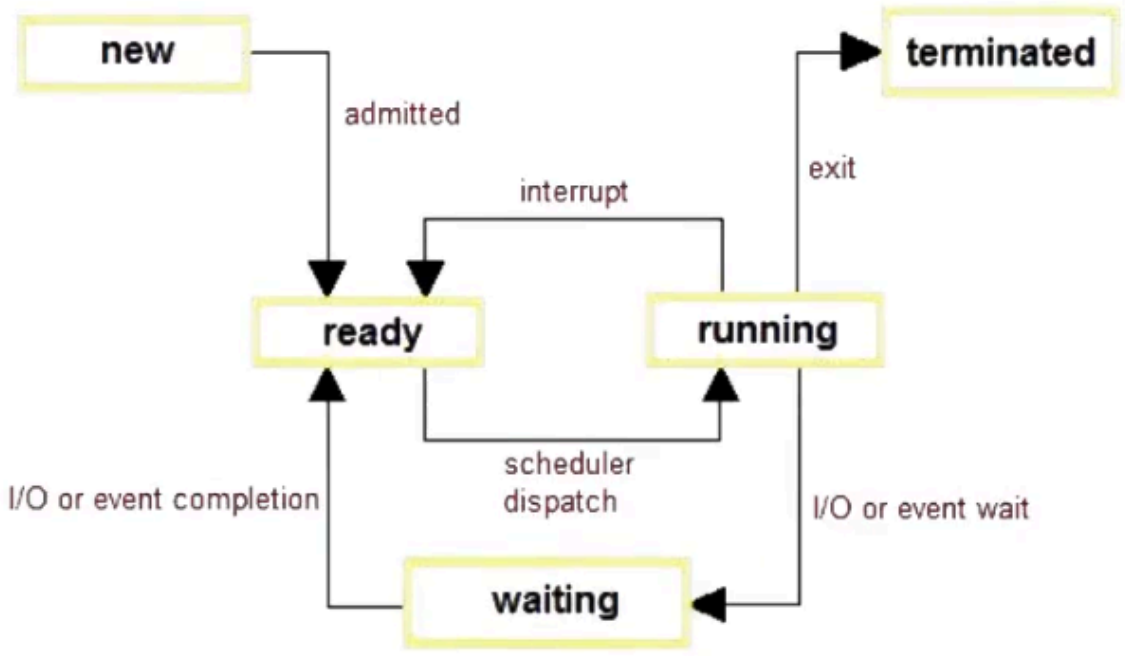
Sources

- Kerrisk, Michael - The Linux programming interface a Linux und UNIX system programming handbook (2018, No Starch Press)
- UNIX and Linux System Administration Handbook, 4th Edition
- Linux Kernel Development 3rd Edition - Love - 2010
- Andrew S. Tanenbaum - Modern Operating Systems
- Partially: [Addison-Wesley Professional Computing Series] W. Richard Stevens, Stephen A. Rago - Advanced Programming in the UNIX Environment
- https://pages.cs.wisc.edu/~remzi/OSTEP/

Operating System

Overview

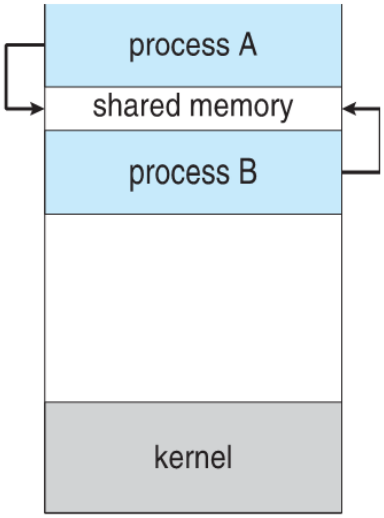
- processes



- Kernel space - kernel handles schedugling of tasks, buffering (saving data when transferring devices/cpu), spooling (holding output for i/o), etc.
- Io software layers - includes user io/user processes, device-free software, device drivers, interrupt handlers, hardware
- Blocking - app stops all other execution to wait for io
    - No io queue

- Non blocking - put into queue to wait for io so free up cpu
- Memory mapped io - io/cpu share memory space
- Micro vs monolithic - micro is many different kernels with different address spaces, often over network while monolithic is one in one address space
    - Linux is mono
- PCB is process control block; holds this info:
    - Next instrucytion
    - Pid
    - Process state: ready, waiting, new, running, terminated
    - priority/ptr to scheduling queue
    - Memory management info
    - Stats (cpu consumed, limits, etc.)
    - i/o info: devices/files used
    - Each process has this:
        - Heap for dynamic alloc
        - Stack for local
        - Text for the literal code
        - Data - static/global var
- Parent process can terminate child if orphanprocess is not possible, if memory limit exceeded, result isn't needed
- Kernel is first to be created; all are usually children of kernel at least
- scheduler
- Scheduling process: job queue -> ready queue -> cpu does work -> either exit, add back to ready queue, or go to i/o queue and i/o
    - Queue is just linkedlist
    - i/o, exit, or wait for cpu
- Long term scheduler - sends user-submitted process from new to ready queue (loads them into memory)
    - Average incoming rate == avg outcoming rate
    - Determines how many processes can run at once (degree of multiprogramming)
- Short term/cpu scheduler - moves from ready to running queue
    - Allocates cpu; is very fast
- Midterm scheduler - removes process from memory and swaps out with another. (is responsible for reversing this)
    - Used for reducing multiprocessing (swapping)
- These are different from context switching
    - Context switching is faster on risc than overlapped register windows
-
- Interrupt
- Interrupt for system call or preemption
- Need to save context in pcb
- Context switch - Save curr context and restore prev

- Interprocess communication
- Shared memory - memory is not shared between processes, but this allows comm buffer
    - This solves underflow (not enough producing) and overflow (not enough consumer) of producer consumer dynamic
    - Unbounded buffer - producer produces unlimited (consumer may have to wait)
    - Bounded buffer - must take turns between producer/consumer



    - Msg passing - message queue: processes publish to one end and read from other to communicate msgs
- Scheduling
- Arrival time - time at which arrives at ready queue
    - Completion time - time it finishes running
- Burst time - time cpu takes to process
- Wait time - time process waits in ready queue
- Turnaround - time it took waiting + computing (complete - arrival)
- Throughput: number of processes executed per unit of time
- Some can be preempted once in ready queue; some can't
- Scheduling algorithms
- First come first serve
- Literally as it says
- Bad since if big process comes first then slow down os
- Sjs
- Shortest job first
- Bad cuz starves large onces if we keep on getting short jobs
- Best to make it preemptive otherwise will have large wait times; preempt while waiting for i/o or something
- Round robin
- See linux implementation
- Multi level queue
- Different levels of priority like tier list

- Shortest remainingt ime
- Synchronization
- Mutual exclusion, progress: need to ensure holder uses it, then bounded waiting: bound wait queue/wait time
- Critical section - section where shared resources can cause race conditions
- Cannot just disable interrupts when entering a critical section; could be used maliciously
- So therefore use lock/mutex
- Peterson's solution: have flag that set to true before enter critical section, then remove once done. This way only one per in critical section
- Semaphores - wait until s >0. Then decrement
    - Allows for x to use resource
    - If x is 1, then is mutex (mutual exclusion)
- Busy waiting - unbounded wait time. Block it so isn't just waiting and wasting cpu (that way others can use cpu)
- Deadlock - circular dependency list (need resource that another holds but that other is waiting for resource that curr is holding (but won't release until it gets the other))
    - Caused by: mutual exclusion, hold while wait, no preemption
- How to prevent deadlocks:
    -
- Banker's algorithm
- Threads - smallest set of instructions that can be managed independently by scheduler
- Are faster to start/cleanup, share memory, lightweight; however, can create race conditions/deadlocks
- User level thread: fast context switching due to no kernel intervention, but kernel does nto see threads as separate entities; thus some system calls may be starved. One thread blocks other threads
- Kernel level thread - kernel knows these threads. *mor switch
    - Thread blocks itself only. Slow termination/creation/switching
- Mapping of user to kernel
    - One to one - good but expensive
    - Many users to one kernel - can be blocked by one system call
    - Many to many (rare)
- One thread per core does the best
    - https://stackoverflow.com/questions/1718465/optimal-number-of-threads-per-core
- Dispatcher
-

Memory

- Fragmentation - holes are too small to fit chunks in
    - External -creates large number of nontinguous small holes that are too small to fit chunks in
        - First fit/best fit suffer from external (creates small unusable holes)

            - Compaction, segmentation, paging with moderation can be used to beat this
            - Usually for disk since large files are split up
            - Usually for ram this is due to programs being loaded/unloaded
        - Internal - unused space in fixed storage units like blocks
            - Best fit can beat this
            - Usually in ram due to using random access more
- Paging
    - Fixed blocks stores data; need page number and offset
    - Page size is determined by hardware
    - Is faster than segmentation
    - Causes internal fragmentation (large holes)
    - List of free frames kept track by os
- Segmentation
    - Variable blocks store data; need section number and offset
    - Size determined by user
    - List of holes kept track by os
    - Causes external fragmentation (too small holes)
    - Slower than paging
- Dynamic loading - load only when needed
- Dynamic linking - load based on dependencies of module x
- Contiguous - memory stored in one chunk. This creartes holes
    - First fit, best fit (min), worst (biggest possible) fit are all algos
- Static load - load program before execute
- dynamic/demand loading - load only when need
- Segmentation - noncontiguous memory
- Hierarchy of memory

Virtual memory

- Paging - mapping physical to virtual address (virtual memory so store more than can in memory than physical limit)
- Thrashing - spending more time paging than computing
    - Fix with page fault frequency. If too many page faults, then give more memory frames. Can remove frame once frequency is low enough
- Demand paging -paging on demand only and the memory that need
- Page replacement - either queue or done with some other lru algo~
- Benefits
    - Allows for switching/multiprogramming
    - Allows for larger than physical memory programs
    - Faster at times due to only needing particular segment of memory
- Drawbacks
    - Slower than ram
    - Most of the time is slower in general
    - Less stability

- # File system
- In linux, files/directories are represented by inodes, which point to others in DAG tree fashion
    - file attribute:lots of ls -l info
    - direct , indirect_1, 2, 3 mappings:
        - Indirect to handle larger files so not as many block pointers in inode
            - Pointer to pointer that has many more pointers
        - Direct for smaller to avoid wasting pointer overhead
    - Logical block number - is local block index so can have file blocks in order but not worry about physical positioning. Many per file; specific to file
    - Direct mappings: Entries containing direct forward mappings. These mappings provide a mapping from the logical block number to the location of the physical block on disk
    - Indirect_1 Mappings: Entries containing indirect mapping requiring one level of indirection. These mappings map a logical block number to a table of direct forward mappings as described above. This table is then used to map from the logical block number to the physical block address.
    - Indirect_2 Mappings: Entries containing indirect mappings requiring two levels of indirection. These mappings map a logical block number to a table containing Indirect_1 mappings as described above
    - Indirect_3 Mappings: Entries containing mappings requiring three levels of indirection. These mappings map a logical block number to a table containing Indirect_2 mappings as described above.
    -
- Sequential access - next ptr to read files
- Direct access - use index
- Indexed sequential - index + next ptr
- Name, identifier, type, location, size, protection, time/date
    - Usage, mounting for dir
- https://cseweb.ucsd.edu/classes/sp16/cse120-a/applications/ln/lecture15.html
- Alias - having multiple names for one file for version management/sharing names, etc.
    - Early binding/hard binding - link created at file creation
    - late/soft/symbolic links - target is determined on each use. can dangle (point to dead object)
- Hard links require low level names
    - Implemented via inode - low level struct that represents file
    - Directory is DAG; even if follow symbolic link, going back up one goes to its parent
- File descriptor table (process specific) - list of indices that point to open file table (system wide)
    - 0 is input, 1 is output, 2 is error
    - Open file table is doubly linked list to ensure proper size control
    - Has:

            - Current offset, reference count, file permissions, read-only/write-only, in-ram version of node pointer, pointer to structure with file operations like read/write/close
                - read/write pointer maintained by open file table so know when write happens and so read doesn't affect file position in another process
- Open file table is so can see other file entries, but it is still one to one entry to file descriptor, meaning each process has its own read pointerf
- Memory mapped io
    - Reduces overhead for reading from file every time by mapping memory of file into process addresses. Reading from file means call kernel and that has to write i/o
        - Only pages needed mapped
        - MAP_PRIVATE ensures that pages are duplicated on write, ensuring that the calling process cannot affect another process's view of the file.
        - MAP_SHARED does not force the duplication of dirty pages -- this implies that changes are visible to all processes
        - Mmap maps, munmap unmaps
    - This can force pages out of memory since competes with others
    -
- open()/close() creates/closes sessions so that session overhead created once and shared by many other processes
        - Requires: pathname resolution, current offset for read, long term state management
- Forking copies exact file descriptor, meaning children/parents have the same file instances. What they read/write affects eawch other
- Ext4
- Is a common file system manager used for linux
- How it allocates files:
    - Extent allocation
        - Has table of available blocks and extents
        - Is for larger files
        - Allocates by extents instead of block by block
        - Extents are contiguous range of blocks
            - Uses fixed size blocks
        - May have to split up still but way less. Adding onto file is likewise
        - Tries to allocate in contiguous manner. Sometimes large files split up, esp when already fragmented
    - Block allocation
        - Like extent but with blocks
        - Usually for smaller files
    - Delayed allocation
        - Delay allocating until file written to disk

        - Allows for “seeing into the future” so no constant changes and corrections
    -
- Extent based allocation
- Block cache
- Is cache for files (in ram)
    - Is kept track of in inode’s address space
        - This tells where the cache is
- Cache hit - no need to read from disk
- Cache miss - read from disk, then store in cache
- Writing to file - write to cache, then cache is marked as dirty. When evicted, if is dirty, is written back
    - This is called flushing
    - Is periodically done outside of eviction anyways
- Uses lru
- Types of file systems
- virtual
- Tmpfs:
    - A temporary file system that stores data in the system's virtual memory (RAM).
    - Files are lost when the system reboots.
    - Commonly used for temporary storage like /tmp or /run.
- Sysfs:
    - A virtual file system that dynamically represents the system's devices and their configurations.
    - Mounted at /sys in Linux.
    - Provides a structured view of kernel objects like devices, drivers, and buses.
- Procfs:
    - Virtual file system
    - Mounted at /proc.
    - Represents system and process information as files (e.g., /proc/cpuinfo).
    - Useful for monitoring and debugging.
- Devfs:
    - Represents device nodes (e.g., /dev).
    - Dynamically manages device files.
- Cgroupfs:
    - Mounted at /sys/fs/cgroup.
    - Manages control groups for resource allocation and monitoring.
- Debugfs:
    - Mounted at /sys/kernel/debug.
    - Provides debugging information for kernel developers.
- Disk-Based File Systems:
- Ext4:
    - Default file system for many Linux distributions.

    - Journaling support for crash recovery.
- NTFS:
    - File system used by Windows.
    - Supports advanced features like file compression and encryption.
- FAT32/ExFAT:
    - Simple, widely compatible file systems.
    - Commonly used in USB drives and SD cards.
- XFS:
    - High-performance journaling file system.
    - Suitable for large files and scalability.
- Btrfs:
    - Modern file system with features like snapshots, pooling, and data integrity checks.
- ## Types of file allocation strategies
- Have much larger files than ram so need to accommodate scale
- Contiguous Allocation: Files are stored in contiguous blocks, but this suffers from external fragmentation and inefficient file growth.
- Linked Lists: Used to store files in non-contiguous blocks, but access is slower because of the need to follow pointers.
- File Allocation Table (FAT): A table where each entry points to the next block in the file. It avoids fragmentation but requires substantial memory to store the table, especially for large disks.
- Inode-Based Allocation: Another approach to managing file blocks, where each file has an inode containing metadata and pointers to data blocks.
- ## ELF
- Executable and linkable format (ELF) - file used to create executable
    - Contents
        - Elf header - has metadata like little/big endian (readelf -h)
        - Program header - has all the address segments to load such as code/stack. Specifies to os which parts of file should be in memory (virtual memory) and which parts are readable, executable, writable
        - Section header - outlines logical divisions in text so linker knows what section is code, which are variables, etc.
        - Text - the instructions (read-execute)
        - rodata/data - variables (read-write) and rodata (read only data)
        - Bss - used during runtime to store global/static (zeroed out during compile)
        - https://www.baeldung.com/linux/executable-and-linkable-format-file
-

- i/o

Caches

- Cache miss - read from disk, store in cache
- Cache hit - read from ram
- Caching strategies:
    -
- Caching write strategies
    -

Multiprocessing vs multithreaded vs multitask

- Multiprocessing is many processes in parallel on different cpus with different memory spaces/resources
- Multitasking is swapping between different processes on 1 cpu (context switching)
- Multithread is having multiple threads on several or one cpu within one process (shared memroy/resources)
    - Krnel threads are managed by kernel; user thread is unknown to kernel
- Multiprogramming is using i/o switching (swap to another process when other is doing i/o and no need cpu anymore)
-
- Multiprocessing is best for cpu-bound while multithreading is best for i/o bound (also shared memory/state)
    - Multiprocessing for making cpu intensive computations parallel
    - Multithreading for making any task run faster due to parallelism

Concurrency

- Process states: new, running, ready, waiting, finished
    - Ready suspended, waiting suspended if queues are full
-

Virtual memory

persistance

Misc

- Execve - system call to execute a program

    - Pass in file path, argv[] (args), envp[] (environment variables where format is key=value in string)
        - Copied from
- System v abi - a specific application binary interface, or protocols for how programs interact with OS at binary level
    - Function arguments are registers and stack
    - Uses elf and fork()
    - Defines memory management (shared libraries, stack, heap) and data structures like arrays
- Hard vs symbolic links
    - Hard links are for redundancy, so if one link is deleted, file isn't deleted until last link is deleted
        - Points to inode
            - Cannot link to directories
    - Symbolic links can be changed to point to something else, so is useful for version control (point to different versions depending on needs)
        - Is just hte path to the target file, so if target file is deleted, symbolic link is useless
            - Can link to directories
- Table used for interrupt handlers - IVT - interrupt vector handler. In newer hardware, it is called interrupt descriptor table (x86)
    - Os indirectly interacts with this when setting IRQ
    - Ivt: maps interrupt vectors (numbers) to memory addresses of ISRs (interrupt service routines, aka programs to handle the interrupt)
    - Process:
        - Ivt is populated at boot
            - Offset = INT # × Size of IVT Entry
            - Interrupt numbers and offset usually stored unless know size of IVT entry. Offset is where find interrupt handler address
        - Irqs are set and if triggered, then goes to ivt entry irq corresponds to
            - Set by: request_irq() -> ivt entry is set to point to interrupt handler requested
        - When hardware gets signal, looks up ivt entry irq corresponds to, then makes pc jump to interrupt handler (IVT offset) from ivt entry key's address value
    - Interrupt signal number is different from interrupt handler index in ivt
- Terminals
- Tty stands for teletype machines. Is physical hardware device to interact with system
- Now is just reference to terminal, which ahs become virtualized with a gui and virtual terminals
- Pseudo terminal
- Used for:

    - Ipc - interprocess communication, so that translations of output to input is facilitated by just putting it through a terminal so no need to make process to process specific communication and just reuse user input
    - Connection to remote
        - Master terminal (ptmx) handles communication and input/output with remote and slave. Connection initiated with ssh
        - Slave terminal (pts) handles user input and communicates with master terminal
    -
- Interruptable sleep
- Uninterruptaable sleep (D) - can only be woken by wake up call
    - Usually when is in system kernel call, io call (network/disk), hardware response, critical system resource waiting, slow network calls
        - If network is constantly slow, then too many processes may be in uninterruptable sleep
            - Need debugging tools like top to see if is uninterruptible and why
            - Can lead to deadlock if network depending on curr, where curr is waiting for network
            - May need timeout and retries systems, along with error handling
    - Choose this when don't want to corrupt data, or if want atomic state
- Interruptable sleep - can be worken up by wake up call/signal
    - Requires extensive coding when doing this over uninterruptible
    - Usually put to sleep like this whenn waiting for some computation, resource, etc.
    - Can be stopped with ctrl c
- Processes cannot be killed in kernel to avoid corrupting data. Any calls sent by user are taken into account once swap back to user context
    -
- Now there exists a killable state: is uninterruptable but accepts fatal signals that will interrupt the sleep state
- Ways to stop uninterruptable sleep
    - Get ppid (parent pid) and kill it (usually is shell and kills child)
    - Prevention: keep drivers up to date
    - Restart ssystem/suspend to disk
    - https://www.baeldung.com/linux/uninterruptible-process
- Containerism - running multiple servers on one machine without changing machine
- Can do so with namespaces, cgroups
- Namespaces - isolate resources
- pros
    - Gives illusion of being its own machine due to its own space separate from global (some pid-hard coded legacy programs depend on that)
        - Good for multiple webservers on a machine
    - No conflicts of resouce management

    - Ecapsulation - can't see other namespaces
-
- Can create specific spaces where a particular attribute's set space is unique to that space
    - Ex: pid namespace means that all pid's are considered unique within that space and do not match globals
        - Create pid namespace: one process is chosen as root of namespace. All its childrens are added and their pid's are made unique for that namespace, different from global
- Ex namespaces
    - Pid, starts at pid 1 (so can have own set of pids)
    - Mount (so can have own file system)
    - Network interfaces, ip, routing tables
    - Uts (hostname/domain) - container can set its own hostname without affecting whole machine
    - Ipc - no leaks/peaking at other ipc namespaces
    - User (isolates user, groupids) - that way have hierarchy between users in global and in namespace, where namespace root might just map to a global normal user type
    - cgroup
-
- cgroups
- Manage resource allocation and limits for specific containers to prevent hogging
    - Cpu time allocation
    - Ram and swap usage
    - i/o bandwidth (limit to specific rate)
    - Network bandwidth
    - Number of pids
    - Access to devices
    - Huge page total size
    -
- What isn't shared between child/parent processes using fork
- Not: name space, network interface
- What is: memory, file descriptors, signal handlers
- Cpu utilization
- Total time cpu is utilized
    - Time cpu is busy / total time * 100
    - Use top, htop, mpstat, iostat to check
- Scheduler saturation - system's scheduler is overwhelmed by needs of processes
    - Longer load times
    - High cpu utilization does not necessarily mean schedule saturation. If schedule is struggling, then that is saturation
- How to prevent
    - Cpu pinning

    - Increase number of cpu cores
    - Better load balancing
    - Use cgroups or reevaluate nice values
- Raid
- Okay, here's a concise summary of common RAID levels, their pros, cons, and purposes:
-
- *   **RAID 0 (Striping):**
- *   **Purpose:** Performance. Data is striped across disks.
- *   **Pros:** Fastest performance, full usable capacity.
- *   **Cons:** No redundancy. One drive failure loses all data.
-
- *   **RAID 1 (Mirroring):**
- *   **Purpose:** Redundancy. Data is mirrored across disks.
- *   **Pros:** Excellent data protection. Can survive multiple drive failures (depending on how many mirrored copies).
- *   **Cons:** Half the usable capacity. Write performance can be slower.
-
- *   **RAID 5 (Parity):**
- *   **Purpose:** Balance of performance and redundancy. Data and parity information are striped across disks.
- *   **Pros:** Good read performance, decent redundancy (can survive one drive failure). More efficient storage than mirroring.
- *   **Cons:** Write performance can be impacted. More complex to implement.
-
- *   **RAID 6 (Dual Parity):**
- *   **Purpose:** Enhanced redundancy. Similar to RAID 5 but with two parity disks.
- *   **Pros:** Can survive two drive failures.
- *   **Cons:** Lower usable capacity than RAID 5. Write performance can be impacted.
-
- *   **RAID 10 (Stripe of Mirrors):**
- *   **Purpose:** High performance and redundancy. Combines striping and mirroring.
- *   **Pros:** Excellent performance and redundancy.
- *   **Cons:** Half the usable capacity. Expensive.
-
- *   **RAID 01 (Mirror of Stripes):**
- *   **Purpose:** Redundancy and performance. Mirrors striped sets.
- *   **Pros:** Good performance and redundancy.
- *   **Cons:** Half the usable capacity. Expensive. Less common than RAID 10.
-
- *   **JBOD (Just a Bunch of Disks):**
- *   **Purpose:** Combine disks without RAID features.
- *   **Pros:** Uses all disk space.

- * **Cons:** No performance benefit or redundancy. A single drive failure loses its data. Not technically RAID.
-
-

rtos

Linux

Process management

- Kernel is portable
- Is multithreaded
- System call to request kernel resources (admin resources)
- process/thread are the same in linux; all are tasks. Just handled differently
- Process descriptor - describes tasks; is doubly linked list of tasks (task structs)
  - Task struct is pcb (process control block) in theoretical OS
  - Thread struct is within and holds next instruction ptr
- fork() (clone) - create new task by copying parent (makes child)
  - Done through copy on write (parent and child shares same copy until write, then diverge)
  - Vfork: same as fork but parent table not copied
- Fork process:
  - Duplicate task structure and set pointers, check not over max process limit, adjust stats in child, set child as uninterruptible, copy parent flags but make sure exec isn't marked, assign child pid, copy/share resources, cleanup and return child ptr
- Task states: running, interruptible (waiting for signal), uninteruruptible (wait without signal), stopped, traced (debugging trace by other task)
  - Running upon init is not truly running, just ready, then runs upon context_switch()
- Kernel task has kernel status and can only be created from kernel
- Process termination: sets task structure's flag to PF_EXITING, removes kernel timers, exits mm and accounting and semaphores and files, notifies parent (exit_state=EXIT_ZOMBIE, until parent notifies read (just says if is interested/not)), remove itself from schedule
- Release process descriptor: removes process pid, removes resources, notifies potential zombie leader's parent, free pages used by task struct, etc.
- REparenting: child is zombie but parent exits before and never receives message, causing child to forever be zombie. Reparenting gives child parent from sibling

Process scheduling

- Preemptive multitasking - tasks can preempt each other (interrupt temporarily, for determined time called timeslice. Global decisions)

    - Cooperative - process determines how long can use processor (no global)
- i/o bound - process spends lots of time on input/output (graphical usually)
- Process bound - process spends a lot of time on running code

Priority

- Higher > lower vruntime, same -> round robin
- Two types of priority:
    - Nice - user-adjustable suggestion: -20 to +19 (positive is less priority (nicer))
        - For normal
    - Real-time - adjustable (root), 1 to 99 (higher); have strict timeslices
        - For sensitive; > normal
- Timeslice - how long something has to run before it is preempted
    - Io bound does not need long; process boudn does
    - Timeslice is inverse function of system load, nice inverse (higher means less time)
- Linux is very fair - allocation is a concept of time, not amount of processor. 50% means 50% of time approx, so if process only uses processor when woken up by user, that means when it wakes up, it uses a lot more processing time since was slept
- Nice values aren't the best since causes either fast swapping with low priority only, long swapping with low and high priority. Nice value differences can be big if nice gap is big. Timeslices need minimum tick. Priority boost given to processes waking up to boost interactivity/instantaneousness (but is unfair)
- Cfs does not assign constants like timeslice, but proprotionality of cpu for time (cannot run multiple), swapping with round robin
    - Still use nice value
- Targeted latency - desired window of time being allotted to all (smaller means more items have processing time, but higher switch cost relatively)
    - 20 means will be split across n items
- Minimum granularity - minimum time running so switch not too costly relative to time run
- Virtual runtime - actual runtime normalized by number of runnable processes
- update_curr(): add delta_Exec (actual runtime since last update) to vruntime after dividing delta_exec by total processes (more processes = less runtime for each so better "multitasking")
- Smalles vruntime goes next for run
    - Red-black tree used to keep track of all runnable processes (self balancing BST)
        - Better to cache so can get leftmost fastest (set as lseft. If changed, set it to its successor (inorder)
- Adding processes to rBST: called after wakeup or fork, update new process stats/vruntime, insert into rBST based on vruntime (update left cache if becomes leftmost)
- Removing processes to rBST: block/terminate, remove from rBST, rebalance and update leftmost pointer if need to (leftmost pointer is minimum vRuntime)

- # Short term scheduling
- ## Context switching and waiting
- Wait queue - processes waiting for something to occur; queue cycles through and if no signal to run, then that process is sent back to end of queue
- Context switching - swapping from one process to another; must load in virtual memory mapping and restore stack and register/etc. Info
- Need_resched flag tells kernel when to call schedule()
    - It is set when preempted or higher priority awakens
- Linux kernel is preempt, so kernel code can be preempted. Only true when kernel has 0 locks
    - Can only preempt if preempt_Count is 0. Increases when acquire lock; decrease when release
- ## Spaces
- User space - normal apps run here
    - If apps need kernel space, they do system call
- Interrupt - signal that event needs attention; immediately goes from user to kernel
- Interrupt handler - handler (catch) for interruption. Runs in kernel space
- Kernel space - root access essentially
- ## Real time policies (static/soft)
- Fifo - queue and runs in order until block/yield
- Rr - run fixed time before go next
- eal-time priorities range from 0 to MAX_RT_PRIO - 1 (default is 100). The normal tasks' nice values range from 100 to 139, overlapping with the real-time priority space.
- ## Processor affinity and yield
- Tasks have processor affinity, stored in bitmap in task struct. This tells kernel what processors task can be allowed on
    - Is inherited by parent
    - Can be manually adjusted
    - Cache locality - keeping processes on same processor to ensure cache not invalidated (soft affinity)
    - Load balancing is automatic
- Task_running tasks can yield to other waiting tasks

● Kernel

- System call - do it bby making specific exception (can't system call in user space otherwise)
- Each system call is assigned a unique nonchangable number
- Api exists so that can have same interface but different implementation across various systems
- 2 ways to ensure kernel does nto do bad stuff from system call: check that user has permission or use safe copying
- User_copy_from and user_copy_to(dest, info ptr, data size) are functions to copy from src and to dest while ensuring:

    - Ptr refers to user space, is within user's space, is readable/writable/exectuable
    - This ensures, unlike memcpy, that kernel does not blindly copy data
    - Returns number of bytes that didn't copy
- Check if process has specific keyword capability: capable(keyword)
- Must log system call (for debugging)
    - Each associated with own number
    - Like read to 0
- Making own system call
    - Syscall Number: Each system call is assigned a unique number (__NR_foo) to identify it in the syscall table.
    - Macros: _syscalln macros simplify invoking system calls directly, handling register setup and the trap instruction.
    - Not worth it due to inconsistency across other architectures

● Interrupt handler

- Asynchronous vs synchronous interrupts: asynchronous is to send message (from non-related process) while synchronous is to handle error/kernel call for current
    - System calls are synchronous; hardware is often asyncrhonous
- Top half: part of asynchronous interrupt that is taken care of immediately
- Bottom half: part that is taken care of later
- Irq - interrupt request - register an interrupt function
    - Request interrupt with request_irq(irq num, function pointer to irq handler, flag, name of device interrupting, device ptr for shared interrupts) - returns 0; other if error
    - Use irq_free to do the opposite
    - There are dedicated itnerupt lines (irq num) to communicate interruptions
- Interrupt handlers are like this : irqreturn_t handler(int irq, void *dev)
- Interrupt handler flags:
    - IRQF_DISABLED: Disables all interrupts while the handler runs (used for performance-sensitive interrupts).
    - IRQF_SAMPLE_RANDOM: Contributes interrupt timings to the kernel's entropy pool for random number generation.
    - IRQF_TIMER: Indicates the handler processes system timer interrupts.
    - IRQF_SHARED: Allows multiple handlers to share the same interrupt line.
- Driver allows os to interact with hardware. Interrupt is better than polling (no constant checking)

● Bottom half - asynch

- Bottom half since top half with irqf disabled disables all other interrupts, which could slow down processes
- No longer exist: task queue, bh
    - Task queue used to be like work queue but allowed synchro between tasks
- Softriq
    - Interrupt context

    - Number of registered softriqs statically determined at compile time; cannot be changed at runtime (limit of 32; only 9 exist)
    - Can be interrupted, meaning that if the same softriq is run again while it was interrupted a second time, then concurrency issues
    - Basically scales better since has no formal lock
    - Associated with queue? of them along with bitmap where 1 represents need to be done
    - Runs in kernel
- Tasklet
    - Interrupt context
    - Cannot be interrupted; race condition safe
    - Run by kernel
    - Built on top of softriq
    - Is a linked list; each is a tasklet_Struct
        - Vars: Reference count, state, handler func ptr, data, next
- Work queue
    - Process context
        - Can sleep, allocate memory, wait for io
    - Requires threads to be assigned the delayed work. However, this allows it to be in process context and not interrupt context, which means it won't take up compute time from other processes
    - Runs in kernel thread
    - Allows for concurrency
    - Is just a queue otherwise with workers with tasks
    - Worker thread each have delayed actions assigned to it. Structure represents worker info
    - Use schedule_Work() for immediate; schedule_delayed_work() for delayed
    - Can flush (won't flush delayed). Cancel delayed with cancel_Delayed()
- Softriqs handled when load gets too heavy; otherwise doing it whenever interrupt results in starving user and doing it after init of new one only starves sofriq

● Locking

- Lock data if it is shared or would be called again by same type of process during multiprocessing
- Lock granularity - how much data is locked (course is lots; fine-graind is small)
- Deadlock means og waiting on resource that is waiting on og (four way stop everyone waiting for each other)
- Sofriqs need locks since can run simul; tasklets don't unless have global shared
- Locks are atomic so no need to worry about race condtiions within
- 32 bit Atomic integer methods act on atomic_t type
    - Atomic_t USED TO only be able to use 24 bits for integer value and then rest was lock. Now it is 32 bits
- 64 bit amotic integer methods uses atomic64_t (long counter inside) - very rare
- Atomic bitwise like set_bit, clear_bit

- Spin lock - held by at most one process. If contended, wait loop (spin) until ready (can sleep but this takes context switches since need to find new task to run after this sleeps). If not, just take the lock
    - Can be used in interrupt handler since no sleep (semaphore can't)
    - Can disable interrupts while holding lock if want to
- Writer lock - locks out readers and other writers
- Reader lock - locks out other writers
- Bkl used to exist for kernel but made kernel serializable. This is bottleneck in smp. Therefore, now just use regular locks (sequential locks usually, are a class of locks that require order of task execution)
- Better: semaphore -
-
- Allows for longer lock holding
- Can be preempted, thus not affecting latency
- Enforces mutual exclusion if binary
- Binary (aka mutex) and counting semaphores - binary is just either one has it or not. Counting shows how many hold the lock
    - down() is used to obtain counting semaphore (decrements count)
        - If new count is zero or greater, lock is acquired (positive x means x can be acquired for this lock)
            - down_interruptible() allows for interruptible whereas down doesn't
        - Release is up()
- reader/writer semaphores also exist
- Use trylock() to try to get lock
    - Regular down or lock just sleeps
- Mutexes
- Are semaphores with counts of one
- Enforces mutual exclusion
- Similar functions to semaphore but no down/up
- Recursive unlock not allowed (no interrupt then call same process type and unlock)
    - Locker must unlock
    - Process cannot exit while holding mutex
    - Only managed by api

- When to use semaphore or spin lock

Table 10.8 **What to Use: Spin Locks Versus Semaphores**

Requirement	Recommended Lock
Low overhead locking	Spin lock is preferred.
Short lock hold time	Spin lock is preferred.
Long lock hold time	Mutex is preferred.
Need to lock from interrupt context	Spin lock is required.
Need to sleep while holding lock	Mutex is required.

-
- Completion variables
- Is variable that is signal that thread x has finished so thread y knows (or threads)
- Smp - symmetric multiprocessing
- Parallel processors share memory space
- Big kernel lock is bad
- Is global spin lock for kernel
- Is recursive (no deadlock since is spin)
- Can sleep
- Can only acquire in process context
-
-
- Sequential locks
- Is a lock that dynamically determines read/write: when daata is written to, lock obtained and sequence counter is incremented. If sequence from time x to time y is different, then write occurred
- Good for lots of read, little writers
-
- Other
- Barrier/orders - sometimes compilers reorder variable setting if no dependencies
- However, rmb() prevents this such that no reordering across the barrier in which line this rmb() is in
    - Is reading barrier
- wmb() is writing barrier
-
- Disabling preemption - a preempt_disable().

Network file system - linux file sharing

- Network file system - system to allow computer to access remote files
    - SMB (server message block) is for windows, afs (Andrew file system) is focused on scalability and security
- Virtual file system - teh abstraction interface for both nfs, local (no need to know underlying file system. This is wrapper)
    - C objects: Superblock object, file object (open file object), inode, directory
    - Mount, then have superblock object (represents mount)
    - Inode object represents files, directories, fifos, etc. are either in disk for file systems or memory for memory i/o. Can look up inode by calling lookup() on parent
    - Address space object used to cache
    - Process:
        - File oriented call -> kernel activates and calls vfs
        - Maps vfs commands to target file systems' commands by handling different implementations where fat-compatible systems don't handle directories as files, then this system converts to driver code, and executes it
    - Tmpfs - file system that stores file data in ram
    - Sysfs - mounted at /sys in linux; represents system's devices and configuration
- Nfs provides mount or entryway into virtual file system
- Nfs process
    - Client and server egotiates version; lowest in common wins
    - Udp and tcp negotiation; connection-oriented (tcp) is selected first if both have it available. Otherwise, udp (non-connection)
    - File size negotiation: bigger is better; ver 3 has unlimited but goes by 32 kb bids
        - Ver 2 is 8kb
    - Client requests mount port from server portmap service, gets it and pings that service, then server gives file handler/system, then asks for nfs port number and requests for data using file handler/nfs port number
    -
-

Virtaul memory

Caching

persistance