Sources

- **Brendan Gregg - Systems Performance_ Enterprise and the Cloud (2020, Pearson)**

Os vocab to use over basic “slower”

- **Latency** (instead of "slower")
- **Throughput** (instead of "speed" or "performance")
- **Serialization overhead** (for the cost of sequentially executing operations)
- **Contention** (when multiple processors or threads compete for a shared resource)
- **Scalability constraints**
- **Fine-grained locking** (for locks that minimize waiting time)
- **Synchronization primitives** (e.g., locks, atomic operations, barriers)
- **Concurrency control mechanisms**

Practical stuff for linux

- Signals
    - Ctrl \ is sigquit, which creates a core dump (memory state upon termination for debugging)
    - Ctrl z is sigstop
        - Can then use fg to run it again in foreground
        - Or bg then to run it in backround
    - Ctrl c is sigint
- If you want to run process in background there’s 2 ways
    - Tie it to bash (child of bash)
        - Terminates when terminal does
        - Just do CMD & (or ctrl z then bg)
            - Is background
    - Or tie it to init
        - Nohup CMD &
        - CMD &, then Disown %1
        - Setsid CMD
        - Tmux then run once in persistent your cmd
        - systemd-run --user --scope your_command
        - Create systemd service (need config file); this starts auto upon boot
-

Glibc

- standard/general library for c in linux
- Provides wrappers for syscalls
    - Some add additional logic before calling syscall
- Important memory: malloc free
- Provides pthreads; aka posix threads

    - Share memory but have diff stacks + registers
- Dynamic linking - links libraries to dynamically linked executable lazily
- Signal wrappers like signal, sigaction, sigsetjmp, iglongjmp
- Fork and execve
- Open and fopen (fopen uses internal buffer while open is direct syscall)
    - `
- getaddrinfo() resolves domain names to IPs.
- Security
    - Stack Canaries (-fstack-protector)
    - 🔹 ASLR (Address Space Layout Randomization)
    - 🔹 FORTIFY_SOURCE (Buffer Overflow Protection)
    - 🔹 Secure Memory Allocator (tcache for malloc())

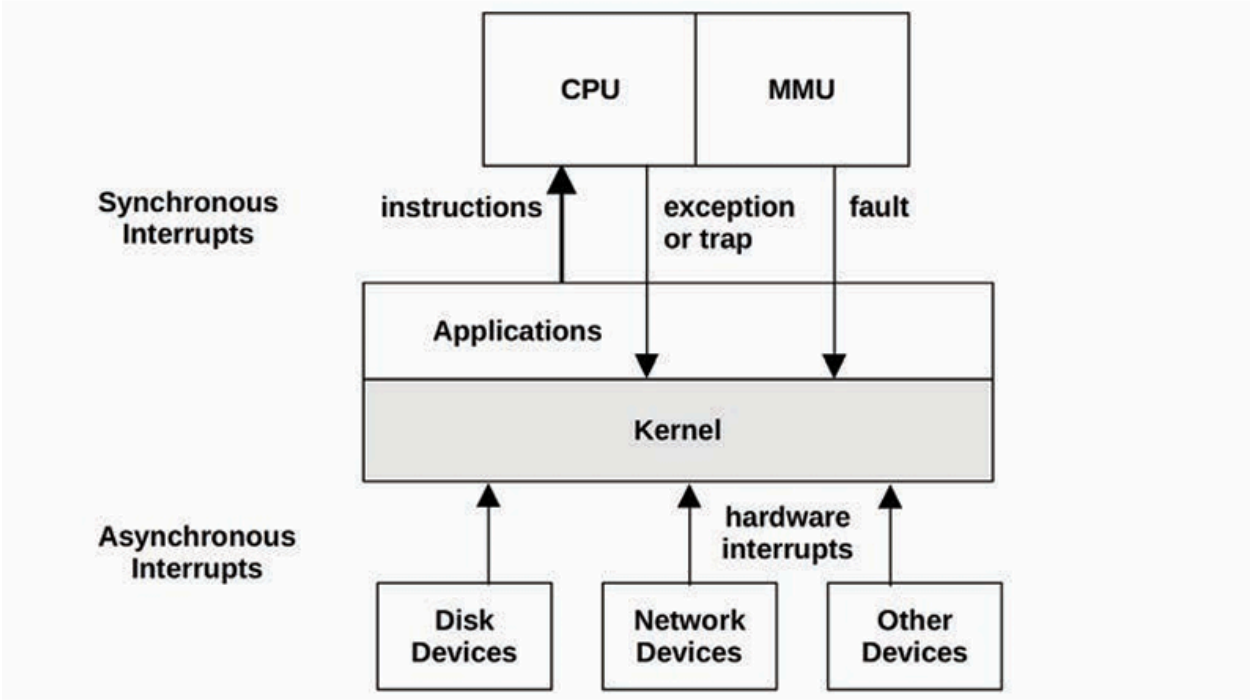
Synchro

- Lock overhead
- Is waiting time (stall time), caused by contention
- Is extra cpu used to run instructions to get/release lock
- Out of order instructions - a problem for multithreaded
- Execute other nondependent instructions while waiting for memory, like add before load finishes
- Allows for faster cpu processing due to less stalling but a problem for multithreaded
- Cpu does faster instructions first (add not load, etc.)
    - Fast ops done before stores/loads. Only if have no other dependencies
        - However cpu only knows this for current program
            - If other threads depend on this program, it is over
                - Ex: while (flag == 0); // (3) Wait until flag is set
                - print(x); // (4) Expect x == 1
                    - Flag is changed before x is, making x print old value
- Affects memory synch
- Writes may not be immediately visible to other cores
- Fix:
    - Use atomic in C to prevent reordering; is a type of fence/memory barrier
    - std::atomic<int> x(0);
    - std::atomic<int> flag(0);
    -
    - void thread1() {
    - x.store(1, std::memory_order_relaxed);
    - flag.store(1, std::memory_order_release); // Prevents reordering
    - }
    -
    - void thread2() {
    - while (!flag.load(std::memory_order_acquire));
    - int value = x.load(std::memory_order_relaxed);

    - std::cout << value << std::endl; // Always prints 1
    - }
    -
    - Setting cpu serialization instr like CPUID (x86) forces execution order but is costly.
- Actual barrier
    - #include <stdatomic.h>
    - int x = 0, y = 0;
    -
    - void thread1() {
    - x = 1;
    - __sync_synchronize(); // Prevents reordering
    - y = 1;
    - }
    -
    - void thread2() {
    - if (y == 1) {
    - printf("%d\n", x); // Ensures x is 1 if y is 1
    - }
    - }
    -
    -

- System

nerated by software instructions. These are pictured in Figure 3.4.

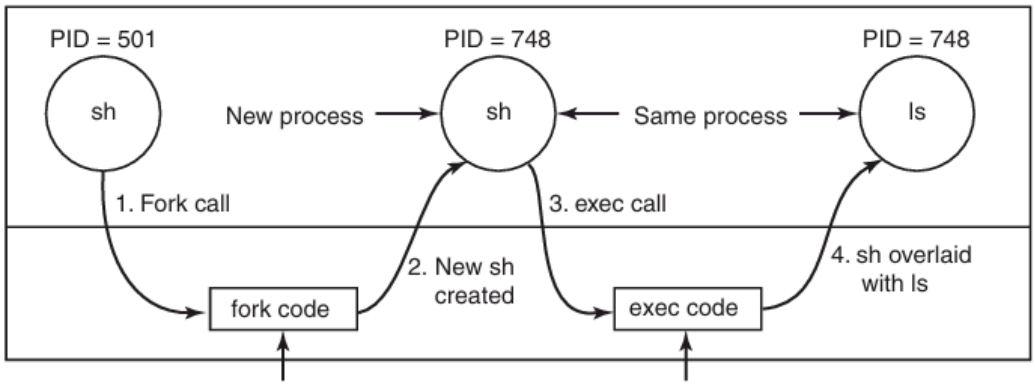


- important of these differences are
  - The kernel has access to neither the C library nor the standard C headers.
  - The kernel is coded in GNU C.
  - The kernel lacks the memory protection afforded to user-space.
  - The kernel cannot easily execute floating-point operations.
  - The kernel has a small per-process fixed-size stack.
  - Because the kernel has asynchronous interrupts, is preemptive, and supports SMP, synchronization and concurrency are major concerns within the kernel.
  - Portability is important.
-
- Kernel memory is not pagable; no one to enforce memory access
- 4 kb - 8kb stack on x86
- Spinlock, semaphores used for concurrency
-
- Common signals
- Process related
- SIGKILL (9): Immediately terminates a process (cannot be ignored or handled).
- SIGTERM (15): Requests process termination (can be caught and handled).
- SIGINT (2): Sent when pressing CTRL+C in the terminal.

- SIGHUP (1): Sent when a terminal is closed.
- SIGSTOP (19) / SIGTSTP (20): Stops a process (SIGSTOP is uncatchable, SIGTSTP is sent by CTRL+Z and can be caught).
- SIGCONT (18): Resumes a stopped process.
- SIGALRM (14): Sent when an alarm() expires.
- Memory related
- Sigsegv - segmentation fault
    - Process accesses memory that it shouldn't (invalid memory address), like dereferencing null pointer
        - Usually accessing memory that's not allocated (protection invalidation)
    - Kernel generates it
    - Process usually terinates. Sometimes caught for logging/cleanup
- Sigbus - bus error
    - Process accesses memory incorrectly, like misaligned memory, or accessing unavailable physical memory
        - Usually impossible physical memory access
    - Kernel creates it
    - Process usually terminated. Sometimes caught
- oom/killer - not a signal technically
    - Kernel kills process due to running out of memory
        - Gives sigkill
- Sigpwr - power failure
    - Power loss, hardware power loss, low battery/failure
    - Generated by kernel/hardware
    - Process handles this via graceful shutdown (saving, logging)
- Execution instruction related
- Sigfpe - floating point exception; illegal math operation like /0
    - Memory related floating point operation error
    - Generated by hardware/kernel
    - Handle: process can either fix it or terminate
- Sigill - illegal instruction; process tries to execute illegal machine-level instruction, causing corrupted memory
    - Memory corruption, bug
    - Generated by cpu running instructions
    - Termination but process could handle it
- Sigtrap - trace/breakpoint trap; hit breakpoint
    - Happens when hit breakpoint
    - Thrown by program with breakpoint
    - Usually debugger handles this. Without debugger, just quit
- Cpu
- Sigxcpu - process exceeds cpu limit
    - Kernel generates this when process goes over its limit for cpu
    - Process can catch this and adjust behavior or else terminated
-

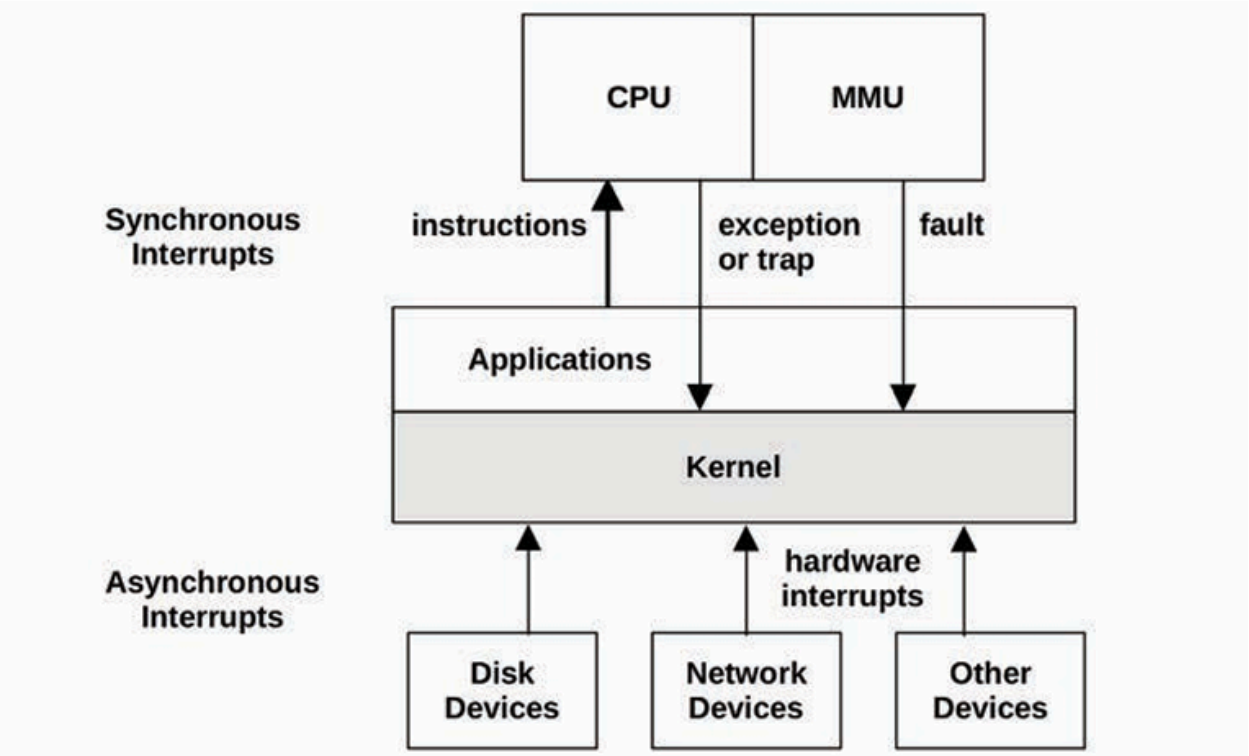
- Common syscalls
- read()
- write()
- mmap()
- clone()
    - Used to create threads
- pthread_create()
    - Used to create threads; threads use task_struct too but share memory instead of having own pid, memory space, etc.
- fork()
    -
- Exit
    - Param: int, gives exit code int
- exec()
    - Pathname (c string), argv (vector of arguments for program), envp (array of c strings null terminated)
- Handling related
    - sigaction(int signum, struct sigaction *act, struct sigaction *oldact): Defines a signal handler (act->sa_handler or act->sa_sigaction).
    - kill(pid_t pid, int sig): Sends a signal to a process (pid > 0: specific process, -1: all processes, 0: same group).
    - raise(int sig): A process sends a signal to itself.
    - alarm(unsigned int seconds): Sends SIGALRM after seconds (overwrites previous alarm() calls).
    - pause(): Suspends a process until it receives a signal.
- Network
- sendmsg()
    - Used to send data
    - Parameters: msg_flags, iov (iovec array; user buffer where data should be sent)
    - Msg_zerocopy is a flag that can be set so user packets copied directly into nic buffer instead of into kernel buffer, then into nic buffer
        - Disables some kernel checks + ruins compatibility for some systems
            - Requires additional memroy mapping to share nic buffer with userspace
- Context switching
- Cpu migration
- Lock obtained for task so nothing else modifies it
-
- cons
- With new process:
- Tlb would have to be replaced if doesn't have what need (tlbs are cpu specific)
- Numa penalty: memory locality would change, if memory is further away from new cpu, then higher latency

- Move from one cpu queue to another
- Can cause contention on shared locks (moving to another cpu, and other task replaces you and needs your same lock)

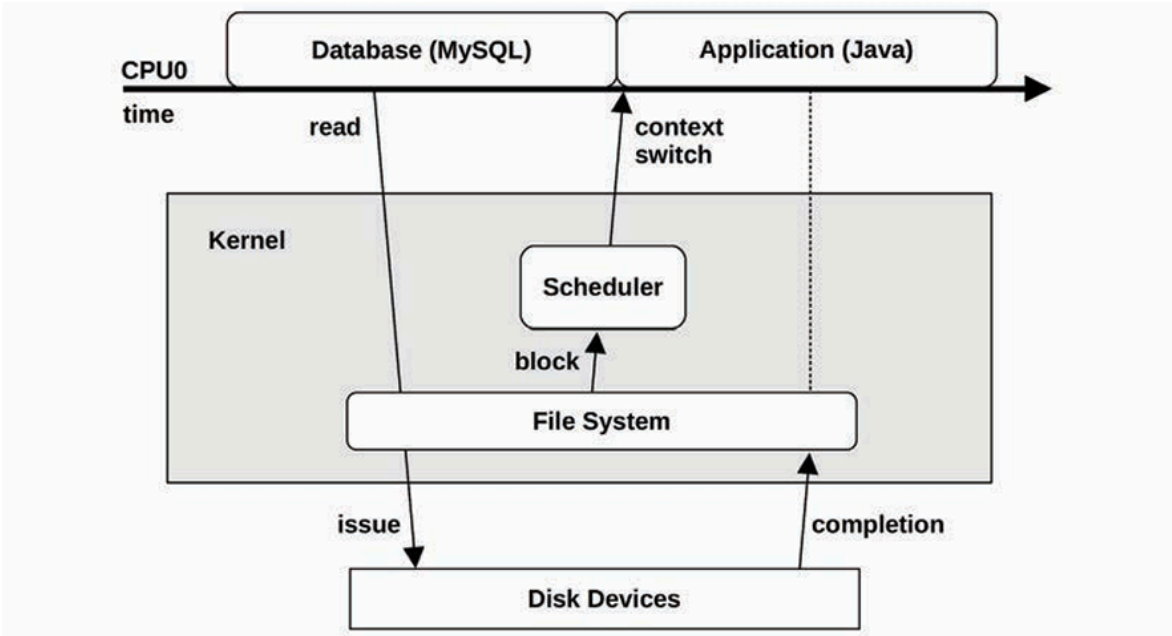


-
- Task struct
- Process priority, recent cpu usage, time spent sleeping
- pointers to text, data, and stack segments.
- Swap info
- Masks for signals to show which are blocked, caught, in progress, ignored
- Register save states for switches
- Current system call state, param, results
- Cpu user and system time
- Fixed kernel stack
- Current state of task, events waiting for

- Syscall vs interrupt, isr vs interrupt handler, ivt

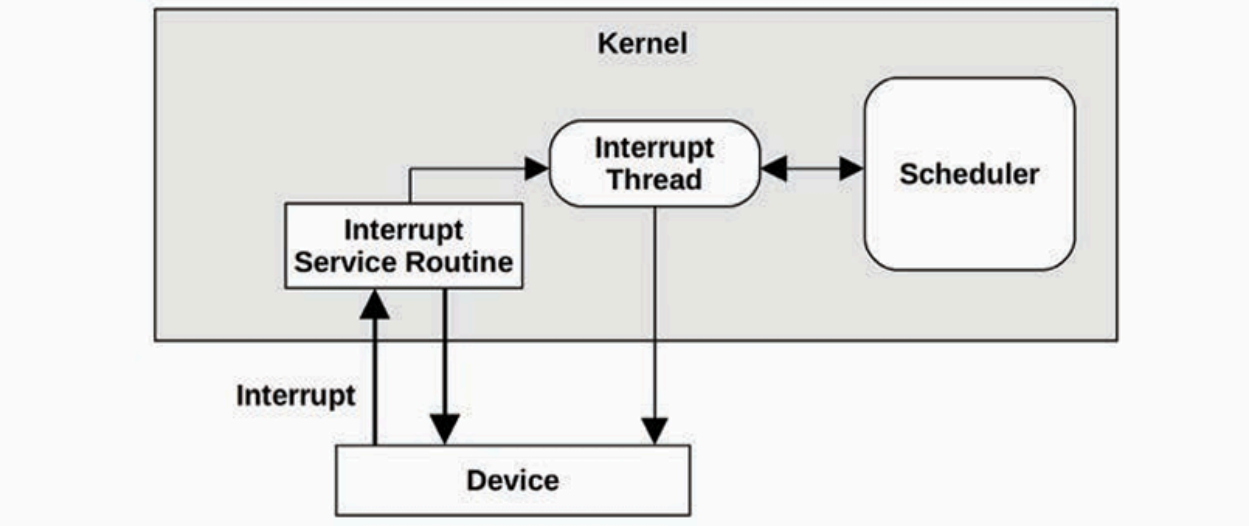


- gure 3.4 Interrupt types



-
- Asynch interrupts
    - Network packets arrive

    - Completed io
    - Hardware Input, hardware failure
- Synchronous interrupts
    - Traps - deliberate call to syscall; calling the int instruction with a vector for a syscall handler
    - Exceptions
    - Faults, like memory fault



-
- Inlinux, network drivers have irq to handle top half to handle packets, then softirq for bottom half
- , the kernel can temporarily mask interrupts by setting the CPU's interrupt mask register.
- Some high priority events can be masked as non maskable interrupts (NMI)
- Latency of 10 ms for 100 hz; kernel at 60, 1000, 100 times per second (hz). Clock can interrupt powered off idle items, consuming extra power
- Idle thread - what is scheduled when ther eis no work for cpu. Usually just check for any new work in loop. In linux, can call hlt to power down cpu
- Init is from /sbin/init
- Process life cycle

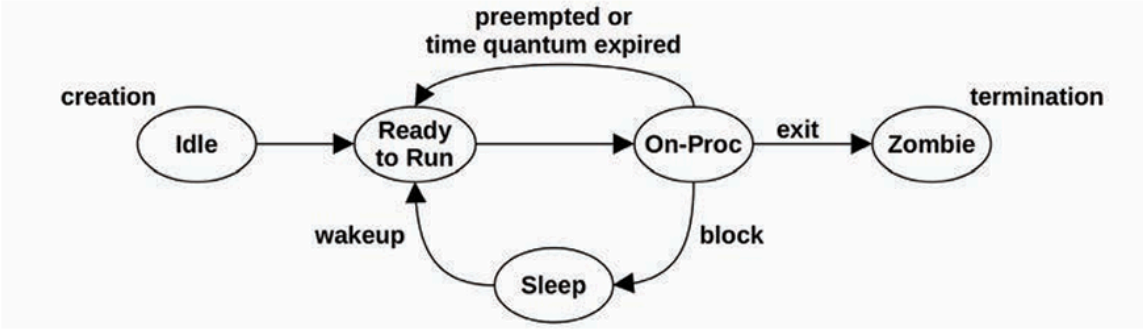
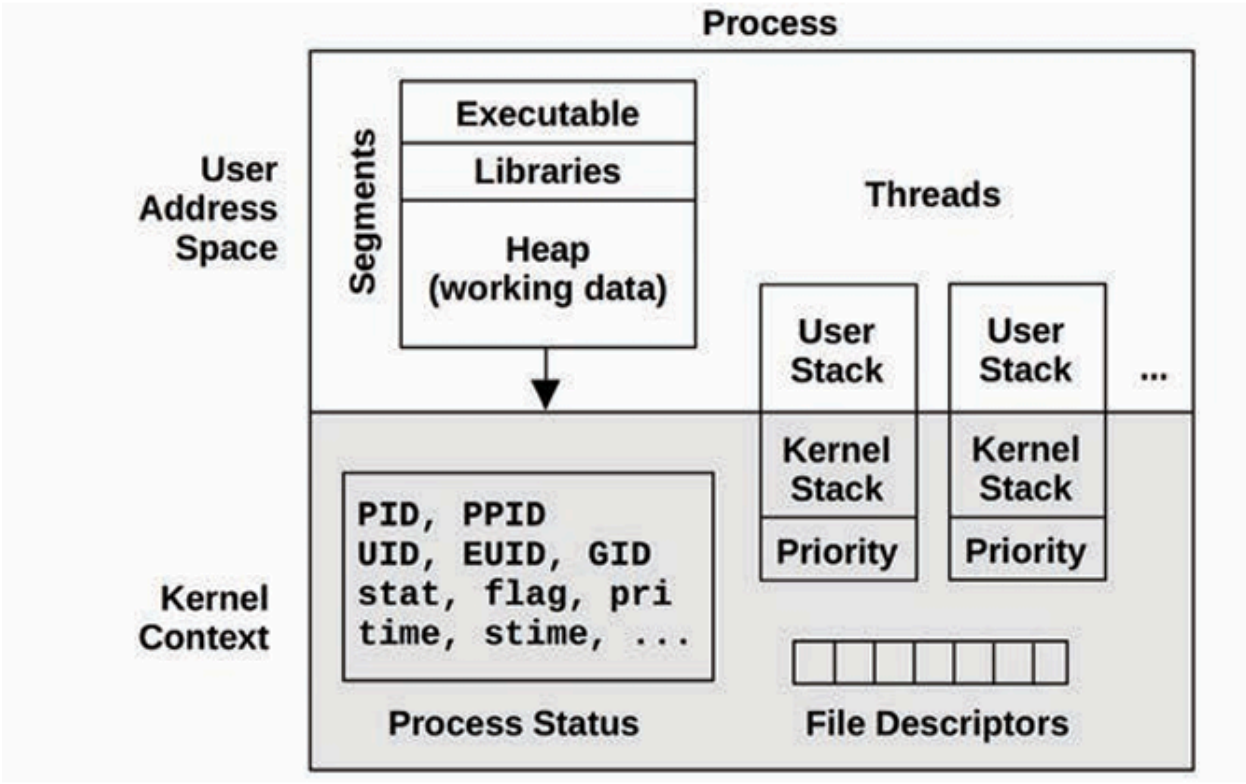
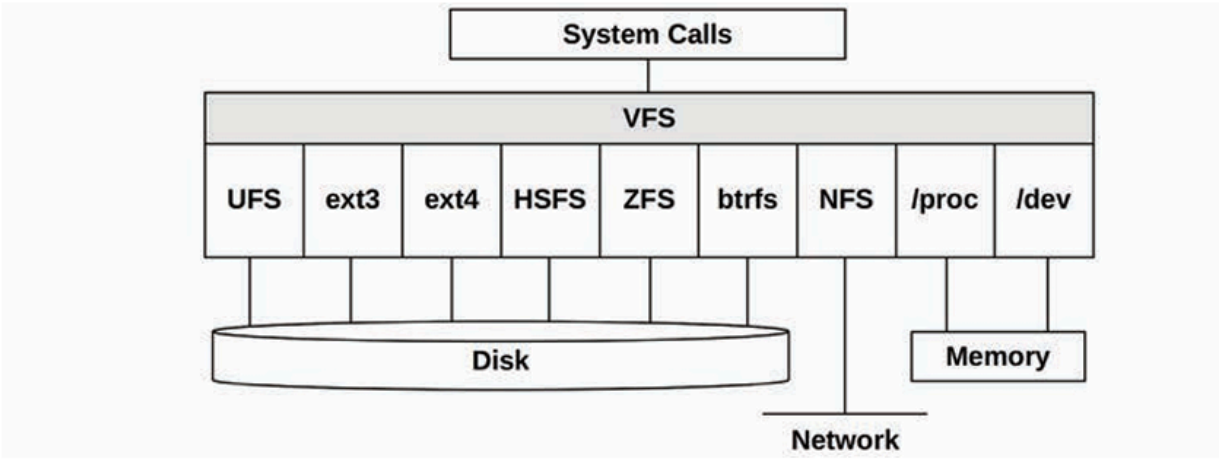


Figure 3.8 Process life cycle



- Process swapping

could more easily coexist. Its role is pictured in Figure 3.14.



-
- syscalls
    - Can block or sleep (slower)
    - Software interrupt; run by kernel but initiated by user
    - Process calls syscall (preempting itself)
    - Uses kernel stack
- Interrupt
    - Must be fast
    - Hardware interrupt; run by kernel bu tinitiated by hardware
    - Can preempt processes
    - Uses interrupt stack
- Isr
    - Short, handles time critical before deferring to bottom half
- Interrupt request handler
    - Is basically isr but also bottom half (Broad term)
- Ivt is the table that maps interrupt numbers to ISRs
- Wait vs run queue
- Wait queue - queue for io; this way, processes don't sit in run queue, taking up space/schedule time when not waiting for cpu. Put to sleep to free cpu in this way
    - Put them to sleep so that way cpu scheduler does not have to check if they are done waiting for io (no polling. This is an implicit switch). Once awake, cpu considers them again automatically (less for cpu to consider; no additional scheduler needed)
- Run queue - queue for cpu; processes in this are in the task_running state
    - Each cpu has one in modern systems; load balancer helps balance

- Caches

Table 3.2 **Example cache layers for disk I/O**

	Cache	Examples
1	Client cache	Web browser cache
2	Application cache	—
3	Web server cache	Apache cache
4	Caching server	memcached
5	Database cache	MySQL buffer cache
6	Directory cache	dcache
7	File metadata cache	inode cache
8	Operating system buffer cache	Buffer cache
9	File system primary cache	Page cache, ZFS ARC
10	File system secondary cache	ZFS L2ARC
11	Device cache	ZFS vdev

3.2

	Cache	Examples
12	Block cache	Buffer cache
13	Disk controller cache	RAID card cache
14	Storage array cache	—
15	On-disk cache	—

-
- Ipi
  - **Purpose:**
    In a multi-core (SMP) system, IPIs are used for communication between

CPUs. For example, one CPU can send an IPI to force another CPU to update its state or flush its caches.
        - **Usage:**
          IPIs are crucial for coordinating tasks such as scheduling, TLB shootdowns (updating page tables across CPUs), or for debugging and performance monitoring.
    - **BPF (Berkeley Packet Filter) / eBPF (extended BPF):**

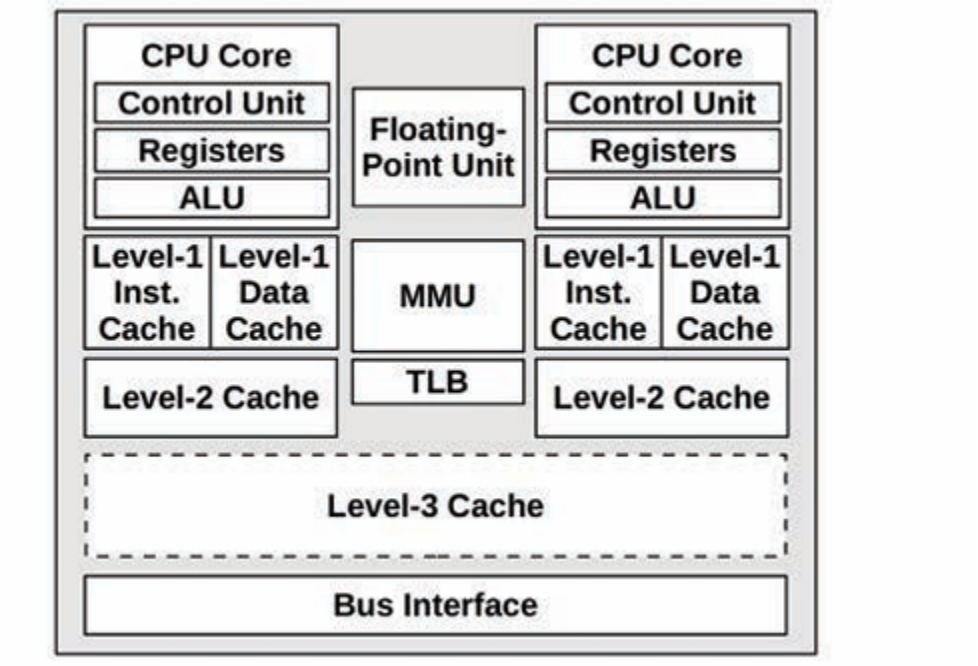
        - **Original BPF:**
          Initially developed as a mechanism to efficiently filter network packets in kernel space.
        - **Extended BPF (eBPF):**
          Now a powerful in-kernel virtual machine that allows you to run custom code safely in the kernel. eBPF is used for network packet processing, tracing, security monitoring, and performance analysis.
        - **Usage:**
          Tools like `tcpdump` and modern tracing tools use eBPF to collect data without modifying kernel code.
    -
    - Micro vs monolithic
        - Micro has minimum software: memory, thread, ipc
            - User can choose file system, network, drivers
            - Additional ipc needed for io though
        - Monolithic is everything together
- System calls
- Creating one
- Should be clear/have purpose, have checks, have as little args as possible, can have flags, copy_from_user() and copy_to_user() used
- Syscalls found in arch/<arch>/kernel/entry.S. add yours tehre
- Locate the header file for system calls, which is typically in include/asm/unistd.h. Add #define for ur syscall num there
- Add the function in kernel/sys.c or a
- Irq
- Interrupt request - signal sent from hardware and caught by kernel
- Interrupt service routine run after proper interrupt handler matched to irq in ivt
- Maskable interrupt - can be turned off
    - Nonmaskable - cannot
- Irq handler - irq number and device number passed as params (to differentiate calls)
    - Interrupt handler sends signal to processor and then jump to predefined point in kernel to handle irq
- When interrupt hapopens, it is masked so that way can't run again
- Caching
- 2-list strategy (employed by linux, similar to lru)

- Have cold list and hot list
- Cold list has less least active items from hot list
- Hot list has recently used items
    - Balanced so have equal num, moving back/forth
- Cold list is evicted from only
    - Could be extended to n lists
- Overcomes only used once pages that lru struggles with`

Cpu

- Each cpu processor contains cores that can do work
    - Each core has hardware threads, which is hardware that allows multiple software threads to run on a core
- Wide
    - If cpu is 3-wide, means can process 3 instructions per cycle
    - Is for cpus that support superscalar execution (parallel execution) in itself
- Cisc vs risc
    - Cisc is variable length (1 to 15 bytes)
    - Risc is shorter and fixed (32 bytes usually)
- Run queue for running on cpu
    - Os just sees each as a cpu; no cores or threads
- Cpus have clock rate - this is what is advertised for cpus
    - Faster this is, generally faster programs run, but not always
    - It also depends on what the cpu is doing
    - Instructions can take x clock cycles to run
        - Faster the clock rate faster this executes
- Instructions include
    - Instruction fetch, instruction decode, execute, memory access, register writeback
        - These are done in pipeline to be in parallel
    - Each stage takes at least one cycle to execute
        - Stall cycle - cycles of doing nothing to avoid ????
    - Optimizations
        - Branch prediction - when doing jumping to another instruction (like for loop or conditional), predict where might go and start executing it. If didn't go there discard (hurts performance). Otherwise it is parallelism
- High utilization is not bad as long as you get results
- Instructions per cycle - how many instructions are run per computer cycle
    - Slow memory - have more dram
    - Having faster clock rate is not necessarily better
        -
- Saturation - having to wait for cpu;
    - Preemption helps higher priority
    - Priority inversion - when lower priority prevents higher priority from running due to holding resources higher priority needs
        - Aka balckmailing

    - Priority inheritance helps
        - Basically if we have thread A holding resource, and thread C wakes up from io and is blocked, when b is run due to being next unblocked thread to run, given thread B exists and > A, and c > b, then c preempts A
        - Basically C is blocked. B is run next due to next highest priority. If B has higher priority than whatever lock process is blocking C, preempt the lock.
- Multithread - one cpu per thread, many threads in a process
- Multiprocess - one cpu per process, many processes for a program
- Thread vs process
    - Thread does not need fork, exec, clone. Just uses thread api
        - Needs less memory than process due to shared memory. Processes use ipc (reqwuires context switches, more cpu)
        - Threads may have cpu contention and memory fragmentation
        - Threads crash easily unlike processes
- multithreading/process require one thread/process per cpu (for subset of cpus, or perhaps all)



-
    - Multiple levels so can have correct size and speed of access
    - Level 1 takes a few cpu clock cycles, level 2 is around a dozen. Level 3 is ?
    - Main memory is 60 ns (240 cycles for 4 GHz processor)
        - Mmu adds latency too
- Associativity of caches
    - Fully associative - use cache eviction policies; can store anywhere
    - Direct mapped - hashed values get placed in particular buckets
        - If have same hashed value, cache performs poorly due to a lot of conflict misses (remove something you need)

        - Compulsory miss is cold miss; just miss due to not loading yet
        - Then cache miss is general just a miss
      - No priority
      - Simpler and faster
    - Set - direct mapping to bucket where fully associate cache eviction is performed
      - Can have different policies per bucket
- Data cache/inst cache are separate in level 1, but combined in level 2
  - Others may include
    - performance-related components that may be present include:
      - **P-cache**: Prefetch cache (per CPU core)
      - **W-cache**: Write cache (per CPU core)
      - **Clock**: Signal generator for the CPU clock (or provided externally)
      - **Timestamp counter**: For high-resolution time, incremented by the clock
      - **Microcode ROM**: Quickly converts instructions to circuit signals
      - **Temperature sensors**: For thermal monitoring
      - **Network interfaces**: If present on-chip (for high performance)
- P-state - states that describe performance of cpu; c-state - statest that describe idle of cpu

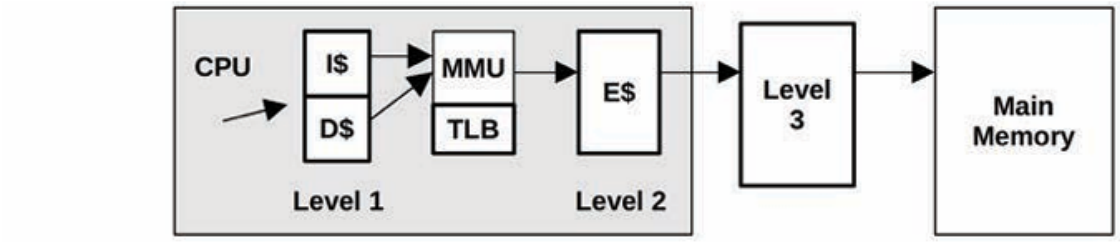
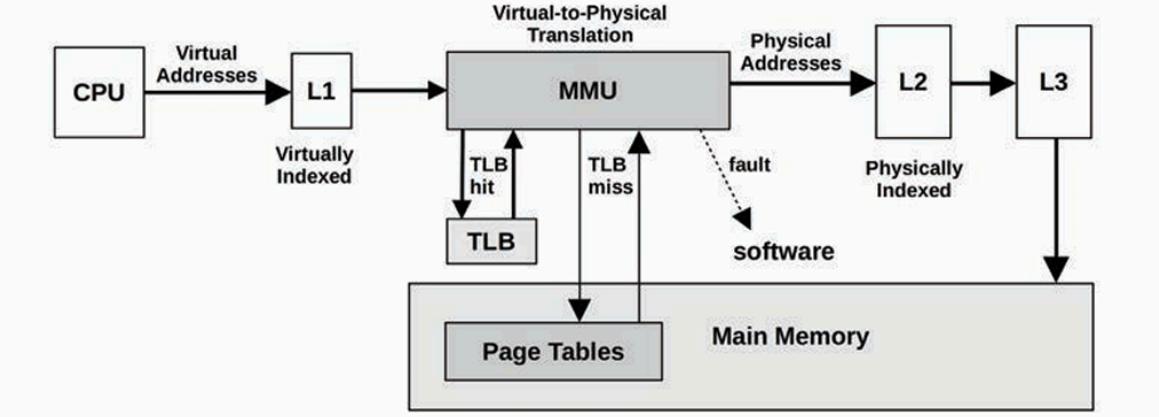


Figure 6.6 CPU cache hierarchy

-
- Pmcs
  - Count hardware stats, like cache misses, hits, cpu stall cycles (and types), resource/memory io reads/writes/stalls
- Column address strobe latency - time it takes to send column request and data returned for reading
- Uniform memory access - system bus
  - Cpus have access to all ram. With singular system bus
- Non uniform memory access - direct + interconnect bus
  - Cpus need to access ram by jumping thru each other
    - Like cpu – dram 1 – cpu 2 – dram 2
      - 2 hops for cpu 1 to get to dram 2
        - Memory (direct) bus is what connects dram + cpu. Cpu interconnect connects 2 cpus
- Cpus have 3 caches:
  - Lvl 1 is fastest but smallest. There is separation: data, instructions. This is so they don't take up more than fair space (contention; not muc to begin with), and different access patterns: instr - sequential and data - random. Also allows for fetching data + instr at same time

    - Level 2 has instr + data
    - Lvl 3 is slowest but biggest



-
-
-
- Pipeline
- ` for instructions
- Yielding
- Voluntarily give up cpu usage when time slice not expired
    - For:
        - Io block
        - Explicit yield call
        - Waiting due to pthread cond_wait
        - Sleeping
        - Priority based schedulers
        - Kernel preemption

- Scheduling

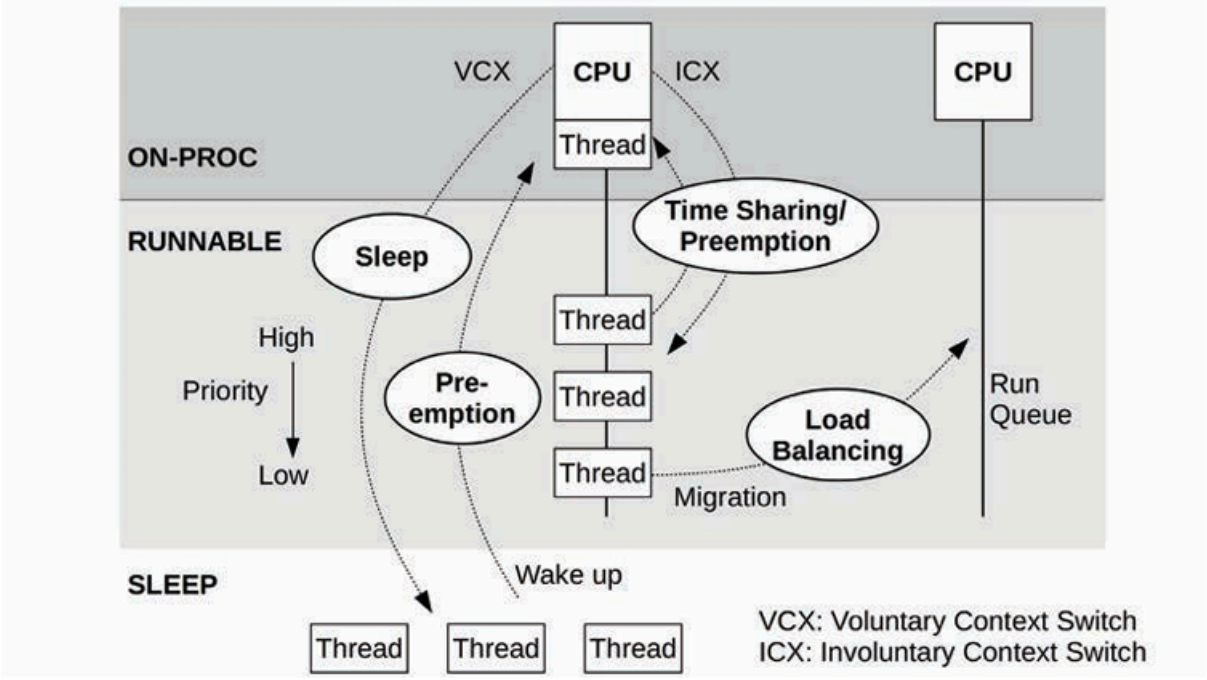
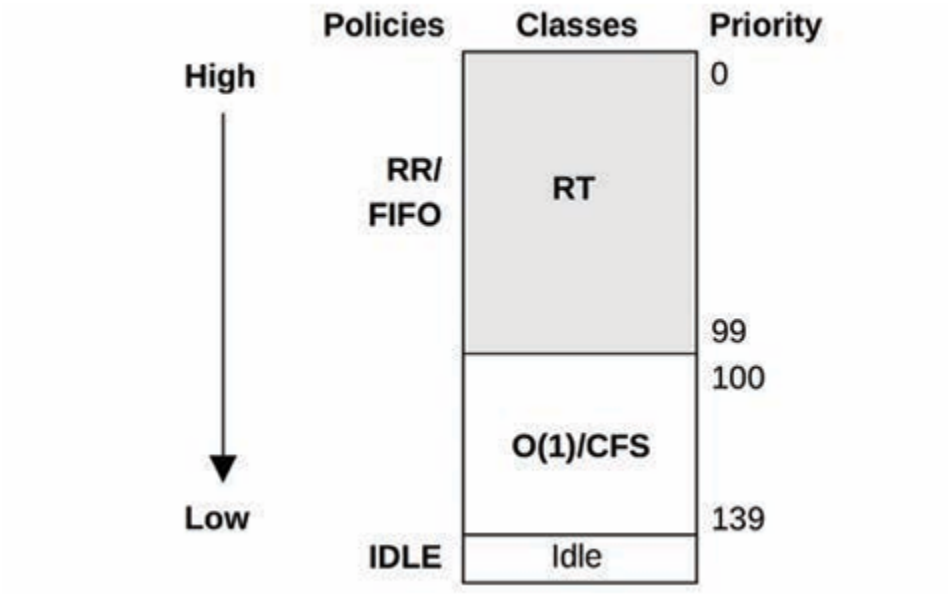


Figure 6.12 Kernel CPU scheduler functions

-
- Load balancing between cpus, preemption, time sharing



-
- Idle thread - runs on cpuw hen there’s nothing to run (usually tells cpu to use lower power/halt)
- Types of linux schedulers

    - Round robin. Uses RT.
        - Not Preemptive at same priority level (outside of timeslice)
        - Rt: priority preempts
    - First come first serve (FIFO). Uses RT.
        - Not Preemptive at same priority level
        - Rt: priority preempts
    - Normal - o(1) or cfs is used. Default time sharer for user processes
    - Batch - uses cfs. Usually used for cpu bound. Not used to interrupt other io bound interafctive work
    - Idle
    - deadline
- Gpus

Table 6.6 **CPUs versus GPUs**

Attribute	CPU	GPU
Package	A processor package plugs into a socket on the system board, connected directly to the system bus or CPU interconnect.	A GPU is typically provided as an expansion card and connected via an expansion bus (e.g., PCIe). They may also be embedded on a system board or in a processor package (on-chip).
Package scalability	Multi-socket configurations, connected via a CPU interconnect (e.g., Intel UPI).	Multi-GPU configurations are possible, connected via a GPU-to-GPU interconnect (e.g., NVIDIA's NVLink).
Cores	A typical processor of today contains 2 to 64 cores.	A GPU may have a similar number of streaming multiprocessors (SMs).
Threads	A typical core may execute two hardware threads (or more, depending on the processor).	An SM may contain dozens or hundreds of streaming processors (SPs). Each SP can only execute one thread.
Caches	Each core has L2 and L2 caches, and may share an L3 cache.	Each SM has a cache, and may share an L2 cache between them.
Clock	High (e.g., 3.4 GHz).	Relatively lower (e.g., 1.0 GHz).

-

memory

Virtual memory

- Virtual memory area - segments of virtual memory allocated to process
    - Code, heap, global vars, runtime vars, stack, mapped files
    - Tracked with vm_area_struct
        - Has start/end address, flags (rwx), backing file (if mapped from file), metadata
- The do_mmap() - map anon memory to process's address space. it can create new VMAs or merge adjacent ones if they have the same permissions.

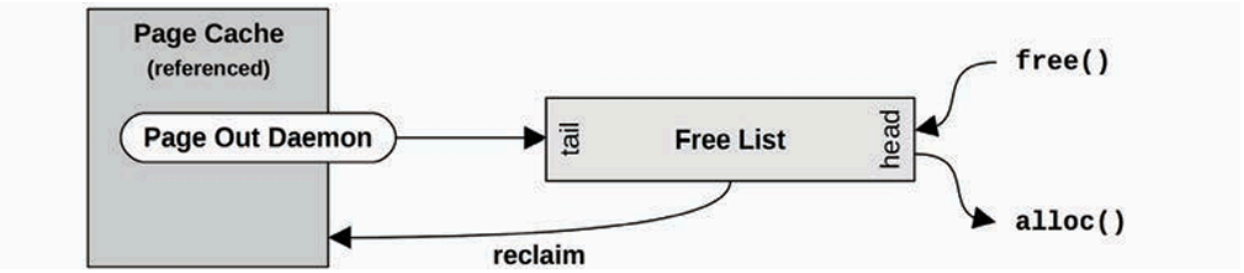
- The function parameters include file (the file to map), addr (the start address), len (length of mapping), prot (permissions), flag (mapping type), and offset (file offset).
    - do_munmap() unmaps
- Each page table stored in mm_struct
- Mmap
- Used to load libraries, memory mapped io, allocating large memory regions without fragmentation
  Do_mmap actually handles it (helper func)
    - Mmap just validates it
- Page table levels - usually 4 levels, depends on architecture. X86_64 uses 4 levles
- . What Are PGD, PMD, and PTE?
- These are parts of the page table hierarchy, and yes, it depends on the architecture.
- Many modern architectures (e.g., x86_64) use a multi-level page table hierarchy to manage virtual memory.
- Levels of page tables (for x86_64, 4-level paging):
- PGD (Page Global Directory) – Top-level, points to PMD entries.
- PUD (Page Upper Directory) – Used in 4-level paging (skipped in 3-level paging).
- PMD (Page Middle Directory) – Points to PTE entries.
- PTE (Page Table Entry) – Maps virtual addresses to physical addresses.
- Shared memory mapping and cow
- If processes share same memory mapping, page tables can reference same physical memory to save space
- Cow - share same memory pages upon fork until write, then make a copy
-
- Buffer vs page cache vs block buffer
- Page cache is cache for pages from files so don't have to read from disk upon loading into memory
- Block buffer - cache for raw block io, aka block cache. Is part of page cache now
- Disk Io buffer - temporary holding spot due to consumer/producer varying speeds of operation
- Other buffers
- Journal buffers
- Driver buffers, like network circular buffer
- User-space Buffers:
-
    - Implemented by libc or applications (like stdio's buffering)
    - Examples include fully buffered, line buffered, or unbuffered I/O
    - Controlled through functions like setvbuf()
- Socket Buffers:
-
    - Used for network communications
    - Include separate send and receive buffers

    - Managed by the sk_buff (socket buffer) data structure
    - Configurable through socket options and sysctl parameters
- ipe Buffers:
-
    - Used for inter-process communication through pipes
    - Implemented as circular buffers in kernel memory
    - Size is typically limited (often 64KB by default)

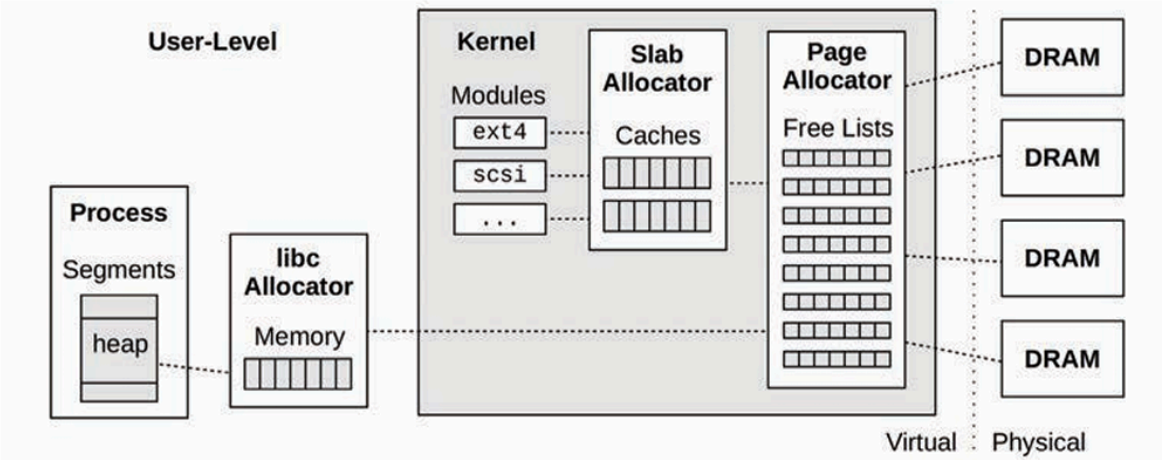
TTY Buffers:

- Line discipline buffers for terminal I/O
- Handle input/output between user applications and terminal devices
-

- misc
- Free lists



    -
    - Memory freed is added to queue to use for future
    - Consumed by allocators
- Memory segments are just sections of addresses where the heap, stack, etc. go in virtual memory
    - Free in heap does not deallocate truly; it leaves it in for future allocations for that process (re-exec). mmap/munmap wil actually return memory



-
- Dma
- Bypass cpu to writ eto ram directly
- Zones
- Memory has zones
    - Dma, normal, movable (like file caches), high address ranges (to include kernel)
        - Dma for legacy dma of network interfaces, graphic cards, etc.
        - Normal is for most apps and kernels
            - Inside each, data can be unmovable (for critical kernel structs), movable (to reduce fragmentation), reclaimable
- Non uniform memory access (numa) - memory zones split based on physical closeness to processors. Kernel allocates memory to node related to cpu. numa node (cpu group and local memory)
- Memory cleanup
- Memory defragmentation - kernel groups free memory into large contiguous blocks
    - Done in compaction memory thread (proc/sys/vm/compact_memory)
    - Moves pages around with page migration
        - Can move to make larger free blocks, or to support better numa balancing (improve memory locality for specific node by moving useful pages there)
        - How it’s done
            - Page is locked (marked as migration, but can read)
            - New physical page allocated
            - Copied done (others can still access it)
            - Now virtual address points there
            - Old page freed
- Memory reclaim (most preferred to least)
- Reaping - slab reclaim
        - Reclaim unused kernel objects in slab caches

    - File-backed page not needed, dropped (clean dropped)
        - No longer held by process/is written back already (clean), or was never written back
    - Kswapd - More needed: dirty/needed pages are moved to swap space, increasing disk io
        - Since is background task, wakes up at thresholds: High threshold: no action, low: runs in background to gradually free memory, min: foreground + aggressively frees memory
        - Can turn on direct reclaim to just run in foreground when memory is critically low
    - Forces dirty pages to be written back (even more disk io)
    - Oom killer - kills lower priority + high memory usage + higher oom score (based on past two items) processes first
- GENERAL LIST

slab Reclaim (Kernel Object Reuse)

- The **slab allocator** reclaims unused kernel objects.
- It periodically frees unused slabs when under memory pressure.

Page Cache Cleanup

- The kernel reclaims file-backed pages that are no longer in use.

Process Exit Cleanup

- When a process terminates, its memory pages, file descriptors, and other resources are automatically freed.

Periodic Memory Reclaim (kswapd)

- A background kernel thread (`kswapd`) monitors memory usage.
- It proactively frees memory when pressure is high.

Reference Counting

- Used for objects like **inodes, dentries, and file handles**.
- When the reference count drops to zero, the object is freed.
-
- kernel
- Stack
- Fixed sm all size
- Is per process
- Either 1-2 pages
    - 1 - 8kb, 16 kb for 32 bit and 64 bit respectively
    - 2 - 4, 8

- Moved to single page
    - To prevent having to look for 2 contiguous free pages due to fragmentation
        - Was supposed to prevent fragmentation due to being in 2, but with others causing fragmentation, this was hard to look for
    - Makes processes consume less memory (1 less page)
- If overflow, can corrupt kernel structures, like thread_info structure, which is at end of each process's kernel stack

High memory

- Refers to memory kernel can't access since it's u sed for userspace
    - Must be allocated manually therefore
    - Kernel usually handles low memory addresses
        - Below 4 gb range
- n certain architectures (like x86), physical memory beyond a certain threshold (e.g., 896MB on x86) is considered high memory
    - Even though can address up to 4 gb
    - For high memory, need to map with kmap() to make permanent mapping for high memory page in kernel's address space]
    - kunmap() to undo
- Atomic mapping - temporary mapping for high address due to not being able to sleep (no interrupts)
    - kmap_atmoic()
        - Temporary since it is not interruptible. Once released, is interruptible. Good for critical tasks

Slab allocator - global

- Contains caches for common fixed kernel data structs
    - Inode, task_struct
    - Each cache consists of slabs, or contiguous block of memory (usually a page)
        - This prevents external fragmentation
        - Can also be processor specific and numa-aware, preventing locking overhead
        - Slabs created dynamically
    - Slab has descriptor that gives state
        - Slab has states: slabs_partial, slabs_empty, slabs_full
            - Partial then empty slabs are used first for new objects. When freed, returned to slab
        - kmem_bufctl_t structure is first free slab
- slabs are size specific bins. full, partial, empty so can fill smallest + lessen frag
- each cache contains multiple slabs
- each has its own free list
- Page allocator serves slab allocator
- Reduces lock overhead by:
    - Having cpu specific caches
- Newer version: slub allocator
    - Reduced metadata overhead compared to SLAB

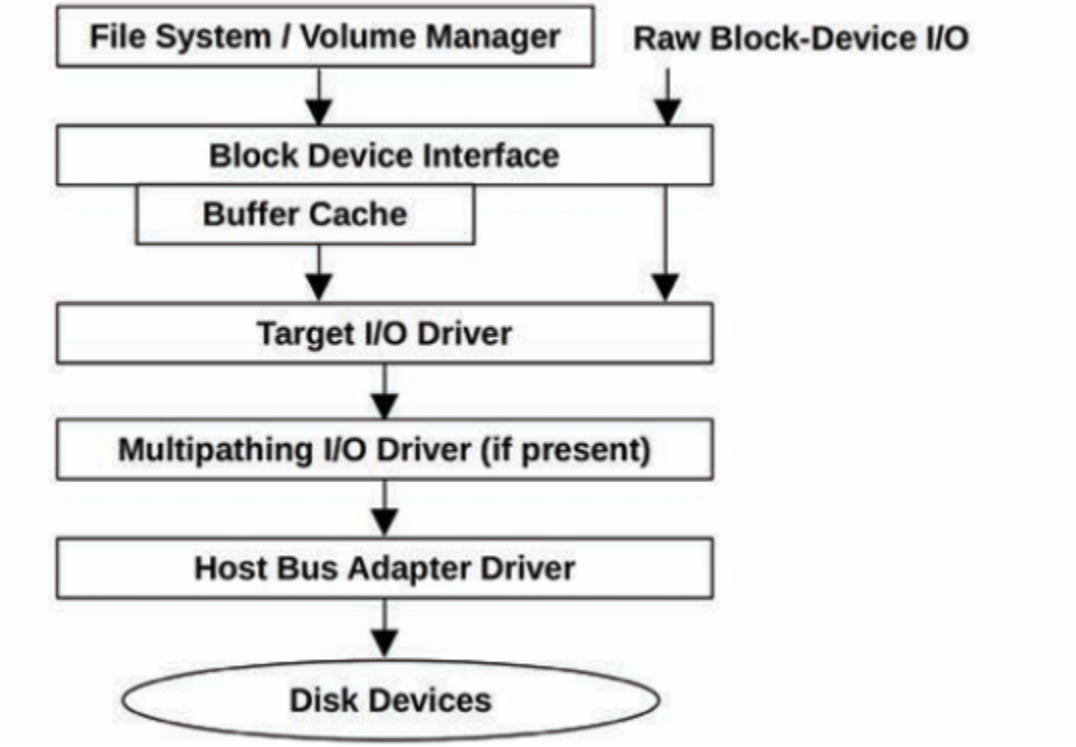
    - Per-CPU caches to minimize locking contention
    - Direct slab access without complex queuing structures
        - Queues for each partial, full, empty slabs
    - NUMA awareness for multi-processor systems
- Page allocator - global
- Lowest levvel out of all allocators
- Manages dynamic virtual memory
- Uses free list and bitmap systems
    - Usually numa node specific?
- Combines small alloc (below page size, like malloc) into pages to prevent internal fragmentation
- Linux uses buddy allocator algorithm
    - Each numa node has its own free lists
    - Free lists are organized in buckets: specific powers of two
        - Freeing happens in buddies so that way we don't need extensive searching or linked lists
        - Basically, we check buddy if it needs to be freed if curr is freeable
        - What determines a buddy: 2^n block at address A -> its buddy is at A XOR 2^n (vise versa)
- Libc allocator - per process (each gets one)
- Allocates heap space for user processes (dynamic memory)
    - Uses segregated free lists, buddy allocation, binning for fast alloc/dealloc
        - Binning sorts by size
    - May have locks to prevent corruption for malloc/free
- Uses page allocator if can't serve request (kernel space, requires syscall)

Disk

Block vs sector

- File system:
    - Block - smaller than a page
    - Logical blocks are abstraction of file system disk space. Tehy are mapped to physical blocks, with some offset (locial 1 is physical 123)
- Ssd and hdd
    - Physical block - chunk of disk space; 128-256 pages or more
        - Is smallest unit in ssd
        - Smallest unit that os deals with
            - Pages are smaller tho (4 kb to 16 kb)
    - Sectors - 512 bytes (used to be) to 4 kb (now)
        - Smallest unit of storage in disk
- Disk controller does translation from logical block to physical
    - There's a table - physical : logical
        - Disk controller when gets request for block, reads this then serves physical block

    - Why mapping
        - Wear leveling - this helps spread out writes so not one area is more worn out than other
        - If part of disk is damaged, can just map to others
        - Abstraction - can reorder physical without changing logical
    -
- Layers for io

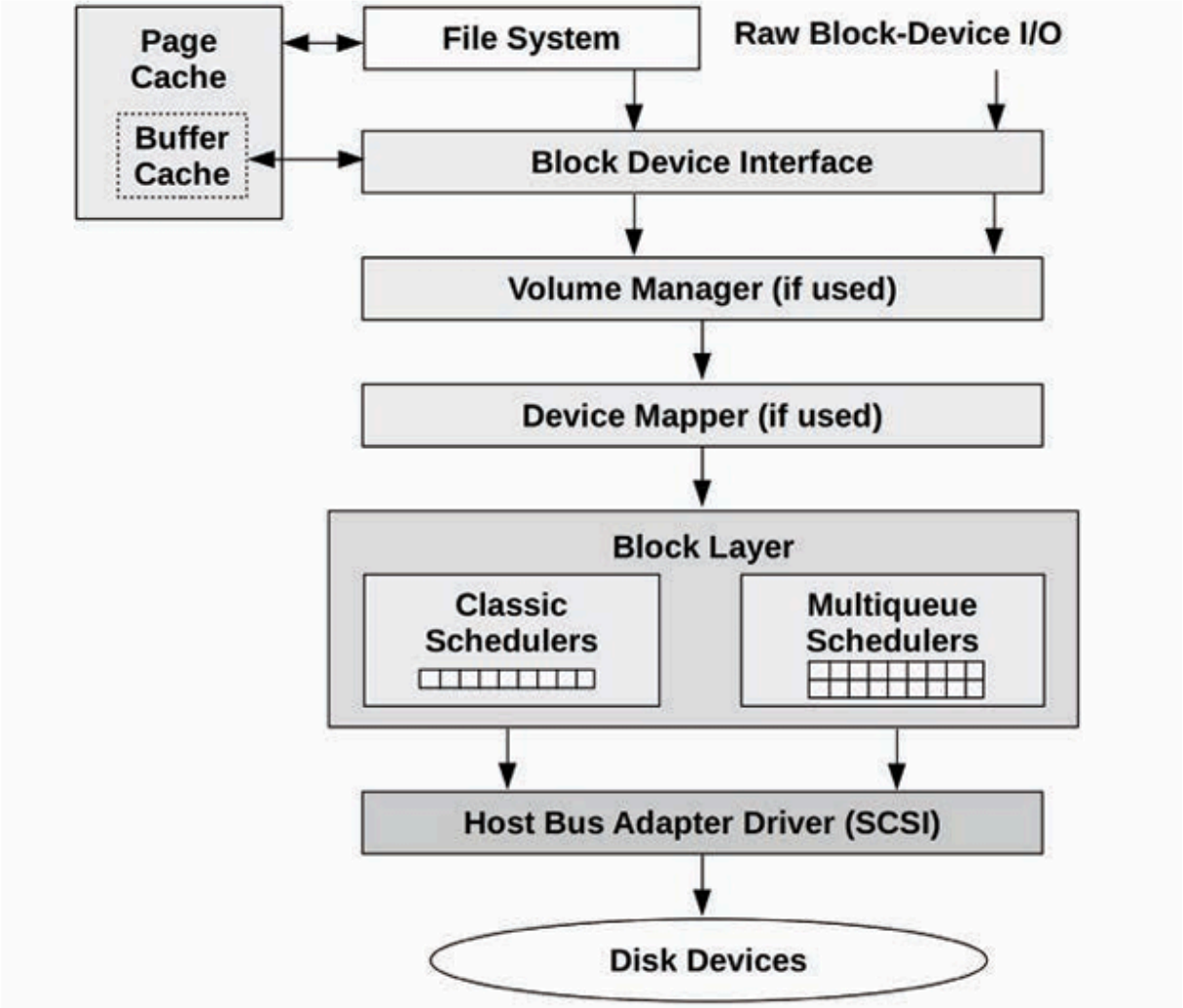


- application
- File system + volume manager combines many disks into logical volumes. File system is interface to interact with disk
- Block device interface handles file system to block translations (db's can do raw so no file system translation)
    - Buffer cache stores frequently accessed disks in memory, also merges writebacks for batching
- Io scheduler - bfq, kyber
- FOR NETWORK STORAGE ONLY: Target io driver converts block (logical) commands to disk commands and sends to specific disk controllers
- Device drivers - for servers
    - Multipathing is load balancer + redirects if one path fails
        - Multipathing means having different physical routes between server and storage (good for parallelism, redundancy/fault toelrance)
    - Hba - is the interfaces - sends io commands to physical storage
        - Host bus adapter. Connects server to storage devices
        -

- Storage device
-
- Special cases
- Direct io - bypasses buffer io cache but nothing else
    - Avoids app + os caching
- Raw block io - skips cache, file system, target io. Sometimes can skip multipathing
    - Must handle metadata, placement, consistency
- Block Io buffer
- Buffer cache
    - Read in -> items stored in buffer. If accessed again, served from here
    - Write -> items store in here. Merges happens so write back in batches, saving disk trips
- Merging
    - Front merge - merge curr in front of prev since they are contiguous curr->prev
    - Back merge - merge curr after prev since they are contiguous prev->curr
    - Lowers cpu/io overhead
- Is buffer where physical disk blocks are mapped to memory. When these are changed, they become dirty
    - Used for reading/writing to disk
- Dirty is flushed back when:
    - Fsync or sync system calls are called
    - Dirty pages become old (avoid data loss if crash)
    - Memory falls below certain threshold
- Can use sysctl to change freq of writeback + threshold for dirty data
- Buffer head vs bio
- Each buffer has buffer head - describes meta data like flags, associated page in memory, block disk number, pointers to related data.
    - Buffer head describes portion of individual pages; must be chained to represent more pages
    - Chaining: due to being split up, needd chain of buffer heads one for each page
- Tied to page caches, making it not scalable. Had to split up large io requests multiple times to fit one page each. Required chaining multiple noncontiguous buffers due to being split apart
    - Now replaced with bio, which represents a single io op
    - Buffer io can span across many pages
        - Supports scatter-gather io (non contiguous)
          Therefore can use dma
    - Important fields
        - struct bio_vec *bi_io_vec → An array of segments within pages.
        - unsigned int bi_vcnt → Number of segments in bi_io_vec.
        - sector_t bi_sector → Start sector of the I/O operation.
        - struct block_device *bi_bdev → Target block device.
        - bio_end_io_t *bi_end_io → Completion handler.

    - Bio_vec - represents segment within a page; has fields representing page, offset in memory, length of data in segment
        - Points to each segment in page to actually create segment
    - Many buffer heads -> just an array of bio_vec. Less overhead without chaining
- Block devices have queues, where schedulers optimize them
-
- Bio usage
- Allocate a bio structure.
- Fill it with pages and offsets using bio_vec.
- Submit it to the block layer.
- The block layer processes the I/O request and calls the completion handler.
- # Sectors vs blocks
- Sectors are smallest addressable units on disk, 512 bytes usually, up to 2 kb
    - Physical property
- Blocks are logical properties. Smallest unit. Built on powers of 2; cannot be larger than memory page. 512 bytes, 1 kb, 4 kb are typical sizes
- Block devices
- They are hdd, ssd, usb drives, raid arrays
- Allow random access; managed by block layer. Use buffer/cache data, unlike character devices
- # raid
- Disk devices - controller allows disk to be seen/accessed. Disk connects to this
    - Jbod - just a bunch of disks
- Raid - multiple inexpensive disks. Often has ram for caching
    - Can be implemented with hardware (controller) or software
        - Hardware is better since cpu checksum + parity calcs are expensive. Better cpus nowadays mitigate that
        - Software easier to fix than dead hardware
- Raid 0 - stripign to improve concurrent access (half of segment in A and othe rhalf in B)
- Raid 1 - mirroring to improve reliability
- Raid 0-1 - mirror of stripes (stripe, then mirror)
- Raid 1-0 - stripe of mirrors (mirror, then stripe the dupes)
- Raid 5 - parity + 0/1
    - Like 1, but allows for more groups due to striping, increasin badnwidth
- Raid 6 - dual parity + 0/1
    - Worse than 5
- 10
    - `
- 3,4,2 not used really
    -
- Parity
    - Is sum of xor of all data; extra memory overhead to store this
    - This way, can ensure data does not get corrupted

    - If a disk fails, then xor remaining with parity to get back that disk (retrace steps)
    - Double parity means can tolerate up to 2 failures
    - Bad because:
        - Writes require rewriting of parity
        - Rebuild can be complex due to reading + computing
- Stripe width
    - Can't be too thin otherwise have to split up io into many excessive small io operations
    - Can't be too thick otherwise no parallelism since all operations just done in one
    - Balance to match workload; smaller for random (would not be next to each other so likely to land on different stripes anyways) and larger for sequential (able to take advantage of larger and not be split up across stripes)
- Structure



- Volume manager

A **volume manager** (e.g., **LVM**, **ZFS**, **MD RAID**) abstracts multiple physical disks into a logical volume, allowing the OS to see a **single logical device** rather than individual disks. When the OS sends a block I/O request, the volume manager determines **which physical disk(s) the request should be mapped to** based on the volume configuration.

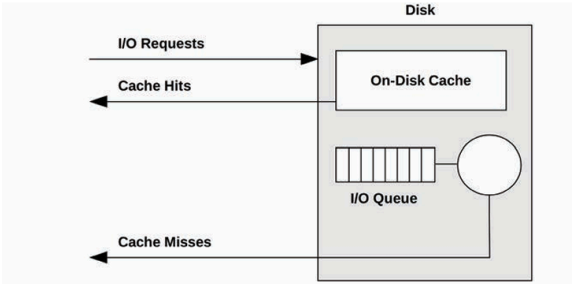
Translation Process:

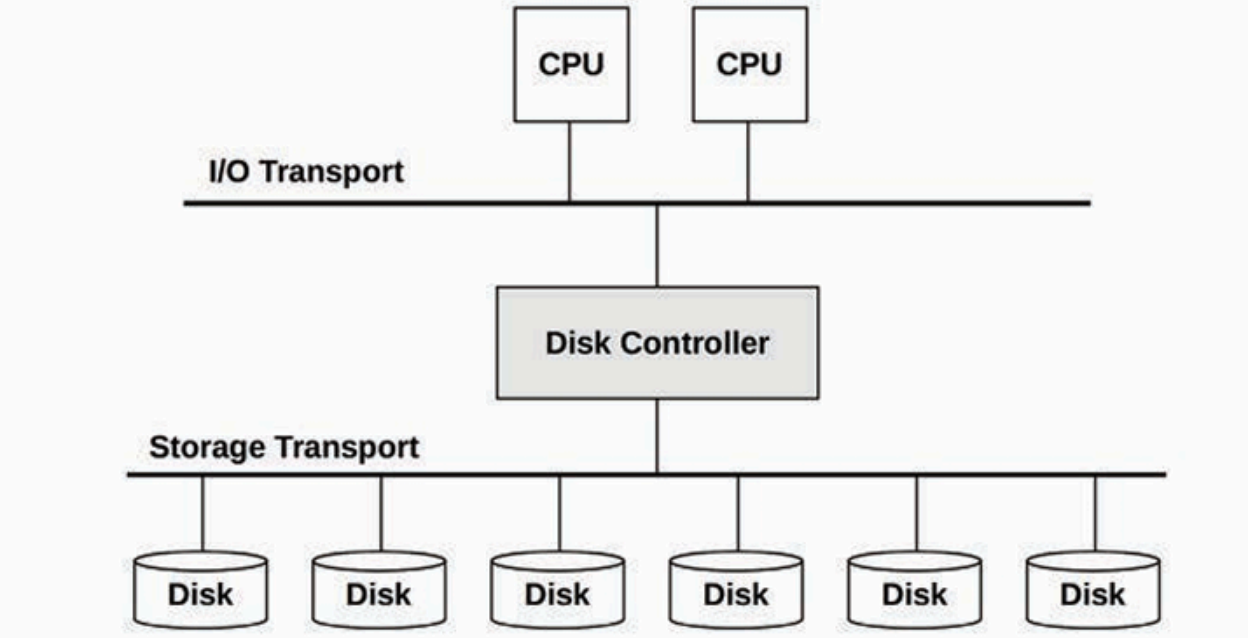
1. **Logical Block Addressing (LBA) → Physical Disk Mapping**
    - The OS and file system see a single **logical address space** for the volume.
    - The volume manager determines which **physical disk and block offset** the request should be directed to.
2. **Different Volume Configurations and Translations**:
    - **Striping (RAID 0, ZFS Striped VDEVs, LVM Striped Volumes)**
        - The volume manager **splits** incoming writes across multiple disks in a round-robin or pre-defined pattern.
        - Example: A 64KB write may be split into two 32KB writes to two separate disks.
    - **Mirroring (RAID 1, ZFS Mirrors, LVM Mirrored Volumes)**
        - The same block request is **duplicated** and sent to multiple disks.
        - Reads can be **load-balanced** across both copies.
    - **RAID 5/6 (Striping + Parity)**
        - The volume manager calculates **parity data** and distributes it across multiple disks.
        - A read request requires fetching data from multiple disks and reconstructing missing data if a disk has failed.
    - **Logical Volume Allocation (LVM, ZFS, etc.)**
        - If a volume spans multiple physical disks but doesn't use striping, the volume manager looks up a **metadata table** to determine which disk the requested block resides on.
        - Example: If LVM has two disks and extends a volume across them, the first part of the volume may be on **Disk A**, and the second part may be on **Disk B**.
3. **Pass the Request to the Block Layer**
    - Translates logical block request to physical block request and sends to block device layer
-

- Scheduliers
- types
- Noop - no schedule; overhead too much for situation
    - Basically fifo + some adjacent merging
- Deadline - can starve due to elevators + priority. Has read fifo, write fifo, sorted queues. Useful for real time systems

    - Sorts requests by physical location (elevator)
    - Goal is to abide by deadlines
- Anticipatory
    - Is deadline but have delay after read in case there are other reads; that way we can aggregate and make lesas trips to disk
- Cfq - allocates io time slices to processes based on niceness, like round robin
- All of the above are bad due to being protected by 1 lock
- Multi level:
    - Bfq - creates queue for each process with a set budget. Each process has bandwidth + time budget. System wide budget too. Supports cgroups
        - Once budget runs out, if there are no others waiting for io, give more budget
    - Mq-deadline. Assigns deadlines. Prioritizes reads over writes. Good for db/enterprise storage
    - Kyber - adjust read/write dispatch queue lengths so that target read/writes can be met. Target read latency and target synchro write latency are only tunables. Good for low latency ssd workloads
        - Shortens queue length if requests are taking too long
            - less latency like this due to smaller queue but we also have less throughput
        - Extends if too fast

Types

- Simple disk
    - Io requests go into queue then go out as completed output
- Simple disk with cache. Can be write back or write through. Can have batteries if writeback as backup power
    - 



- Io wait time - wait for io, io service time - time io request being fulfilled, io request time = io service + wait time
- Kernel io times are referred to as block io times
- Disk wait tiems are referred to as disk wait times

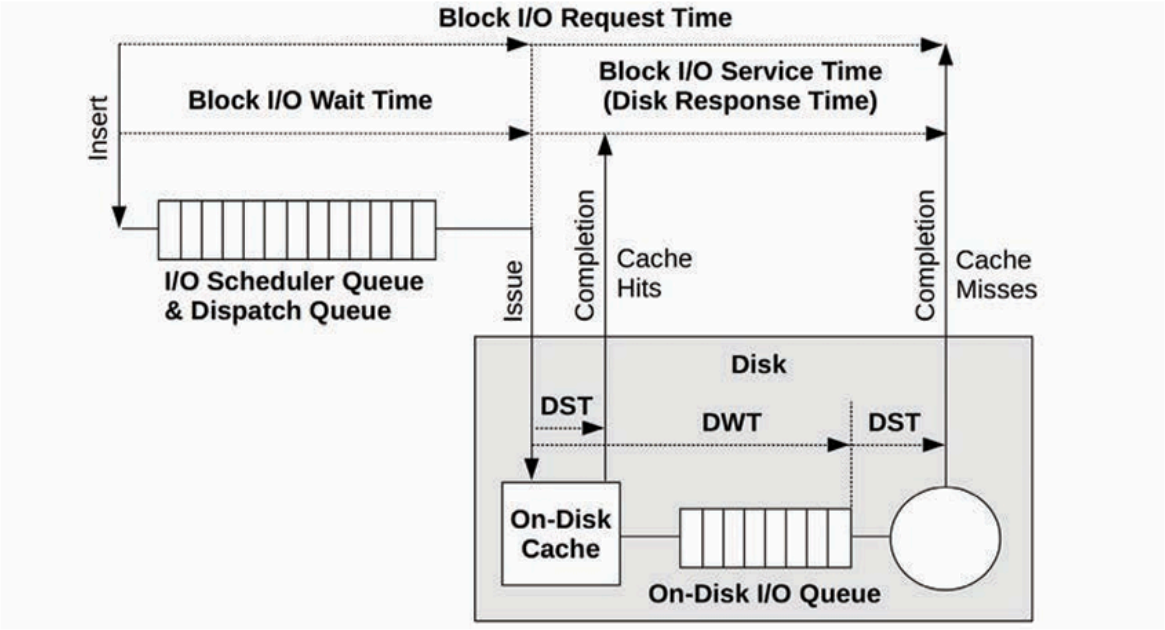


Figure 9.5 Kernel and disk time terminology

- Disk service time = utilization/IOPS
    - Utilization of 60% is expressed as 600 ms

Table 9.1 **Example time scale of disk I/O latencies**

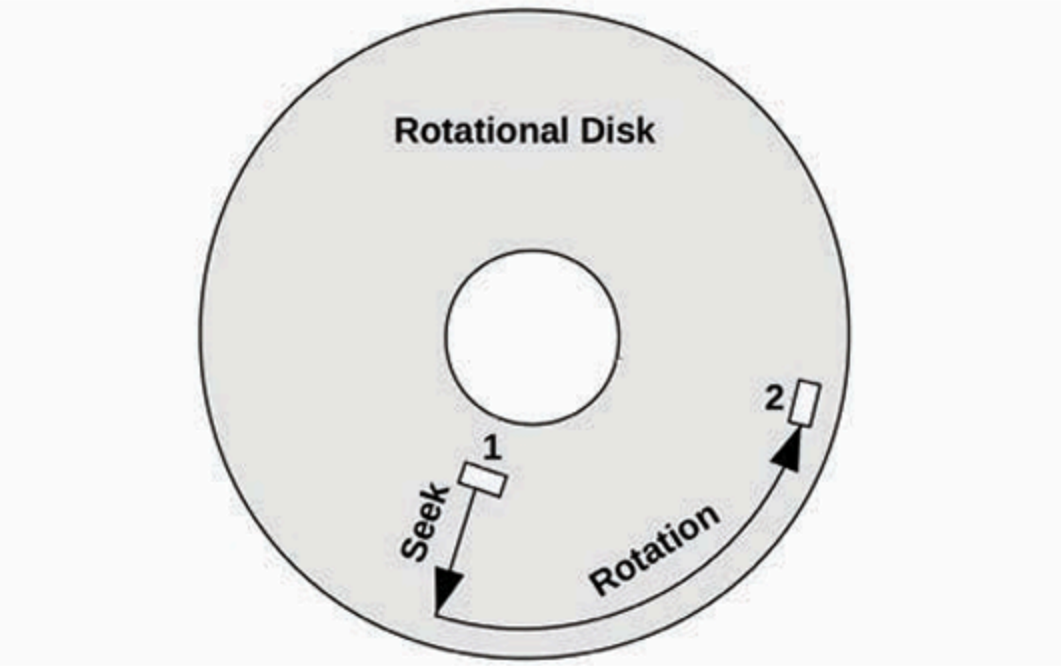
Event	Latency	Scaled
On-disk cache hit	< 100 µs[1]	1 s
Flash memory read	~100 to 1,000 µs (small to large I/O)	1 to 10 s
Rotational disk sequential read	~1 ms	10 s
Rotational disk random read (7,200 rpm)	~8 ms	1.3 minutes
Rotational disk random read (slow, queueing)	> 10 ms	1.7 minutes
Rotational disk random read (dozens in queue)	> 100 ms	17 minutes
Worst-case virtual disk I/O (hardware controller, RAID-5, queueing, random I/O)	> 1,000 ms	2.8 hours

-
- Anything over 10 ms is SUS for reading disk io for storage industry; web servers over 50 ms

Table 9.2 **Disk I/O caches**

Cache	Example
Device cache	ZFS vdev
Block cache	Buffer cache
Disk controller cache	RAID card cache
Storage array cache	Array cache
On-disk cache	Disk data controller (DDC) attached DRAM

-
- Sequential also known as streaming



onal disk

-
- Understanding io ratio helps with designing system
    - High reads -> cache
    - High writes -> more disks to increase iops, throughput
- Asynch io commonly is due to writeback
-
- Hdd
    - max throughput = max sectors per track × sector size × rpm/60 s
    - Best to have reads be sequential for disks so don't have to seek
    - Short stoking - outer part of disk used; inner parts used for low importance work
    - Sector zoning - longer tracks have higher sector count. Longer tracks are on outside. Shorter are on inside. Rotation speed is constant, so outside travels faster (more nonrotational distance covered per rotation)
    - Elevator seeking - eorders io based on disk location, doing things closer to curr first
        - Bad since new arriving io can starve further io
        - Merges adjacent requests (front and back merging) and sorts requests by disk sector
        - Old requests are added to tail, even if unoptimal. This avoids starvation
    - Error correcting code exists at end of each sector for data integrity
        - Can be why disk takes a while
    - Vibrations can cause slow writes
    - Sloth disk - slow disk without raising errors

    - Shingled Magnetic Recording (SMR) drives provide higher density by using narrower tracks. - more compact and suitable for reads but not as much writes. Writes destroy other tracks and those must be rewritten
    - Disk data controller - controls disk, not visible by os. Includes sector zoning
- Ssd - mostly flash memory. Some use dram. Nonvolatile
    - Most are built using NAND flash memory, which uses electron-based trapped-charge storage media that can store electrons persistently4 in a no-power state
        - Has multiple nand channels; ar elike highways for data; allows for more parallelism
            - Each channel onects to multiple nand dies (chips), allowing for more simultaneous access
    - Batching still important because:
        - For parallel writes, want writes to end up in same area, but if we have small ones, we have lots of internal fragmentation when we run this in parallel (chips can't modify same blocks). If we have larger one, we can split up and write in parallel
        - Do larger writes so that way don't have to rewrite to write in a partially filled block
    - Not susceptible to vibration
    - Optimized for 4 kb reads, 1 mb writes
    - Os thinks ssd is just normal hard drive, so it sends writes/reads for 512 bytes to 4 kb, and ssd translates these
        - Ftl handles, this. It is flash translation layer in ssd. Maps logical sectors to physical pages
    - Ftl - flash translation layer; translates logical block addresses to physical nand pages or blocks (pages higher memory overhead but fine grained)
    - reads/writes happen per page (pages are 8 kb to 64 kb)
    - Writes only happen on empty blocks. Can't overwrite like hdd
        - Pages are erased in blocks (256 to 512 kb per block)
        - If need to update block, must erase then rewrite whole block
        - How it's done
            - Copy remaining valid data elsewhere
            - Erase block
            - Write back with modified valid data
        - Thus fast reads slow writes
            - Writeback withi small capacitor as battery is soln
            - Also write in blocks, not anything smaller than that to maximize usage of writes (like no internal fragmentation)
    - Deelting is complicated therefore; need asynch garbage collection
        - Must mark as stale, then garbage collection erases
            - Trim can be used to optimize; if know how many are invalid, then if block is pretty much invalid, can just delete nad not move around
            - Copies valid data elsewhere

            - Erase block
            - Rewrites with modified valid data but no stale data
        - Zoned namespace ssd's delete in large zones to prevent moving
            - Ssd is split up into zones. Writes to zones are sequential. Once zone is full, is marked for reset. This eliminates overhead of garbage collection. Is simpler: once full and not needed, remove. Some moving happens to move valid data but not as much since zones don't fill up as fast due to benign large
    - Cell types
        - **Single-level cell (SLC)**: Stores data bits in individual cells.
        - **Multi-level cell (MLC)**: Stores multiple bits per cell (usually two, which requires four voltage levels).
        - **Enterprise multi-level cell (eMLC)**: MLC with advanced firmware intended for enterprise use.
        - **Tri-level cell (TLC)**: Stores three bits (eight voltage levels).
        - **Quad-level cell (QLC)**: Stores four bits.
        - **3D NAND / Vertical NAND (V-NAND)**: This stacks layers of flash memory (e.g., TLC) to increase the density and storage capacity.
    -
    - Basically single level cell is best, but is most expensive
    - Mlc is most common (high density but worse flash reliability)
    - **SLC (Single-Level Cell) → Best lifespan**, ~1M write cycles (used in enterprise SSDs).
    - **MLC (Multi-Level Cell) → Shorter lifespan**, ~1,000–10,000 cycles (consumer SSDs).
    - **TLC (Triple-Level Cell) → Even worse**, ~300–3,000 cycles (budget SSDs).
    - **QLC (Quad-Level Cell) → Worst endurance**, ~100–1,000 cycles (cheap SSDs).
    -
    - Trim commands lets ssd know which blocks are truly free since os doesn't tell ssd that, just what is available. This prevents write amplification and therefore aging
        - Basically enables os to send command to tell ssd that block is deleted/freed up. Then ftl removes mapping
        - Not default since not every ssd supports it without bugs, plus it causes write/delete overhead
    - Overprovisioning - have extra space in disk than is listed so don't have to garbage collect immediately
    - Controller responsibilities
        - Input: Reads and writes occur per page (usually 8 Kbytes); writes can occur only to erased pages; pages are erased in blocks of 32 to 64 (256–512 Kbytes).

        - Output: Emulates a hard drive block interface: reads or writes of arbitrary sectors (512 bytes or 4 Kbytes).
        - Move around data to prevent aging
    - Extending ssd lifespans
        - Wear levling - spread out writes across many blocks to not overfocus on one
        - Have extra backup memory
        - Move hot data around to prevent overusage of specific blocks
    - Aging problems with ssd
        - Latency Spikes (Outliers)
            - As NAND flash wears out, it takes longer to retrieve data. Data becomes corrupted. Nand flash has limited write/erase cycle count (3000 for tlc). Ecc tries to fix this but too far gone and can't fix it
            - The SSD retries reads and applies ECC (Error Correction Code) to fix bad bits.
            - This causes occasional high-latency reads (bad for real-time applications).
        - ⚠️ Higher Latency Due to Fragmentation
            - The Flash Translation Layer (FTL) manages logical-to-physical mapping.
            - If too many small writes happen, the FTL's block map gets messy (like a fragmented HDD).
                - Fragmentation causes garbage collection to have to do more work, increasing latency
            - Fix? Reformatting the SSD may clean up the FTL and improve performance.
        - ⚠️ Lower Performance Due to Compression
            - Some SSDs use internal compression to store more data in less space.
            - If the data is already compressed (e.g., encrypted files, videos), the SSD can't compress further.
                - Need to uncompress to fetch; not good for latency
            - This leads to lower-than-expected write speeds.
- Storage interfaces

This section gives an **overview of different storage interfaces** used in computing, from older technologies like **SCSI** to modern high-speed storage like **NVMe**. Let's break it down:

---

1. SCSI (Small Computer System Interface)

- **Old-school storage interface (1980s)**

- Originally **parallel** (data sent across multiple wires at once)
- Early versions were **slow** (SCSI-1: 5 MB/s, 8-bit bus)
- **Problems:** Shared bus caused performance issues (bus contention) when multiple devices were active.
- Later, **SAS (Serial Attached SCSI)** replaced parallel SCSI.

2. SAS (Serial Attached SCSI)

- **Modern replacement for SCSI** (2003+)
- Uses **serial** connections (point-to-point) instead of a shared bus.
- **Faster speeds:**
    - SAS-1: **3 Gbps** (2003)
    - SAS-2: **6 Gbps** (2009)
    - SAS-3: **12 Gbps** (2012)
    - SAS-4: **22.5 Gbps** (2017)
- Supports **multi-pathing, redundancy, and enterprise features.**
- **Common in servers & enterprise storage** (e.g., RAID setups).

---

3. SATA (Serial ATA)

- Replaced **Parallel ATA (PATA/IDE)** for consumer storage.
- **Speeds:**
    - SATA 1.0: **1.5 Gbps** (2003)
    - SATA 2.0: **3.0 Gbps** (2004)
    - SATA 3.0: **6.0 Gbps** (2008)
- **Slower than SAS but cheaper.** Common in **desktop & laptop SSDs.**
- Uses **8b/10b encoding** (actual data transfer is 80% of max bandwidth).

---

4. Fibre Channel (FC)

- **High-speed enterprise storage**
- Originally designed for **fiber optics**, now also supports copper.
- Used in **Storage Area Networks (SANs)** to connect **multiple servers to shared storage.**
- **Fast speeds:** Latest **Gen 7 Fibre Channel** supports **256GFC** (51.2 GB/s full-duplex).
- **Expensive** but **reliable for large-scale enterprise environments.**

---

5. NVMe (Non-Volatile Memory Express)

- **Next-gen storage standard for SSDs** (2011+)
- **Key difference: NVMe devices connect directly to the PCIe bus.**
  (Unlike SATA or SAS, which go through a storage controller.)
- **Speeds depend on PCIe version:**
  - PCIe 4.0 (current standard): **31.5 GB/s (x16 lanes)**
  - PCIe 5.0 & 6.0 will be even faster.
- **Why NVMe is faster than SATA/SAS:** ✅ Supports **multiple queues** (up to **64,000 commands per queue**).
  ✅ **Lower latency (~20 microseconds)** compared to SATA/SAS.
  ✅ Better suited for **high-speed SSDs** (used in gaming PCs, cloud servers, and data centers).
- **Supports SR-IOV** (important for **virtualization** in cloud computing).

TL;DR

- **SCSI (Old Parallel)** → Replaced by **SAS (Fast Serial, used in enterprise).**
- **ATA (Old Parallel)** → Replaced by **SATA (Serial, used in consumer devices).**
- **Fibre Channel (FC)** → Used for **enterprise SANs (high-speed shared storage).**
- **NVMe (PCIe-based SSDs)** → The **fastest** and most modern storage interface.

Which One Should You Use?

- **Consumer PCs/laptops? → SATA SSDs or NVMe SSDs.**
- **Servers & high-end workstations? → NVMe or SAS SSDs.**
- **Enterprise data centers? → Fibre Channel (FC) or NVMe-over-Fabric.**

Let me know if you want further clarification on anything! 🚀

-

File system

- Superblock
- Filesystem control block
- Contains inode list, dirty inode list, used dentries, dentries, flags, block size in bytes/bits, etc.
- File object
- Is in memory rep of open file

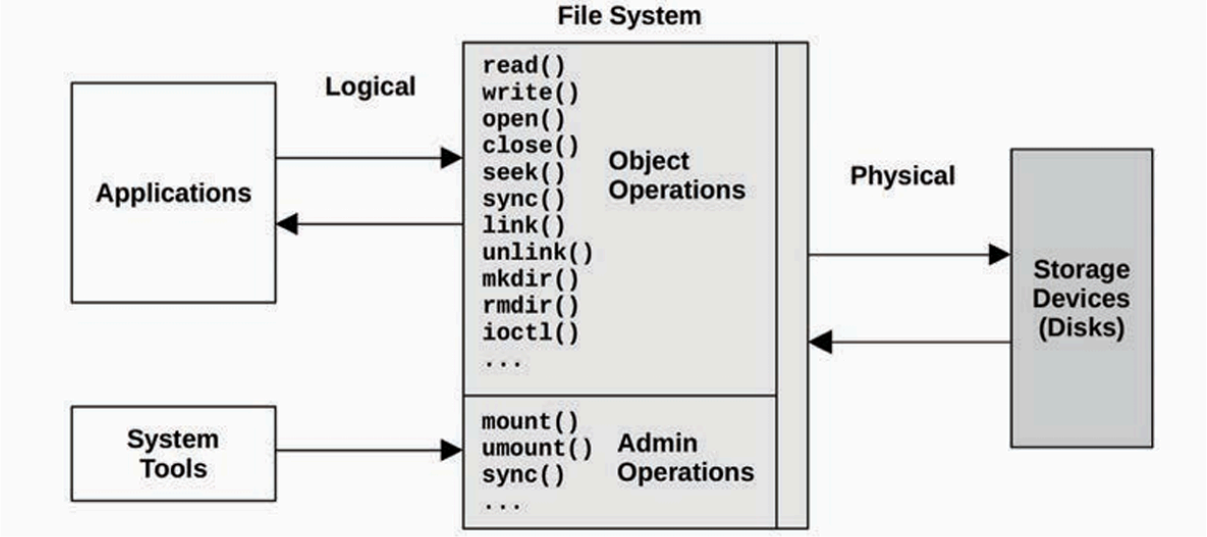
```
struct file {
        union {
            struct list_head   fu_list;       /* list of file objects */
            struct rcu_head    fu_rcuhead;    /* RCU list after freeing */
        } f_u;
        struct path            f_path;        /* contains the dentry */
        struct file_operations *f_op;         /* file operations table */
        spinlock_t             f_lock;        /* per-file struct lock */
        atomic_t               f_count;       /* file object's usage count */
        unsigned int           f_flags;       /* flags specified on open */
        mode_t                 f_mode;        /* file access mode */
        loff_t                 f_pos;         /* file offset (file pointer) */
        struct fown_struct     f_owner;       /* owner data for signals */
        const struct cred      *f_cred;       /* file credentials */
        struct file_ra_state   f_ra;          /* read-ahead state */
        u64                    f_version;     /* version number */
        void                   *f_security;   /* security module */
        void                   *private_data; /* tty driver hook */
```



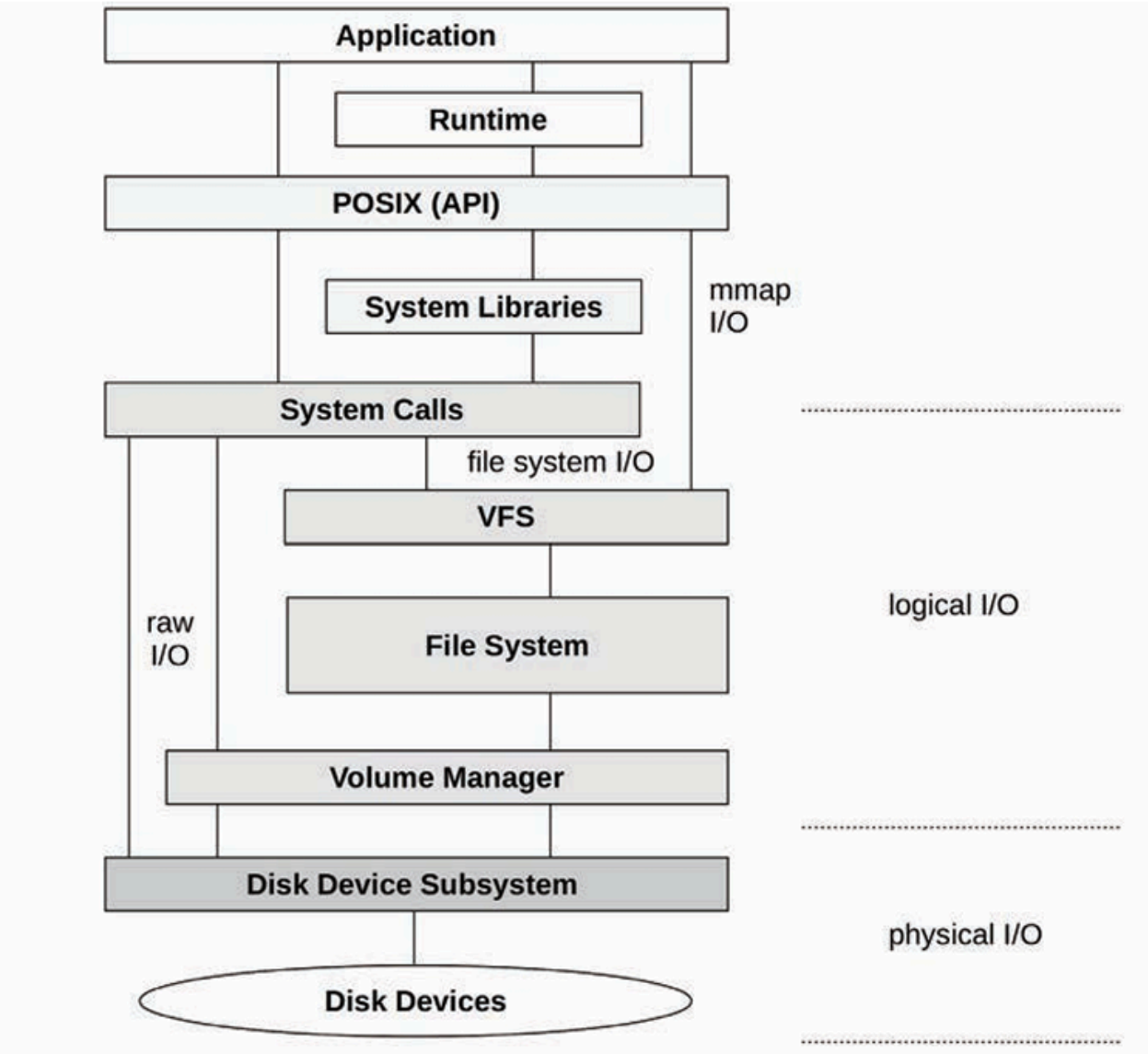
```
        struct list_head       f_ep_links;    /* list of epoll links */
        spinlock_t             f_ep_lock;     /* epoll lock */
        struct address_space   *f_mapping;    /* page cache mapping */
        unsigned long          f_mnt_write_state; /* debugging state */
};
```

-
- Dentry
- Helps with file path resolution
- Unused - does not point to inode
- Used - points to valid inode and has positive reference count
- Negative - empty but cached to speed up lookups (nonexistent)
-
- Is in disk, so cache is needed to speed up lookup
- Reference count - how many references an inode or dentry has (like how many processes are running)
-
- The dentry cache includes:

    - A list of used dentries linked to their associated inode.
    - A "least recently used" (LRU) list for unused and negative dentries.
    - A hash table for fast lookup of dentries based on file path components.
- misc



-



-
- File system cache - is cache in memory for files
- Ram (first level cache) < flash (2nd level cache) < disk
    - For file speed

Table 8.1 **Example cache types**

Cache	Example
Page cache	Operating system page cache
File system primary cache	ZFS ARC
File system secondary cache	ZFS L2ARC
Directory cache	dentry cache
inode cache	inode cache
Device cache	ZFS vdev
Block device cache	Buffer cache

-
- Prefetch aka readahead
    - Is useful for data too big to fit in cache
    - Only for sequential data
        - If curr ptr + 1 == next ptr
    - Before first read, call first read + prefetch. After first read (and kernel passes this data back to user process):
    - Basically will have put next parts of file data in cache for future reads already
    - Not on by default
- Writeback is default usually, but have options for write through (synchronous) for more important data (if power outage were to occur, where you don't want to lose this data)
    - Synchronous for database log writers (backups)
    - Synchronous can be in batches at a checkpoint, just like how asynchronous writebacks are written back at flushes
- Direct io - use filesystem but no cache. Usually disables read cache, write buffers, prefetch
- Raw io - less overhead but more complex since need to mange own writes with disk offsets. Usually best for db so can use its cache instead of file system's
- Memory mapped - directly nmapping files to processes address space and utilizing address offsets. Faster than using syscalls that require context switches
- Degrading performance at capacity - longer to find free block (disk io, cpu higher)
- File defrag
- Defragmentation happens in filesystem level (EXT4, XFS, etc.).
- fsck and tools like e4defrag can defragment filesystems. Not automatic
    - Are background processes
        - Rewrites disks in chunks to avoid blocking
    - Xfs also does it
- Zfs is automatic

- This helps avoid performance issues due to excessive fragmentation.
- Logical vs physical
- Explicit Logical statistics - eading file statistics (stat(2)), creating and deleting files (creat(2), unlink(2)) and directories (mkdir(2), rmdir(2)), setting file properties (chown(2), chmod(2
- Implicit - File system access timestamp updates, directory modification timestamp
    - updates, used-block bitmap updates, free space statistic
- Diff stats are reason for logical vs physical
- Other disk io besides application's syscall read/write (will not show up in strace always)
- Other applications, tenants in cloud servers, rebuilding software raid volume, file syste checksum, backups
- indirect: prefetch, buffer
- Implicit: memory mapped
- Deflated - very very small
    - Cache, write cancellation (write many times before flush back to disk), coalescing (merge sequential io before write back to disk), in memory file system (proc), writing extra parity data
- Microbenchmarked runtimes of file system syscalls

Table 8.2 **Example file system operation latencies**

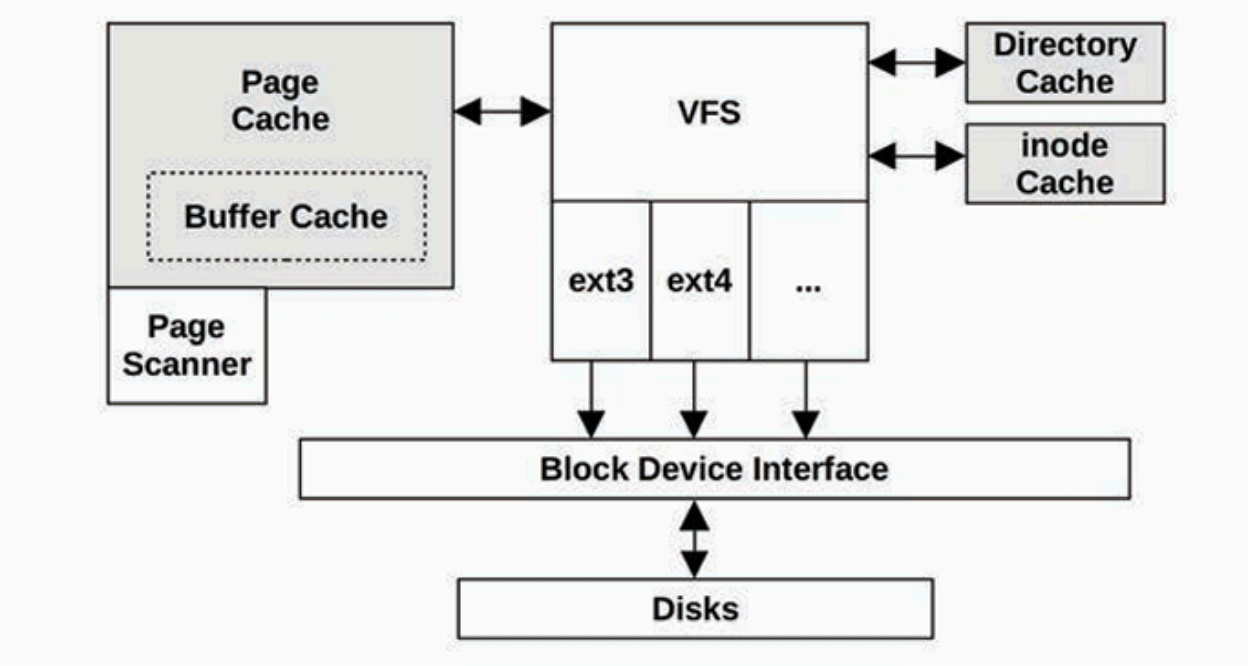
Operation	Average (µs)
open(2) (cached[2])	2.2
close(2) (clean[3])	0.7

[2] With the file inode cached.
[3] Without dirty data that needs to be flushed to disk.

Operation	Average (µs)
read(2) 4 Kbytes (cached)	3.3
read(2) 128 Kbytes (cached)	13.9
write(2) 4 Kbytes (async)	9.3
write(2) 128 Kbytes (async)	55.2

-
- Do it on your own system too
- Above was single threaded
- Special file systems
- Proc
- Tmp
- Dev
- Sys

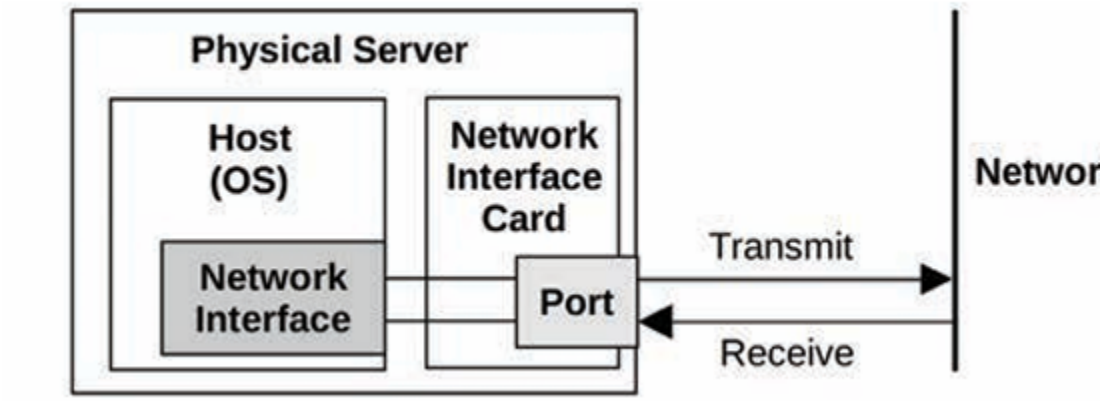
- Vfs
- Is interface between system calls and file system type (implementation)
- Caches



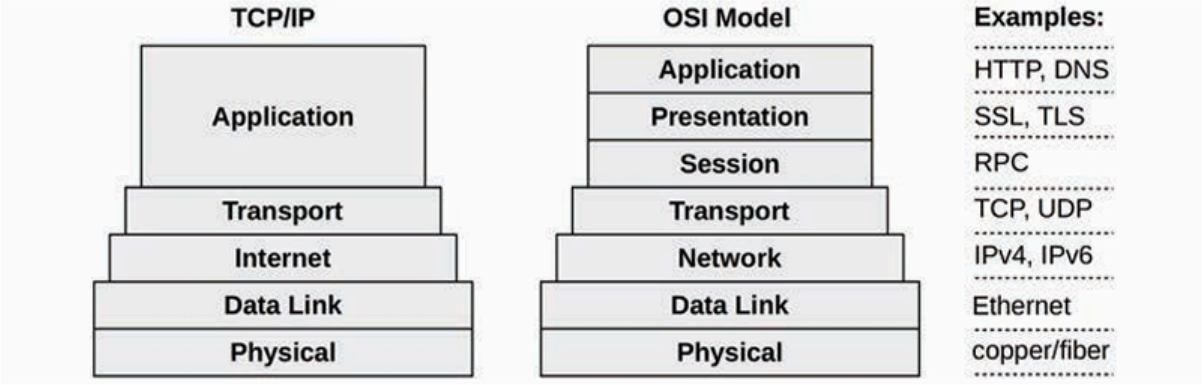
- Buffer cache used to cache blocks from disk, but now is part of page cache
    - Due to being difficult to keep both buffer cache and page cache in sync + tuned
    - Buffer cache needed translation between file offset to disk offset
    - Page cache is dynamic; buffer cache wasn't (now is)
- Pages are flushed to disk b/c:
    - After 30 seconds
    - Sync, fsync, msync calls
    - Too many dirty pages (dirty ratio and dirty bytes (tunables)
    - No available pages in cache
- Dentry cache
    -
- Inode cache
    -
- File system features
    - Block vs extent
        - Block is fixed size; metadata points fo these blocks, but for large files, this may lead to too much random io
        - Extent - continguous space that grows as needed; this leads to less random io (but more fragmentation)
            - We don't want random io in file system; takes too long to read
    - Journalilng

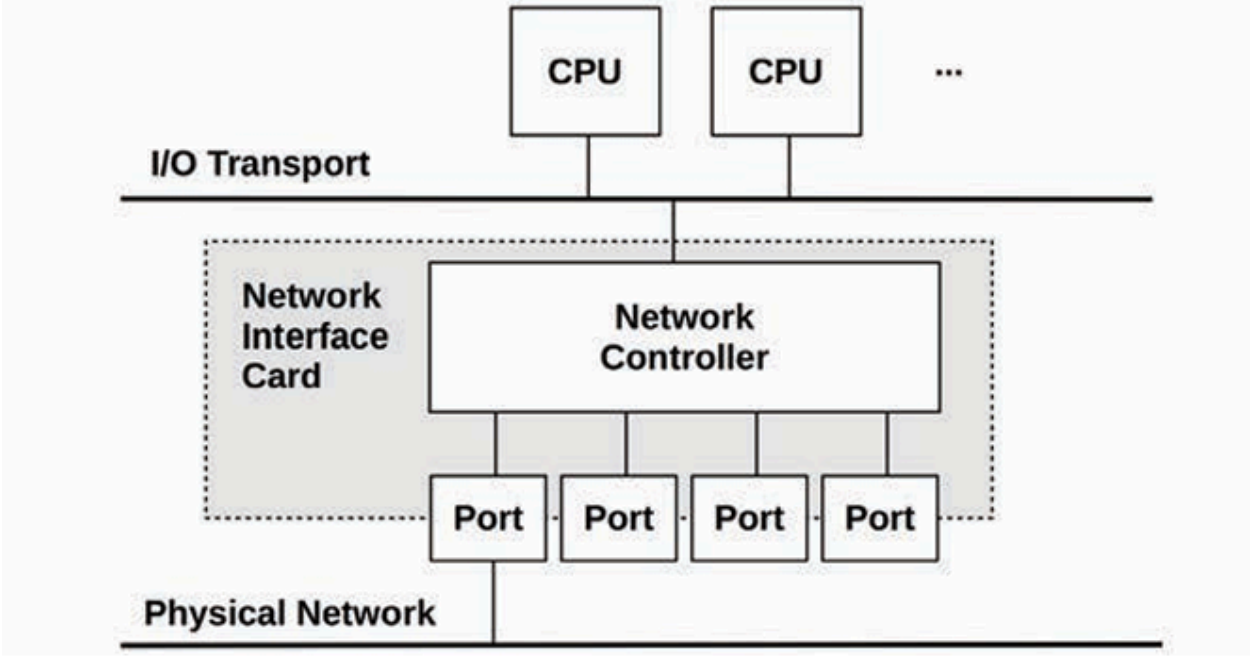
        - Back up: logs so that way changes right before crash can be replayed to replicate chagnes lost to the crash
    - Scrubbing -> checks all data blocks asynch and verifies checksums to detect failed drives as early as possible
        - Hurts read io; should only be done as low priority and times with low workload
    - Volumes -> many disks interfaced as one in file system. One volume per file system
        - LVM and RAID controllers control this
    - Pool - more flexible than volumes. Multiple file systems share pool; can grow or shrink
        - Used by zfs or btrfs. Also possible with lvm
        - Hard to monitor; must checkc individual physical devices
        - Adds cpu overhead due to parity calculations; can be offloaded now to raid controllers
        -
    -
- Example file systems
- `

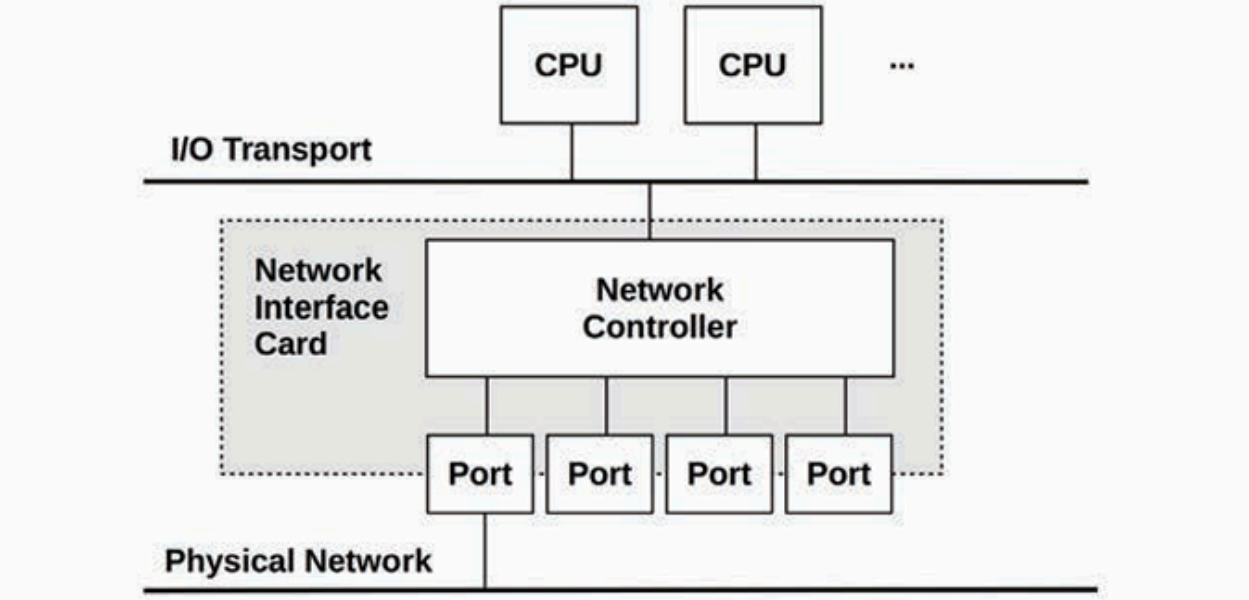
Network



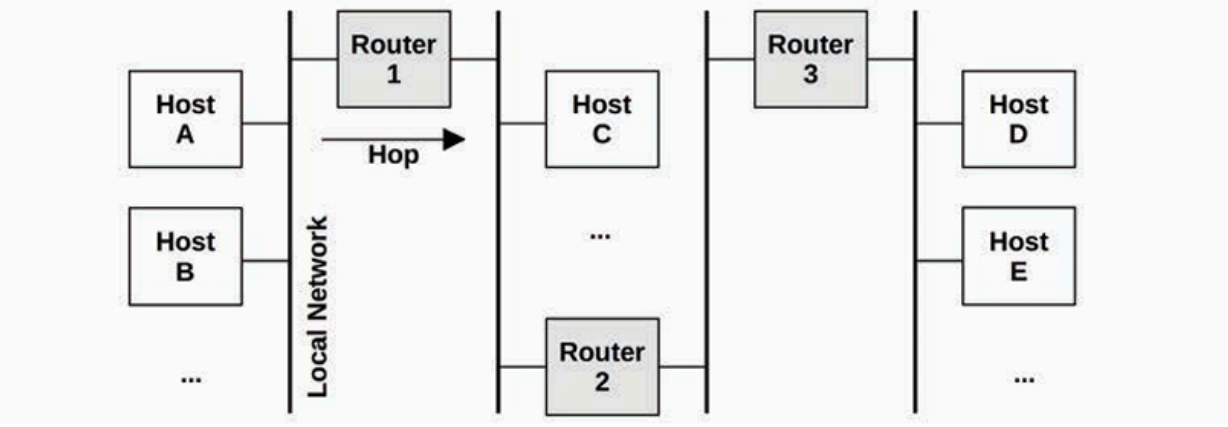
purpose. Two stack models are shown in Figure 10.3, with example protocols.







- At transport layer, data is datagram/segment
- At network layer, it is packet
- At data link, is frame
- Network is group of connected hosts. The internet is a group of networks instead of one big network. This helps with scalability
    - Some messages are broadcasted to all neighbors, and if internet were one net, this would flood
- Network components are shared, meaning if one router is bogged down that another needs, then all is slow



- Multicast - cast to multiple hosts at once; unicast - just between 2 hosts
- Information to route packets is in ip header

Figure 10.5 shows an example of encapsulation for a TCP/IP stack with Ethernet.

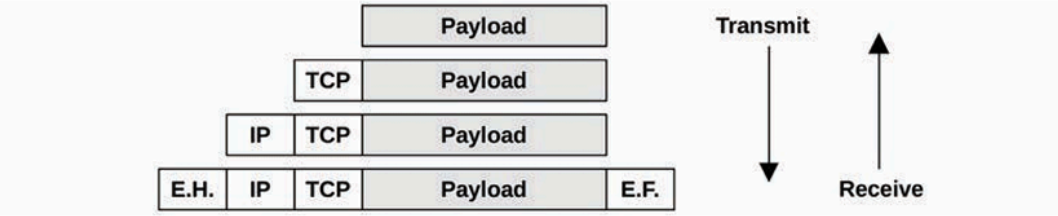


Figure 10.5 Network protocol encapsulation

E.H. is the Ethernet header, and E.F. is the optional Ethernet footer.

-
- Mtu - maximum transmissions size, 1500 bytes for ethernet. Now supports 9000+
    - Some old hardware can't handle. Can either break up or send error
- Tcp offload, large segment offload - offload this to nic
- Ping latency - time it takes for icmp echo to give response
- Connection latency - latency to form conenction before data is transferred
- First byte latency - time from established conenction to get first byte includes think time of server (backlog)
- Round trip time - time that request + reply packets spend in network (time round trip between endpoints)
- Connection life span - time conn start/end
- Larger buffers can mitigate higher RTT due to allowing to listen for more data before block off
- Localhost, loopback`
- Dscp, ecn, ip`
- Socket
- Is an abstraction in OS of network endpoint; is combination of IP and port number; used for network comms
- tcp
- tcp fast open, slow start, congestion/sliding window, selective ack, retransmit
    - Fast open allows you to send data with sync (faster send)
- Tcp timestamp - used to measure rtt
- Retransmission
- Retransmission happens upon duplicates
- sender sends packet num X
- Receiver sends ack and asks for packet num X + 1
- If packet is dropped, sender sends ack up to what it last got + 1 so sender knws what to send (ex: X + 1 even if got X + 2 and 3 already)
-
- Timer based retransmits (rto; retransmission timeout) - time before retransmit. Usually calculated dynamically based on connection specific RTT. first is at least 200 ms, and then exponential backoff makes later ones more lenient

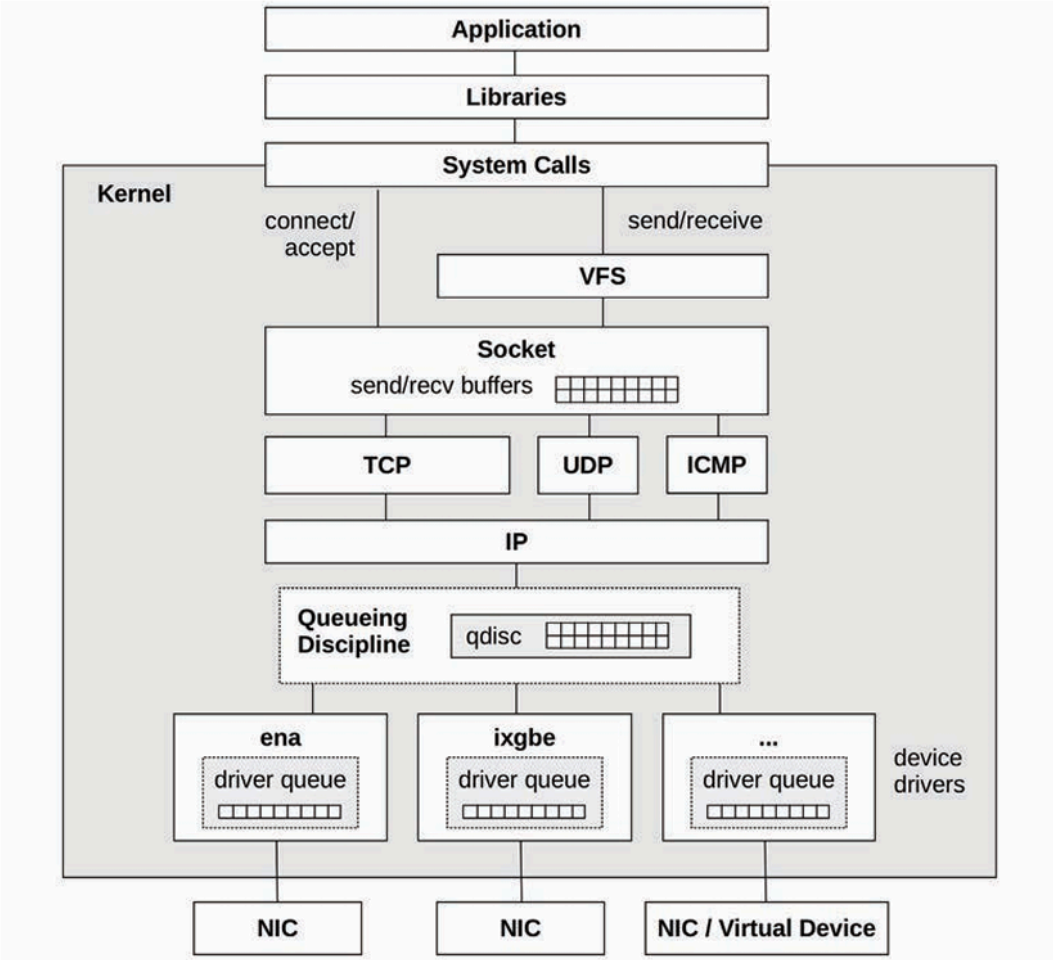
    - slow
- Fast retransmits - duplicates arrive. Once 3 (usually) arrive, then resend what just sent (X + 1 sent twice -> still expects x + 1)
    - For reno, cubic, resends whole sequence after 3 duplicates
    - Reack resends if later ack received and missing an earlier ack
- Newer algos for ack, fast trasnmit
- With regular acks, don't know which packets are missing
    - Could be missing 3 but have 4 and 5, but since ack given would be 4, sender would send 3,4,5 again
- Sack gives which ones it received exactly
    - Replaces nondescript ack
-
- Forward ack - more state information like outstanding info / unack ; better than sack
    - Replaces sack
-
- Rack replaces fast retrnasmit
    - Can be used with sack/fack, but rack is generally better (outperforms)
- Rack-tlp (recent ack with tail loss probe)
    - Tail loss probe:
        - Packets near end may be lost due to not having any future data to trigger duplicate acks. Therefore, must wait instead of fast transmit. This takes time
        - So this sends probe packet instead of waiting. If ack'd then is lost
        - Avoids long rto (retransmission timeout; wait before send again)
    -
    - Rack uses idea that packets originally had an order. This way don't have to wait for multiple duplicates. Just if have x, but not x - 1, and stuff not before x, after short delay, stuff before x is assumed to be missing too
        - If most recent ack is for 13, but not an earlier one, is likely lost
        - Ex: got 1,2,4 ack. 3 is therefore lost. After short delay, 3 is retrans
            - Better than waiting x time or 3 duplicate acks
        - waits a little bit before reteransmit
    - If too many are missing, just uses traditional duplicate ack retransmit
-
-
- Initial window + slow start
- Linux does 10 ckets (IW 10)
- For short flow, like http, window large enough to span data is good. Otherwise can result in congestion/apcket drops
-
- Slow start - once connection established, starts slow. Usually 1-2 mss
- Gradually increase exponentially
    - Oubles with each ack
- Once window reaches sufficient threshold, grows linearly

- If packet loss is detected, go into fast recovery and window is halved
    - This depends on algo used
- Congestion controls - has some retransmission too
- Depends on algorithm
- Algos
    - Reno
        - 3 duplicate acks -> halving both congestion window + slow start threshold
        - Also allows for fast retransmit + recovery
    - Tahoe
        - Retransmit upon 3 duplicate acks. No fast recovery. Resets congestion window to mss (minimum segment size) and enters slow start mode, starting transmission rate from beginning
    - Cubic
        - Cubic function to adjust window. More aggressive than reno
        - Hybrid start function for when to exist slow start phase
        - Is default for linux
    - Bottleneck Bandwidth and Round-trip propagation time
        - Uses RTT and available bandwidth to adjust window based on route; uses probing to adjust
        - Lost packets are secondary metric
        - Sensitive to accuracy issues
        - Estimates bandwidth and calculates optimal sending rate
            - Estimates the bottleneck bandwidth (BtlBw) and the network round-trip time (RTprop).
        - Makes bandwidth match BtlBw × RTprop
    - DCTCP (DataCenter TCP)
        - Uses ecn marks, makming it more dynamic instead of dropping packets (less latency)
            - Is specific bit in ip header
            - Instead of dropping packets in congestion, router marks this so receiver knows; upon detection, sends ecn echo
                - Router marks if queue reaches certain full threshold
            - Sender upon receivinng ecn echo reduces transmission rate
        - But ecn is not widely supported in internet
        - Adjsuts congestion window based on fraction of ecn packets received
        - Best for low latency, high throughput
- Delayed acks - delay acks by no more than 500 ms so that way can batch send
- Fast recovery
- Packet loss is detected. Instead of going back to slow start, temporarily reduces congestion window, then starts increasing linearly
    - Depends on algo
- After recovery, slowly increased
- Hardware
- Interfaces - receive frames. Higher bandwidth means high rcost

- Controllers - access physical interfaces. Driven by microprocessors and attached to system
- Switches and routers - contain buffers and microprocessors
    - Hardware lmitationsa affect bandwidth
    - Need buffers to handle difference in speed in links
        - High speed link vs low speed links
            - high speed is high bandwidth, low latency
                - Refers to hardware and is just a link in the network (a node)
- Driver
    - Network device driver or NIC usually splits up packets into frames
    - Kernel does ip + segmentation if not offloaded
- Blufferboat - holding too much info in buffer, causing high latency (due to delay and sending but not process due to big buffer)
    - Tcp small queues can help
- Checksum offload - offloads checksum verification to nic instead of cpu
- Segmentation offload - lets nic handle packet splitting instead of cpu
- Network interface
    - Has dma so can writ eto kernel memory without asking kernel
    - Tx ring buffer - circular transmit buffer; packets waiting to be sent by nic
        - Buffer holds transmit descriptors
            - Each descriptor has pointer pointing to memory of packet, packet length + status, flags (segmentation offload, checksum offload)
        - Actual packets are in memory
    - Rx ring buffer - circ receive buffer; packets received from nic before kernel processes them
        - This is a driver queue
    - Interrupt coalescing - wait for x packets or x time to pass before interrupting; this way kernel can read more from rx buffer before sending
    - Softirq used to place packets in socket buffers from kernel memory
    - Ways to avoid hardware interrupts from nic (too many and too many context switches with cpu)
        - Data plane development kit - uses poll mode drivers instead of interrupts to put in user space. Used for high throughput ,low latency
        - Xdp express data path - filter/drop/redirect packets before kernel networking processes them, acting as bouncer so network stack doesn't have to waste time on them
            - Runs at the earliest stage in the Linux stack (before iptables, TCP/IP processing).
            - Is still kernel based though so nrequires context switch
            - Can also reroute to other NICs or user space instead of default kernel. Depends on headers + programmed rules
        - Polling - network driver checks if nic has any new packets, as executed by cpu every so often

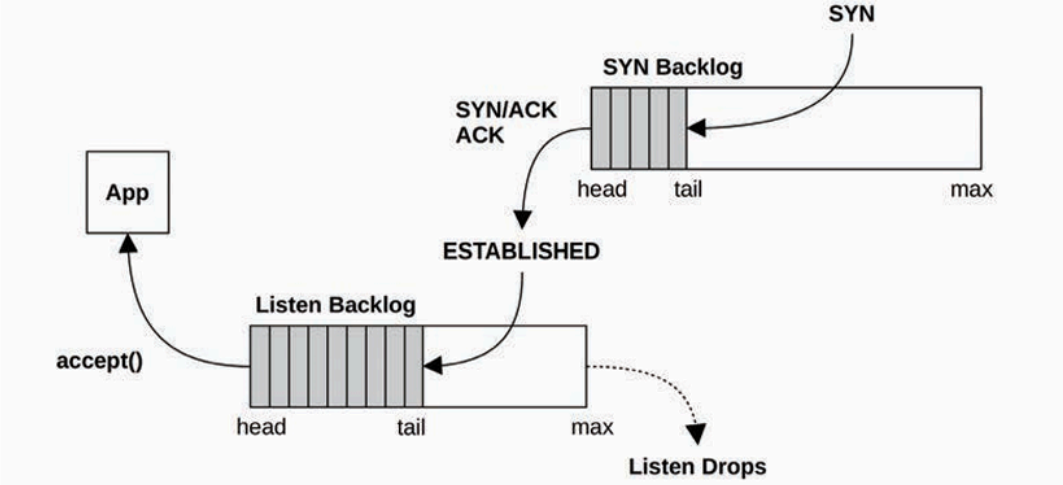
        - If polling interval is long, can cause delays
        - Wastes cpu cycles if there'snothing
    - Napi - polling + interrupts combined
        - Low traffic - uses interrupts (reduces polling overhead when there's nothing)
        - High traffic - uses polls (reduces context swtiches/cpu)
- Ways to avoid excess copying for transmission
    - Usually
        - Traditional Copying (Inefficient)
            - Mem of App -> kernel -> nic -> transmit
    - Use mmap to share buffer between apps and drivers. This way, reduces cpu usage by sending user pacekts directly to nic

- Network stack



-
-
- Generic block layer
    - Is general abstraction for disk scheduler, etc. that os interacts with

- Tcp queues
  - Syn queue - queue for processes waiting for syn-ack
    - acept() means put into receive queue
  - Receive queue - queue to receive packets
  - Transmit queue - queue to send packets
    - These 2 are managed by socket buffers skb
  - Queue for os for tcp
    - Syn flood - spam of syn from bogus ip so no one legit can connect



  - `
  - Send + receive buffering `how queue

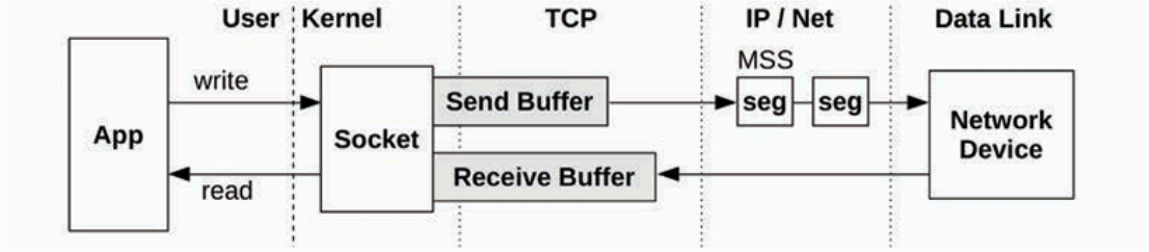
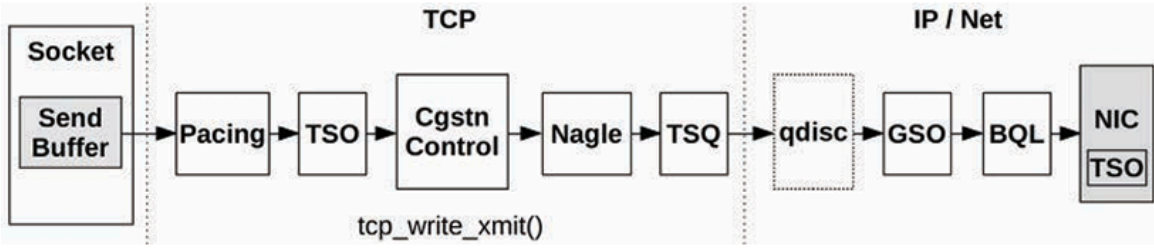


Figure 10.10 TCP send and receive buffers

  - `

xmit() kernel function).



- Ch 11
- Cloud
- Can be added onto/removed as needed
- Includes kuber, etc.

Here's a summary of the text:

The passage discusses cloud computing compared to traditional on-premises servers. On-premises infrastructure requires significant upfront costs, lengthy deployment times, and careful capacity planning to avoid over or under-provisioning.

Cloud computing offers key advantages:

- Server instances can be created and destroyed almost instantly
- Dynamic sizing allows automatic scaling based on actual demand
- Companies can grow from small to thousands of instances without detailed planning
- Pay-per-use billing (by hour, minute, or second) allows immediate cost savings when downsizing

The text highlights successful implementations, including Netflix's daily scaling of tens of thousands of instances and Shopify's reduction of idle time from 61% to 19%.

For storage, cloud provides:

- Instance storage (volatile/ephemeral)
- Persistent storage options: file stores, block stores, and object stores

Multitenancy challenges are addressed through resource management that provides performance isolation between tenants sharing physical resources.

The passage concludes by introducing Kubernetes, an orchestration system that:

- Manages application deployment using containers
- Groups containers into "Pods" with their own IP addresses
- Supports microservices architecture and auto-scaling

- Faces performance challenges in scheduling and networking
-
- vm

Vm appears as process (kvm) to main os

Here's a summary of the text on hardware virtualization:

The passage explains hardware virtualization, which creates virtual machines (VMs) that run complete operating systems. Key points include:

1. Hypervisor configurations:
    - Traditional classification divided hypervisors into Type 1 (runs directly on hardware) and Type 2 (runs on host OS)
    - Modern implementations blur these distinctions with two common configurations:
        - Config A (native/bare-metal): Hypervisor runs directly on processors
        - Config B: Hypervisor executes within a host OS kernel
2. Performance improvements over time:
    - Original implementations used binary translation with high overhead
    - Modern solutions use:
        - Processor virtualization support (AMD-V and Intel VT-x)
        - Paravirtualization for efficient resource use
        - Hardware device support like SR-IOV for direct hardware access
3. Major implementations:
    - VMware ESX, Xen, Hyper-V, KVM, and AWS Nitro
4. Performance overhead factors:
    - CPU: Generally minimal for application code
    - Guest exits: When guest VMs must transition to hypervisor control
    - Exit handlers: Functions that manage these transitions, affecting performance

The text highlights how virtualization technology has evolved to reduce overhead through hardware assistance and optimized implementations.

-
- Guest exit overhead - there’s cpu overhead when guest os in vm needs to finish event
    - kprobes and special tracepoints (like kvm:kvm_exit) help monitor these events
- Guest os maps virtual memory to what it thinks is physical memory (but it is just virtual)
    - The hypervisor (vm runner) is what actually maps to physical memory
- The I/O Challenge:
- Input/Output (I/O) means reading or writing data (like disk operations or network communication).

- In the past, every I/O request from a VM had to be translated by the hypervisor, which slowed things down.
- Modern Solutions:
- Paravirtualized Drivers: These allow VMs to work more directly with hardware, reducing the need for extra translation.
- PCI Pass-Through: This lets a VM use a physical device directly, speeding up data transfers.
- SR-IOV/MR-IOV: These technologies let multiple VMs share a single device efficiently.
- Resource control

Sharing the Physical Machine:

When many VMs (or “tenants”) run on one physical server, they share the same CPU and memory.

How It’s Managed:

- **vCPUs:** Each VM gets one or more virtual CPUs, and the hypervisor schedules their use.
- **CPU Scheduling:** Techniques like BVT, SEDF, or using Linux’s cgroup controls help ensure each VM gets a fair share of the CPU.
- **CPU Caches:** Special features (like Intel CAT) can split up the CPU cache so one VM doesn’t slow down another.
- **Memory Limits:** Each VM is given a fixed amount of memory. Tools like balloon drivers can adjust memory usage on the fly, but they need to be watched because they can sometimes cause performance issues.
- **Disk Management:** VMs use virtual disks, which the host system manages just like any other file system.
- Lightweight virtualiztaoin
- Have speed and efficiency of container but security + isolation of VM
- Emulate minimal features instead of whole pc

1. Container-Aware Performance Monitoring

What It Means:

When you run applications in containers (small, isolated environments), you want to monitor how well they're using resources like CPU, memory, disk, and network. But traditional tools often show the overall system’s data, not just what's happening inside a single container.

Key Points:

- **Block I/O Statistics:**
    - **Block I/O** refers to the data operations (reads and writes) on storage devices (like hard disks or SSDs).
    - **`blkio.throttle.io_serviced`:** This file, part of Linux control groups (cgroups), shows the number of disk operations a container has done.

    - In our example, the numbers didn't change over 10 seconds, meaning that during that period, the container wasn't doing any disk activity.
- **Container Awareness in Monitoring:**
  There are two main ways to view resource usage:

    - **Isolation Approach (A):**
      Each container sees only its own resource usage. For example, if a container is acting like its own server (common in public clouds), it should report only its own activity. This makes it harder to see if other containers (neighbors) are affecting performance.
    - **Visibility Approach (B):**
      The container can see overall host resource usage as well as its own. This is useful in environments like private clouds, where diagnosing issues (like "noisy neighbors" that use too many resources) is important.
- **Resource Control Awareness:**
  Monitoring tools should also show when resource limits (like disk I/O caps) are being enforced. This helps you understand if a container's performance is being throttled due to these limits.

- **Tracing Tools Challenges:**

    - **Tracing Tools** (such as those based on **perf**, **Ftrace**, or **BPF**) are used to collect detailed performance data.
    - **Isolation Needed (A):** If you want to restrict the data to just the container, the kernel (the core of the operating system) must filter events so that only container-specific information is available.
    - **No Isolation Needed (B):** Alternatively, the container might be allowed to see full system data from places like `/proc` or `/sys`, which can help in diagnosing broader issues.

---

2. Lightweight Virtualization

What It Means:
Lightweight virtualization combines the **security and isolation** of full hardware-based virtual machines (VMs) with the **speed and efficiency** of containers.

Key Points:

- **Concept:**
  Traditional VMs (like those managed by hypervisors such as QEMU for KVM) emulate a full computer system with many hardware features. This makes them secure and isolated, but also large and slow to start.
  Lightweight virtualization (often called microVMs) only emulates the minimal necessary

hardware, making them much faster to start and more efficient.

- **Comparison:**
    - **Full-machine Hypervisors (e.g., QEMU):**
      They support many hardware devices and legacy features. For example, QEMU's codebase is very large (around 1.4 million lines of code).
    - **Lightweight Hypervisors (e.g., Firecracker):**
      They focus on what is really needed for server applications. Amazon Firecracker, for example, has only about 50,000 lines of code, making it faster and more efficient.
- **Implementations:**
    - **Intel Clear Containers (2015):**
      Used Intel's Virtualization Technology to create lightweight VMs, later evolved into Kata Containers.
    - **Kata Containers (2017):**
      A project that combines Intel Clear Containers with another lightweight solution (Hyper.sh RunV).
    - **Google gVisor (2018):**
      Uses a special **user-space kernel** (a version of the operating system kernel running in user space) to provide security, which is different from traditional VMs.
    - **Amazon Firecracker (2019):**
      Uses KVM to run microVMs that boot in about 100 milliseconds, combining strong isolation with fast performance.
- Name spaces (containers) vs vm
- Vm has stronger isolation; is different os
    - Hyperviser supervises. Vm specific optimizations exist
- Namespace shares system os
- Namespace faster/less overhead, smaller + easier to port
- Namespace + containers allow for easy development + deployment + including dependencies
- Lightweight vms are in the middle

Hardware Virtualization Observability Summary

The text details observability options for virtualized systems, emphasizing that capabilities depend on the hypervisor type and observation location:

From Privileged Guest/Host Perspective

- All physical resources can be observed using standard OS tools
- Guest I/O can be monitored through I/O proxies
- Per-guest resource metrics are available from the hypervisor
- Guest internal processes cannot be observed directly
- I/O using pass-through or SR-IOV may be difficult to observe

Specific Hypervisor Tools

- **Xen**: Uses `xentop` to monitor domains, showing CPU usage, memory, network activity, and block device operations
- **KVM**: Guest instances appear as processes in the host, allowing tools like `top` and `pidstat` to monitor them
- Guest vCPU exits can be analyzed using `perf kvm stat` to identify performance bottlenecks

From Guest Perspective

- Only virtualized resources are visible
- Linux includes some virtualization-aware metrics like CPU "steal time"
- Kernel tracing tools generally work in hardware VMs but may not in containers
- Latency measurement is valuable for analyzing virtualized device performance

Analysis Strategy

- Physical system administrators can check for bottlenecks and resource control limits
- Guests should focus on analyzing virtual resource usage and latency patterns
- I/O pattern analysis can help identify contention issues
- Serious performance investigations may require coordination between host and guest administrators

The text emphasizes that observability is critical for diagnosing performance issues in virtualized environments, especially given the complexity of modern cloud infrastructures.

Misc

OS Virtualization Performance Summary

The text provides a comprehensive overview of container performance characteristics, resource controls, and observability challenges:

Performance Overhead

- **CPU**: Minimal direct overhead; processes run directly on CPUs with no additional costs from namespaces/cgroups
- **Memory**: Efficient usage with shared page cache between containers accessing the same files
- **I/O**: Some overhead from extra layers like overlayfs (filesystem) and container networking
- **Multi-tenant contention**: Major performance impact from shared kernel resources:
    - CPU cache pollution
    - TLB misses
    - Kernel lock contention
    - Interrupts from other tenants' devices

Resource Controls

- **CPU allocation**: Managed through cpusets (dedicated CPUs) or shares/bandwidth limits
- **Memory**: Controlled via hard limits, soft limits, and kernel memory limits
- **Storage**: File system quotas, swap limits
- **I/O**: Disk throughput/IOPS limits and weights
- **Network**: Traffic prioritization and bandwidth controls

Observability Challenges

- **From host**: Complete visibility of all processes, resources, and container activities
- **From container**: Limited visibility, often showing host-wide statistics instead of container-specific metrics
- **Tool limitations**: Many traditional monitoring tools lack container awareness
- **Resource control visibility**: Need special techniques to identify if containers are hitting software limits rather than hardware limits

The text emphasizes that containers face a paradoxical situation: they're more likely to encounter kernel contention issues while simultaneously removing the end user's ability to diagnose those issues through kernel-level observability tools.