per dispatch -> change mech
            - Minimize the number of dispatch/sec->change poli
    - Why response time explode
        - Real system have finite limits; e.g., queue size
            - If # of process exceed queue size: graceful degradation
        - When limits exceed requests are typically dropped
            - Which is an finite response time for them
            - automatic retries (e.g. TCP) may be dropped
        - if load arrive a lot faster than it is serviced, lots get dropped
        - overhead may explode during periods of heavy load
- graceful degradation
    - system is overloaded when it no longer meet service goals
    - Possible solution to overload:
        - Continue service but with degraded performance: bad performance
        - Maintain performance by rejecting work: user get annoyed, maintain current process's performance level
        - Resume normal service when load drop to normal
    - Should NOT:

- Allow throughput to drop to 0 (stop doing work, no process get served)
- Allow response time to grow w/o limit

==========

- What resources should we schedule
- Sample Scheduling Algorithms & Implications
  - Preemptive vs. Non-preemptive Scheduling
    - Preemptive: scheduler temporarily halts running work to run sth. else
      - Ex: time sharing scheduling
      - HIGHLY DYNAMIC! CANNOT MEET DDL!
      - Process run until it yields or OS decide to interrupt, at which point some other process/thread runs and the interrupted restart later
      - Clock: a hardware to generate interrupt every some time interval to the processor
      - Implications:
        - A process can be forced to yield at any time
          - If a more important process is ready e.g. result of IO completion interrupt
          - If running process's importance lowered e.g. result for too having running for too long
        - Interrupt process might not be in clean state
          - Which could complicate saving & restore its state
        - Enables enforced fair share scheduling
        - Introduce extra context switches: not required by dynamics of process
        - Creates potential resource sharing problems
      - Implementations:
        - Need a way to get control away from process
          - Ex: process make sys call, or clock interrupt
        - Consult scheduler before returning to process
          - Any process awaken/priority raise/lowered?
        - Scheduler finds highest priority ready process
          - If cur, return as usual; if not, yield and switch
      - Clock interrupt
        - Modern processor contain a peripheral device (clock) with limited powers, which can generate interrupt at fixed time interval, temp halt running process
        - Ensure runaway process doesn't keep control forever
        - Key tech for preemptive-scheduling
      - Round Robin scheduling algorithm

Round Robin Example

Assume a 50 msec time slice

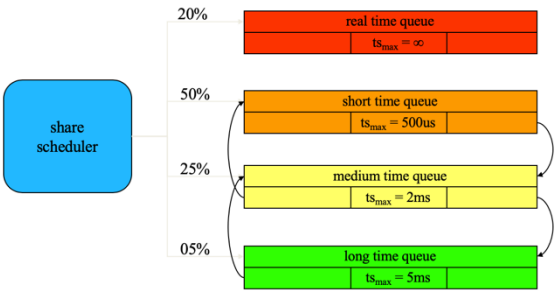
Process	Length	1st	2nd	3d	4th	5th	6th	7th	8th	Finish	Switches
0	350	0	250	475	650	800	950	1050		1100	7
1	125	50	300	525						550	3
2	475	100	350	550	700	850	1000	1100	1250	1275	10
3	250	150	400	600	750	900				900	5
4	75	200	450							475	2
Average waiting time: 100 msec										**1275**	**27**

First process completed: 475 msec

      - Goal: fair share scheduling
      - All process offered equal share of CPU
      - All process experiences similar queue delays
      - All process are assigned a nominal time slice(usually same)
      - Each process is scheduled in turn
      - Runs until blocks/time slice expire, put to queue end
      - Eventually, each process reaches front of the queue
      - Properties:
      - quicker chance to for SOME computation
        - Cost: not finishing any quickly
        - +: for interactive process

        - Far more context switches: can be expensive
        - Average wait time shorten, delay for already served
    - With I/O interrupts
        - Process get halted by round robin scheduling if their time slice expires. If they block for I/O or anything else on their own, the scheduler doesn't halt them. Thus, sometimes acts not different than FIFO
    - Choosing time slice: VIP for performance!!
        - Long time slices avoid too many context switch which waste cycles -> netter throughput & utilize, no different to non-preempt
        - Short time slice->better response time
    - Cost of context switches:
        - Enter OS: take interrupt, save registers, call scheduler
        - Cycles to choose who to run: scheduler/dispatch work to choose
        - Moving OS context to new process: switch stack, non-resident process description
        - Switching process address spaces: map-out old process, ~new~
        - Losing instruction and data caches: slow down next 100 ins
- Multi-level feedback queue (MLFQ) scheduling
    - Profiling (historical data) to decide which queue to put it in, short or long
    - Benefits: acceptable response time for interactive jobs
        - Other jobs w/ regular external inputs
        - Won't be too long before they're scheduled
        - But they won't waste CPU running for a ling time
    - Efficient but fair CPU use for non-interactive jobs
        - Run for long time slice w/o interrupt
    - Predictable real time response: based on known % of CPU
    - Dynamic & automatic adjustment of scheduling based on actual behavior of jobs
    - One time slice length may not fit all process -> create multiple queue
        - Short time (foreground) tasks that finish quickly
            - Short but freq time slices, optimize response time
        - Long time (background) tasks that run longer
            - Longer but infreq slices, min overhead
        - Different queue get different shares of CPU
    - Find balance between good response time & turnaround time
    - Decide which queue to put new process:
        - Start all processes in short time queue; move to longer queue if too many time-slice ends; move back to shorter queue if too few time slice ends; process dynamically find the right place
        - If you also have real time tasks, you know where they belong; start them in real time queue and don't move them
- Priority Scheduling Algorithms
    - Assign each job a priority number, run according to #
    - If non-preemptive: nothing special just ordered by priority
    - If preemptive: when new process is created, it might preempt running process if its priority is higher
    - Source of priority: user specify, or OS decide
    - Problems:

Multiple Queue Scheduling



        - Possible starvation; low priority process may never run
        - May make more sense to adjust priorities:
            - Process that have run for long time -> lowed
            - Process that have not been able to run for long –> raised
    - Hard vs. soft priority
        - Hard: higher priority has absolute precedence over lower
        - Soft: higher priority should get a larger share of resource
    - Priority scheduling in Linux
        - Each process in Linux has a priority (nice value), a soft priority describing share of CPU that a process should get
        - Commands can be run to change process priorities
        - Anyone can request lower priority for his process
        - Only privileged user can request higher priority
    - Priority scheduling in Windows
        - 32 different priority levels: half for regular, half for soft real-time
            - Real time scheduling requires special privileges
            - Using a multi-queue approach
        - User can choose from only 5 of these priority levels
        - Kernel adjusts priorities based on process behavior:
            - Goal of improving responsiveness
- +:
    - Good response time
    - Produce fair usage
    - Good for real-time and priority scheduling
- -:
    - Complex: has to implement context-switch, more bugs
    - Requires ability to cleanly halt process & save its state
    - Poor throughput
    - Possibly higher overhead

- Non-preemptive: scheduled work always runs to completion
    - Description: scheduled process tun until it yields CPU
        - Work well for simple system w/ small # of process, w/ natural producer/consumer relationships
        - Producer/consumer relationship: process B's input is the output of process A; e.g., pipe()
    - YIELD: free the CPU resources
    - Algorithms:
        - First come first served: simplest, use queue
            - Run first process on ready queue until it complete
            - Highly variable delays: depend on process impleme
            - All process will eventually be served
            - Serve well when:
                - Response time unimportant e.g. batch
                - In embedded system e.g. telephone/set-top box: computation brief, natural P/C relationship
        - Shortest job next
        - Real-time schedulers
            - For certain systems some things MUST happen at time: e.g., industrial control system, If you don't rivet the widget before the conveyer belt moves, you have a worthless widget

First Come First Served Example

Dispatch Order		0, 1, 2, 3, 4	
Process	Duration	Start Time	End Time
0	350	0	350
1	125	350	475
2	475	475	950
3	250	950	1200
4	75	1200	1275
Total	1275		
Average wait		595	

Note: Average is worse than total/5 because four other processes had to wait for the slow-poke who ran first.

- => scheduled base on real-time ddl: either soft/hard
- Hard read time schedulers
  - The system absolutely MUST meet its ddl
  - By def, system fails if ddl is not met
  - Solution: careful analysis
    - No possible schedule cause ddl miss
    - Working it out ahead of time
    - Then schedule rigorously by ddl
    - Use static instead of dynamic analysis of the worst =>Source code MUST be deterministic
    - Impossible when: loop (# of iteration unset)

Ensuring Hard Deadlines

- Must have deep understanding of the code used in each job
  - You know exactly how long it will take
- Vital to avoid non-deterministic timings
  - Even if the non-deterministic mechanism usually speeds things up
  - You're screwed if it ever slows them down
- Typically means you do things like turn off interrupts
- And scheduler is non-preemptive
- Typically you set up a pre-defined schedule
  - No run time decisions

- Soft real time scheduler
  - Highly desirable to meet ddl, but can
  - Goal: avoid missing ddl
  - Different class of ddl, some "harder"
  - Less analysis
  - Depends on the particular type of system, might just drop the job whose ddl you missed, might allow system to fall behind, might drop some other job in the future. At any rat, it will be well defined in each particular system
  - Algorithms: earliest ddl first
    - Keep job queue sorted by ddl
    - Prune queue to remove missed ddl
    - Goal: minimize total lateness
    - No static analysis required
  - Example: video playing device
    - Frames arrive: from disk/network
    - Ideally, each frame should be rendered on time to achieve highest user perceived quality
    - If you can't render a frame ontime, might be better to skip it completely, rather than fall further behind

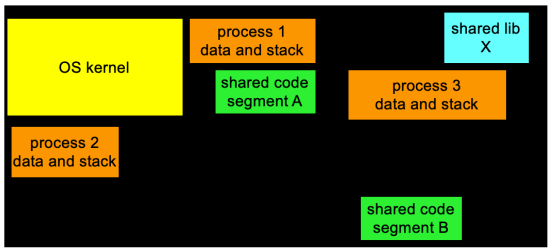
Example of a Soft Real Time Scheduler

- A video playing device
- Frames arrive
  - From disk or network or wherever
- Ideally, each frame should be rendered "on time"
  - To achieve highest user-perceived quality
- If you can't render a frame on time, might be better to skip it entirely
  - Rather than fall further behind

- +:
  - Low scheduling overhead
  - Tend to produce high throughput
  - Conceptually very simple
- -:
  - Poor response time for processes
  - Bugs can cause machine to freeze up (inf. loop)
  - Not good fairness: Piggy process can starve others
  - May make real time and priority scheduling difficult/useles

Memory Management

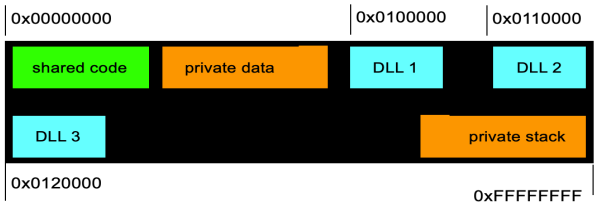
- Physical Memory Allocation



Physical memory is divided between the OS kernel, process private data, and shared code segments.

- Physical and virtual addresses
  - A cell of RAM has a particular physical address
  - But we can have processes use virtual address which may NOT be the same
    - More flexible, but require V->P translation
  - Pointers are VIRTUAL address!

- Virtual Memory

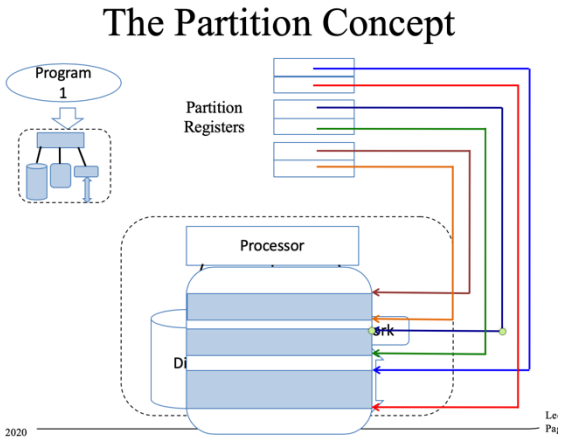


All of these segments appear to be present in memory whenever the process runs.
Note this virtual address space contains no OS or other process segments

- Memory management
  - One of the key assets used in computing
  - Memory abstractions that are usable from a running program, i.e. RAM
    - Limited amt, lots of process need it
  - Memory management goals:
    - Transparency
      - Process sees only its own address space
      - Process is unaware that memory is being shared
    - Efficiency
      - High effective memory utilization
      - Low run-time cost for allocation/relocation
    - Protection and isolation
      - Private data will not be corrupted
      - Private data cannot be seen by other processes
  - Memory Management Problems:
    - Most process cannot perfectly predict how much memory they will use
    - Processes expect to find their existing data when they need it where they left it
    - The entire amt of data required by all processes > amt of available PM
    - Switching between process MUST be fast: can't afford much delay for copy data
    - Cost of memory mgmt. MUST NOT be too high

- Why do we need mem mgmt.?
  -

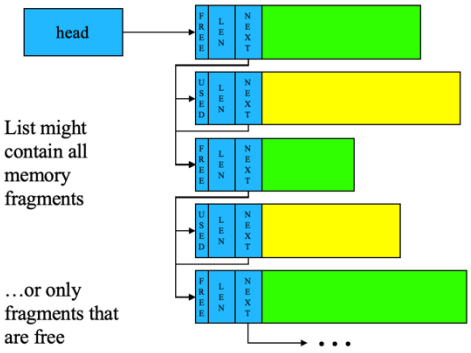
- Memory Management Strategy
  - Fixed partition allocations: in PA
    - Pre-allocate partitions for n processes
      - 1+ per process, reserve space for largest possible process
    - Partitions come in 1 or a few set sizes
    - +: easy to implement
      - Ex: common in old batch systems, alloc/free cheap and easy
    - NO virtual addresses used!
    - Problems:



      - Presumes you know how much memory will be used ahead of time -> inflexible
      - Limits the number of process supported to the total of their memory requirement
      - Not great for sharing memory
      - Fragmentation causes inefficient memory (internal frag)
      - Not used in many modern systems, but possible option for special purpose sytem
        - Ex: embedded systems where know exactly what our memory needs'll be
    - Fragmentation: problem for all mgmt. systems, esp here!
      - Based on processes not using ALL the memory they requested
      - Result: can't provide memory when theoretically can
      - Internal fragmentation:
        - Occur whenever you force allocation in fixed sized chunks
        - Wasted space inside fixed sized blocks:
          - Requestor was given more then needed
          - Unused part is wasted and can't be used for others
        - Caused by mismatch between:
          - Chosen size of a fixed-sized block
          - Actual sizes that programs use
  - Dynamic partitions allocation: in PA
    - Like fixed partitions, except:
      - Variable sized, usually any size requested
      - Each partition has contiguous memory address
      - Process have access permissions for the partitions
      - Potentially shared between processes
    - Each process ca have multiple partitions w/ diff size & characteristics
    - Problems:
      - NOT relocatable. Once a process has a partition, can't easily move its contents
        - E.g. want to expand partition -> can't in-place
        - E.g. compaction (coalesce) to reduce fragmentation
      - Not easily expandable
      - Impossible to support applications with larger address space than PM
      - Subject to fragmentation
    - Partition and expansion
      - Partitions are tied to particular address ranges at least during execution
      - Can't just move contents of a partition to another set of addresses:
        - All pointers in the contents will be wrong
        - Generally don't know which e=memory locations contain ptr

- Hard to expand because may not be space "nearby"
    - Partitions allocate on request, process may ask for new one later, but partitions that have been given cannot be moved spl else in memory since all space after a given partition may have already been allocated
    - Source of problem: contiguous mem allocation! (array-like)
    - Solution: divide partition into small chunks and make them to LL

- Keep track of variable sized partitions
    - Start with one large HEAP of memory
    - Maintain free list: system data structure to keep track of pieces of free mem
    - When a process requests more memory:
        - Find a large enough chunk of memory
        - Carve off a piece of requested size
        - Put the remainder back on the free list
    - When a process frees memory: put it back to free list



    - Managing free list:
        - Fixed sized blocks are easy to track: bit map indicating alloc/free
        - Variable chunks require more info:
            - LL of descriptors, one per chunk
            - Each descriptor lists the size of the chunk and whether free
            - Each has a ptr to the next chunk
            - Descriptors often at front of each chunk
            - Allocated memory may have descriptors too
        - Variable partitions and fragmentation
            - Variable sized partition not as subject to internal fragmentation
                - Unless requestor asked for more than he will use
                - Which is actually common but at least manager gave him no more then he requested
            - Subject to EXTERNAL fragmentation
        - External fragmentations
            - Each allocation creates left-over chunks, smaller ++ as time
            - Small left-over fragment are useless: frag
            - Solutions:
                - Try not to create tiny fragments
                - Try to recombine fragments into big chunks
            - Try not to create tiny fragments
                - 4 algorithm: best/worst/first/next fit
                - Best fit: search for the best-fit chunk w/smallest diff
                    - +: might find a perfect fit
                    - -: search entire list every time; quickly create small frag
                - Worst fit: search for largest size >= equal to request
                    - +: tend to create large fragment for a while
                    - -: search entire list
                - First fit: worst case still O(N), but often much better
                    - +: short search, create random sized fragments

        - -: the first chunks quickly fragments, searches become longer, ultimately it fragments as badly as best fit
    - Nest fit: MOST MODERN SYSTEM!!
        - Tries to get advantage of both first and worst fit
            - Short search (maybe better than first)
            - Spreads out fragmentation like worst fit
        - Guess pointers are a general technique
            - Think of them as lazy (non-coherent) cache
            - If right, save lots of time
            - If wrong, algorithms still work
            - Can be used in side range of problems
- Coalescing: Try to recombine fragments into big chunk
    - All variable sized partition allocation algorithms have external fragmentations; some faster, some spread out
    - To reassemble fragments:
        - Check neighbors whenever a chunk is freed
        - Recombine free neighbors whenever possible
        - Free list c be designed to make this easier
    - Motivation: reduce external fragmentation
    - Challenges:
        - Need to copy, and problem of pointers
- Fragmentation and Coalescing:
    - Opposing process that operate in parallel
    - Coalescing works better with more free space
    - Chunks held for a long time cannot be coalesced
    - High variability increases fragmentation rate
    - Fragmentation get worse over time

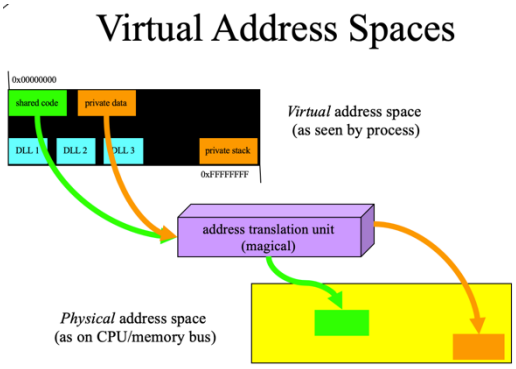
- Summary:
    - Eliminates internal fragmentation: request variable
    - Implementation more expensive
        - Long search for complex free list, carving & coalesce
    - External fragmentation inevitable, coalescing counteract

- Memory request is mot randomly sized
    - Reason: key services use fixed sized buffer
        - File systems (for disk I/O)
        - Network protocols (for packet assembly)
        - Standard request descriptors
    - OS: handle then separately -> buffer pools
    - Buffer pools
        - If there are popular sizes: reserve special pools of fixed size buffers, satisfy matching requests form those pools
        - Benefit: improved efficiency
            - Much simpler than variable partition allocation:
                - no searching, carving, coalescing
            - Eliminates/reduced external fragmentation
        - Must know how much to reserve:
            - Too little: buffer pool becomes a bottleneck
            - Too much: unused buffer space
        - ONLY satisfies perfectly matching requests!

        - Otherwise, back to internal fragmentation
    - How are buffer pools used
        - Process requests a piece of memory for a special purpose
            - Ex: to send a message
        - System supplies one element from buffer pool
        - Process uses it, completes, frees memory
            - Explicitly/implicitly based on buffer size
            - Ex: sending the message will free the buffer behind the process back once the message is sent
    - Dynamically sizing buffer
        - If we run low on fixed sized buffers:
            - Get more mem form the free list
            - Carve it up into more fixed sized buffers
        - If free buffer list get too large:
            - Return some buffers to the free list
        - If the free list gets dangerously low
            - Ask each major service with a buffer pool to return space
        - This can be tuned by a few parameters:
            - Low space threshold: need more
            - High space threshold: have too much
            - Nominal allocation: what we free down to
        - Resulting system: highly adaptive to changes
    - Problem: Lost Memory (Memory Leak)
        - Memory leak: process done w/ the mem but doesn't free it
        - Also a problem when a process manages its won memory space
            - i.e.: Allocate a big area, maintain its free list
        - Long running processes with memory leaks can waste a lot mem
        - Solution: Garbage Collection
    - Garbage Collection
        - Do not count on process to release memory, monitor how much free memory we have got; when run low, start GC:
            - Search data space finding every object pointer
            - Note address/size of all accessible objects
            - Compute the compliment (what is in accessible)
            - Add all inaccessible memory into free list
        - Finding all accessible memory:
            - Object oriented languages often enable this:
                - All object references are tagged
                - All object descriptors include size information
            - Often possible for system resources where all possible references are known:e.g.we know who has which file open
        - General garbage collection: might be difficult
            - Find all pointers in allocated memory
            - Determine "how much" each points to
            - Determine what is and is not still pointed to
            - Free what isn't pointed to
        - Problems with general garbage collection: a location in data/stack segments might seem to contain addresses, BUT:

- Might not be true pointers but other data types whose values happen to resemble address
- Pointers themselves might not be available
- Might be able to infer this recursively for pointers in dynamically allocated structures
- How about pointers in statically allocated (global) areas?

- Relocation
  - Compaction and Relocation
    - Garbage collection is just another way to free memory, don't help much in frag
    - Ongoing activity can starve coalescing: chunks realloc before neighbors freed
    - Could stop accepting new allocations but resulting convoy on memory manager would trash throughput
    - We need a way to rearrange active memory:
      - Re-pack all process in one end of memory
      - Create one big chunk of free space at other end
  - Relocation:
    - The ability to move a process:
      - From region initially loaded into a new and diff region of mem
      - NOT right: all address in the program will be wrong
        - References in the code segments:
          - Calls/branches to other parts of the code
          - References to variables in the data segments
        - New pointers created during the process that pnt into data/stack seg
    - Relocation problem:
      - It is generally NOT feasible to relocate a process
      - May be we can relocate references to code if we kept the relocation info around, but how can we relocate references to data (i.e. ptr)?
      - We can NEVER find/fix all address of references like general GC
      - Solution: make processes location independent
      - Memory segment relocation
      - Natural model:
        - Process address space is made up of multiple segments
        - Use segment as the unit of relocation
        - Long tradition from IBM to Intel
      - Computer has special relocation registers
        - Called segment base registers
        - Point to the start of each segment
        - CPU automatically adds base register to every address
      - OS uses these to perform virtual address translation
        - Set base register to start of region where program is loaded
        - If program is mode, reset base registers to new registers
        - Program works no matter where its segments are loaded
      - Relocation and safety
        - A relocation mech like base registers is good
          - Solves the relocation problem
          - Enables use to move process segments in physical memory
          - Such relocation turns out to be insufficient



- Need protection: Prevent process from reaching outside its allocated memory e.g. overrun end of mapped segment
- Segments need a length/limit register
  - Specify max legal offset from the start of the seg
  - Any address greater than this is illegal
  - CPU should report it via seg exception i.e. trap
- How much problem relocation solved:
  - Can use variable sized partitions butting down on internal fragmentation
  - We can move partitions around:
    - Helps coalescing more effectively
    - Still require contiguous chunk of data for seg
    - So external frag still a problem