1 CPU virtualization

Process : instance of a program in execution with memory, data, state

Operation System OS : software layer interfacing hardware resources and applications, virtual machine abstracting raw hardware; protection, isolation, sharing resources, set of common services; illusionist : private CPU, private memory

1.1 Processes

Program : passive, code & data, stored in file on disk, compiled to executable file (CPU instructions, data)

Process : active/alive, virtual concept, isolated from others, running a program creates a process, instance of an executable; unique ID (PID), memory image (code/data static, stack and heap dynamic), CPU context (registers, program counter, current operands, stack pointer), file descriptors (pointer to open files)

Thread : process can have multiple threads in same address space

Stack : temporary data (function params, local vars, return addresses), grows from high to low address

Heap : dynamic memory allocation, grows from low to high address

Data : statically known at compile time, global variables, data structures

Text : read-only text segment, code and constants

Create process : loading (code and data), memory allocation (heap and stack), initialization (tasks related to IO), ready (transfer CPU control at program entry point)

Time sharing : one task at a time, quickly switching among other tasks

Space sharing : task gets a portion of available space

CPU virtualization : process illusion of exclusive CPU access, CPU is a shared resource among all processes

OS scheduler : keep list of processes and their state (process control block), picks the process to run according to scheduling policy

Process state transition : Running (scheduled), Ready (descheduled), Blocked (IO)

Process control block : PID, process state/context, pointers to parent process, CPU context, pointer to address space, IO status info, pointer to list of open files

Process API : **fork** executes a child process (child different PID, copy of parent process), **exec** executes new program (replaces the memory, same PID), **exit** terminates process (parent exits before child > orphan process adopted by init (PID 1) > die), **wait** blocks current process until one of the child terminates (returns the child PID)

Shell : interactive user interface with the OS

1.2 Process abstraction and API

CPU virtualization : each process illusion of exclusive CPU access when it is shared resource to all processes

Trust : OS is trusted with full access to all hardware capabilities, all other programs are untrusted (restricted rights, disabled dangerous operations)

Limited direct execution : basic technique, execute program as fast as possible with some restrictions; problems : restricted operations (how not to execute privileged code), control process execution on CPU (how to pause/stop)

Hardware help : use protection rings to distinguish execution modes, ring 0 kernel mode (most privileged), ring 3 user mode (least privileged)

Requesting OS services : system call API, transfer execution to OS, meanwhile execution of process suspended

System calls : create/destroy processes, access file system, communicate to other processes, allocate memory; more than 300 in Linux

System calls execution : process executes special trap instruction (exceptions), saves registers to per-process stack, CPU jumps to kernel mode and raises privilege level to ring 0, when finished OS calls special return-from-trap instruction and lower privilege to ring 3, restore registers; handle internal program errors (division by zero, overflow, access not allowed to memory region), produced by the CPU while executing instructions, synchronous CPU invokes only after terminating the instruction

Configure at boot time : OS tells hardware specific handles to use when certain events occur (system call), trap table and entries, system call numbers (process specifies in stack or register)

Interrupts : asynchronous (signals to CPU external event), hardware provides signaling mechanisms for OS to regain control, suspend user process and switch to kernel mode (context switch), execute appropriate interrupt handling code, restores user process

OS control : OS configures timer interrupt, every certain of millisconds, OS can decide which process to execute; nested interrupts : hardware instructions to delay (disable) interrupts and re-enable when safe

1.3 Scheduling

OS Scheduler : stop running process and start another one, policy - how to pick the next, non-preemptive (switch if process blocked), preemptive (switch event event if can continue)

Context switch : kernel mode, cannot go back to same proc. (terminate proc., system call), do not want to go back to same proc. (proc. run for too long, other proc.s should be scheduled); stop running proc. to start another, store proc. state in Process Control Block PCB

Context switch procedure : save running proc. state, select next thread, restore execution state of next proc., passes control (return from trap)

Idle process : all proc.s are blocked, low priority idle proc., never blocks or exec. IO, always at least one process to run

Assumptions : each job runs for same amount of time, jobs arrive at same time ($T_a = 0$), once job started it will finish, all jobs only use CPU (no IO), run-time of jobs in known, single CPU

Utilization : fraction of time CPU executes a job (goal to max.)

Turnaround time : total time from job arrival T_a to job completion T_c (goal to min.), $T_{ta} = T_c - T_a$

Response time : how long it takes until job scheduled, firstrun time $T_f, T_r = T_f - T_a$

IO awareness : usually IO slow (milliseconds), scheduler incorporate IO, schedule another job during IO, better CPU utilization

First In First Out FIFO : non-preemptive, challenged by long running tasks (long jobs delay short jobs, long turnaround times), convoy effect (many short potential get queued behind one heavyweight requirement), bad response time

Shortest Job First SJF : non-preemptive, challenges come if you start a long job right before a short job come in (convoy effect), bad response time

Shortest time to completion first STCF : preemptive, when new job enters select the one with least time left, prioritize short jobs, bad response time

Round Robin RR : preemptive, run job for fixed interval (or time-slice) then go next in queue, responsiveness increases turnaround (for equally long tasks)

Multi-Level Feedback Queue MLFQ : preemptive, support general purpose scheduling, batch process (response time does not matter, cares for long run times), interactive process (critical response time, short bursts), first optimizes turnaround time (important for batch process), then mini. response time (better interactivity); use past behavior as futur predictor, multi. lvl. of RR, each level has higher priority, process at higher level scheduled first, high levels have short times slices, lower level run longer

MLFQ rules : run high priority first (if equal, run first), process start at top priority, if A uses full time slice lower its priority, periodically move jobs to topmost queue

2 Memory Virtualization

Address : location of a byte in memory, byte addressable

Address space : abstraction, set of addresses accessible to a program, static (code, globals) dynamic (stack, heap)

Dynamic memory : amount mem. required depend on program, input size unknown at compile time, recursive function calls

Stack : First In Last Out FILO, push (add elem.) pop (take elem. out), reversed of insertion order, one per thread/proc.

Invocation Frames : store local variables and context to run function (callee), compiler does it; on stack (simple, sequence of active parent frames), allocated in func. prologue + freed when return

Heap : randomly allocated mem. objects, statically unknown size or allocation patterns, lifetime/size unknown, `alloc` creates, `free` destroy

Straw man : pre-allocate mem. region (4KB), alloc returns start address of free region + increment used size; + simple, - no reuse & will run out of mem.

Free list : abstract head into list of free blocks, track free space, list available mem. obj. and size, alloc split free block and put remaining to free list, free add block to free list

Better implementation : alloc find fitting obj., first fit (first object in list), best fit (closest to size), worst fit (largest object); free merge adjacent blocks

OS interaction : OS give proc. large mem. region (sbrk, mmap syscalls to allocate mem.), runtime library manages mem. region (libc), allocators are critical for memory allocation (perf. reliability security)

Memory virtualization : enables isolation (requires separation), proc. prohibited to access mem/registers from others proc.

Uniprogramming : one program at a time on one machine, OS and program present in physical memory, no abstraction (physical addr.), no isolation

Multiprogramming : time-sharing OS, many program load mem. simultaneous; transparency (agnostic of physical mem./other mem. proc.), protection (not corrupt other proc.), efficiency (perf.s, no waste memory : fragmentation), sharing (proc. may share parts of mem.)

Virtual memory : virtual address space, address space of proc. starts at 0x0, map virt. addr. to physical addr.

Time sharing mem. : save mem. to disk in context switch; + better space saving, - bad perf.s (due to IO)

Static relocation : relocate program to assigned area, compile code with PC-relative addr., adjust pointers in code/globals section, only one addr. space, no physical/virtual space separation, - no separation & adjusted overhead & bugs in proc. crashes other proc./OS

Dynamic relocation : hardware mechanism, translate mem. addr. from program viewpoint to hardware view, interposition hardware intercepts mem. access dynamically translate from Virtual Address VA to Physical Address PA

Indirection : intermediate layer to access or manipulate data/resources, batching (group operations to amortize cost), caching (store data locally for faster access)

Memory Management Unit MMU : hardware translate VA to PA, OS configures MMU (ring 0), special instructions for config., exceptions results in trap switching to ring 0

Base register : simple MMU, translate VA to PA by adding offset, store offset in special "base" register (OS controlled), each proc. has diff. offset, not secure

Base and Bounds : keep two values, base (minimum addr.), bounds register sets virtual limit of address space (highest physical), CPU checks if in bounds (return physical addr.) or throw exception (seg. fault); + secure (isolate) & performant, - no mem. sharing & waste of physical mem. (pre-allocate) fragmentation

Fragmentation : inefficient use of storage space, internal fragmentation - allocated mem. larger than requested (intra-process), external fragmentation - total mem. space satisfy requested mem. but not contiguous (process wide)

MMU segmentation : one base and bound per memory area (registers : code segment CS, data segment DS, stack segment SS, user-defined ES FS GS), allow proc. several continuous mem. regions, OS segments independently anywhere in physical mem., minimize mem. waste, sharing (segmentation introduces protection bits, read write execute) - code segment read & execute, share code segment with other proc., reduced external fragmentation; issues : segment backed by physical memory, external fragmentation still happens (heap, stack large enough)

2.1 Paging

Paging : eliminate requirement of contiguous physical memory for allocation (+no external fragmentation, fast to allocate), divide address

space into fixed size chucks (pages);- internal fragmentation, space to store translation, additional memory redirection

Page : minimal unit of address space, virtual page (process context), physical mem. divided into array of fixed-sized page frames, each page fram contain single virtual memory page, pages should be small enough (mini. internal fragmentation) 1-16 KiB, virtual address contiguous (not physical), translation in page table

Virtual address : virtual page number (select the page amont 2^l pages) + offset (location in page among 2^l bytes)

Address translation : MMU, virtual page number HW translation to frame number, keep page offset

Page table : virtual-to-physical addr. translations (page table entries PTE, page frame number PFN), every process has one page table (in memory, managed by OS), pointer to page-table stored in special register (PTBR), saved restored by PCB in context switch; present bit (valid translation), protection bits (permissions), dirty bit (page modified), access/reference bit (tracks page popularity)

Linear page tables : use a lot of memory, need bigger pages (large internal fragmentation) or multi-level paging

Multi-level page tables : one or more indirection levels allow space efficient encoding, each level add memory lookup address translation

Translation look aside buffer TLB : cache of recent virtual addr. translations, TLB hit/miss, miss expensive, locality of reference helps, may become invalid (context switch)

Swapping : main memory is not enough for all processes, store unused pages on disk, OS can reclaim memory, over-provision (hand out more memory than physically available), present bit in MMU (if not trigger page fault, OS takes over)

3 Concurrency

Concurrency : managing multiple tasks with a single processing unit

Parallelism : performing multiple (part of a) tasks using multiple processing units

Threads : execution contexts expressing opportunities for concurrency (mask IO latency, prioritize threads) and parallelism (multiple CPU), share address space and data

Threads issue : threads interleaving - uncontrolled scheduling of threads on shared memory (non-deterministic behavior)

Race condition : timing or order of events affects the correctness of the program

Data race : when one thread accesses mutable variable whine another is writing to it without synchronization

Critical section : code portion that access shared region concurrently

Mutual exclusion : only one thread execute the critical section at any point (others wait)

Locks : mechanism used to ensure atomicity via mutual exclusion, one shared lock per critical section, one thread acquires (hold) the lock, others wait for release

Interruptible lock : turn off interrupts in critical section, no hardware interrupt, no scheduler, used in single-processor systems; + simple; - privileged operations, no support for multiple locks, single processor, starvation, lose hardware interrupts

(Faulty spin lock) : use shared variable, simple loads and stores; violates mutual exclusion property

Test-and-set lock : hardware supports atomicity, test the old value and simultaneously setting memory to a new value; unfair lock, starvation

Compare-and-swap lock : write a new value if the old one is the same as expected; same behavior as test-and-set

Spinning locks : busy-waiting problem (spin when waiting), waste CPU cycles; mutex - waiters (park) go to sleep and lock holder wakes up the waiters to release (waiting queue), ensure fairness

Condition variable CV : concurrency primitive, allows threads to wait for certain condition before proceeding; a thread waits on condition (waiting queue), another thread signals; API - wait(c) waits condition, signal(c) wake up one waiting thread, broadcast(c) wake up all

Semaphore : similar to condition variable, object with integer value; API - sem_init(s, v) initialize, sem_wait(s) wait for available slot, sem_post(s) increments slot by 1; semaphore with value 1 acts as spinlock, binary semaphore acts as mutex (value 0)

Atomicity violation bugs : sequence of operations intended to execute atomically are interrupted, unexpected states; sol. : use a common lock between threads

Order violation bugs : expected sequence of operations not followed, incorrect program behavior; sol. : use a CV to signal

Deadlock : two or more processes unable to proceed, each one waiting for other to release; sol. : impose total ordering + obtain all resources or nothing at once

4 Persistence

Bus : common single set of wires for communication among hardware devices

Canonical device : controller with device registers (status, cmd, data), device internals; CPU interacts with device controller (write device registers), device controller signals CPU through memory polling or interrupt

Device protocol : wait until device ready, set data and command, wait until command completed

Port-mapped IO : CPU uses designated IO ports, each device has assigned port, special instructions to communicate

Memory-mapped IO : CPU uses load/store instructions, hardware maps control registers to physical address space; high performance

Data granularity : single byte at a time (keyboard); whole blocks (disks, NICs)

Access pattern : sequential access (tape), random access (disks, CD) some overhead

IO interrupt : generates interrupt whenever it needs service; +handles unpredictable events well; -high overhead

Polling (spinning) : OS periodically checks a device status register; +low overhead, -waste CPU cycles

Coalescing batching : device waits for some time until more requests complete, batches all responses and send everything

Programmed IO : CPU tells device what data is, one instruction for each byte/word, efficient for few bytes, consumes CPU cycles proportional to data size

Direct Memory Access DMA : CPU tells device where data is, controller access to memory bus, transfer data to/from memory directly, efficient for large data transfers

Device drivers : specialized pieces of code in kernel interacts directly with device, standard internal interface, same kernel IO system call interact easily with different device drivers; top half accessed in call paths from system calls (open/close/read), bottom half communicates with the device; example of encapsulation (same API), OS implements support for API based on device class

Disk latency : seek time + rotation time + transfer time

Disk scheduling : how should OS schedule IO requests to minimize seek time, relative position between requested and disk head matters more than length of transfer

Redundant Array of Inexpensive Disks 0 RAID : file is striped across disks, no redundancy of data (no fault tolerance), best performance (cumulative bandwidth utilization), total capacity is sum of capacities, no data security, reduced reliability (more disks > higher prob. of fail)

RAID 1 : Duplicate file blocks, deals with disk loss, does not handle corruption, total capacity is capacity of 1 disk, reads can be parallelized, writes are equivalent to one disk, expensive, critical infrastructure (sensitive information)

RAID 5 : parity (fault tolerance), if one disk fails, one can reconstruct its data by XOR-ing all remaining drives, reliable, fast (very fast reads), writes become complicated, affordable, data-center environments

RAID combinations : RAID 01 - two disk on RAID 0 that are mirrored RAID 1; RAID 10 - two mirrored disk on RAID 1 that are used with RAID 0

4.1 File System

Purpose : given set of persistent blocks, manage these blocks efficiently (non-volatile storage, organize files metadata and permissions)

IO abstraction stack : layered abstractions - cache blocks (recently read from disks), buffers recently read, Block device interface single interface to many devices, data read/write fixed size blocks, Device Driver translate OS abstractions to hw specific IO device details, MM IO DMA interrupts controls registers

File system abstraction : need for long-term information storage (outlive program), support concurrent accesses, persistent named data, files & directories

File : named collection of related information recorded in secondary storage, linear persistent array of bytes; data, user or application puts in it; metadata information added and managed by the OS, OS/FS does not care or understand content

File abstraction : inode and device number (persistent ID), file name (human readable), file descriptor (process view)

OS view : inode - Low-level unique (per FS not global) ID assigned to file by FS, metadata of file, each file has exactly one associated inode, inodes recycled after deletion, multiple file names can map to same inode (hard links)

Inode table : storage space split between inode table & data storage, files statically allocated, require inode number to access file content

File name : URL, local, remote, directory; untyped files (array of bytes)

Directory : special file stores path to inode mapping, mark if a file maps to regular file; links - file pointers (don't contain data) reference another file, hardlink - maps file path to file inode number (mirror copy of original file, same inode number as original file), symbolic (soft) link - maps file path to different file path (actual link to original file, new inode number allocated)

File descriptor : do expensive tree traversal once, store final inode/device number in per-process table

Permissions : bits - rwx, owner group, others

File system API : create - open path flags permissions (returns file descriptor fd), close - close fd, read - read fd *buff count, write - write fd *buff count (return # bytes r/w), manage file offset - lseek fd offset whence (repositions file offset), unlink delete - unlink pathname, synch write - fsync fd, get metadata - fstat fd *statbuf

Multiple file systems : FS mapped anywhere into a single tree, any directory can be mount point; mount allows multiple FS on multiple volumes form a single logical hierarchy

FS implementation : given large set of N blocks, need data structures to encode file hierarchy and metadata; overhead (metadata vs data size) should be low, internal fragmentation should be low, efficient access of content (external fragmentation, #metadata access), API

File system layout : FS stored on disks, divided into partitions, sector 0 of disk master boot record MBR (bootstrap code, partition table), first block of each partition has boot block

FS superblock : one per FS, metadata about FS, # inodes, # data blocks, where inode table begins, info. to manage free inodes/data blocks, read first on mount

Contiguous file allocation : all data blocks of each file allocated contiguously; + simple, efficient; - fragmentation (large external frag.), usability (know file size on craetion)

Linked blocks file allocation : each file is linked list of blocks, first word of each block points to next block (rest is data); + space utilization, simple; - performance (random access slow), implementation (mix data and metadata), overhead

File allocation table FAT : decouple data and metadata, linked list information in single table (all pointers in central table); + space utilization (no external fragmentation), simple; - performance, overhead (limited metadata)

Multi-level indexed : tree of pointers; file is fixed asymmetric tree, fixed sized data blocks (4KB), root of tree is file's inode (metadata, set of 15 pointers, first 12 to data blocks, 13 to block containing pointers to data block, 14 double indirect, 15 tripple indirect); efficient in finding blocks, efficient in sequential reads, simple to implement, supports large files and small files (small overhead); + space utilization (no external fragmentation), simple, performance (low seek cost); - overhead (low metadata overhead extra reads indirect

access)

Directories : special files, set of data blocks mapping file names to inode number; root directory has inode number 2

Performance : number of I/O, speed of I/O, impact on one program, impact on all programs; latency, throughput, I/O operations per sec IOPS

Improve performance : caching - avoid unnecessary operations; batching - group operations to increase throughput (perhaps at expense of latency), delay idempotent operations; add a level of indirection - convenient abstraction

File system buffer cache : block cache, OS reads block of inode many times; map inode, block offset to page frame number, read returns without performing disk I/O; may use all unused memory

Batching : perform fewer operations with larger transfers, possible for consecutive blocks belonging to same inode (disk fragmentation metric)

Delaying : delay write operations, perform them asynchronously, reorder operations to maximize throughput; - content lost if OS crashes; write-back caches (delay writes, - potential inconsistency), write through caches (write synchronously, - slow)

Crash consistency : atomically move FS state from one consistent state to another

File system checker FSCK : after certain number of mount operations or after crash, check consistency; hundreds of consistency checks (superblock, FS size, link count equal to directory entries); slow and can take hours (scan full disk)

Journaling : limit the amount of work required after crash, get correct state (not just consistent state); multiple disk updates into single disk write, write ahead (short note to a log "journal" with changes about to be made to FS data structures), if crash occurs, consult the log; no need to scan the entire disk

Transactions : set of actions grouped together; Atomic (all or nothing happen), Consistent (from one correct state to another), Isolated (transactions do not interfere), Durable (on completion, effects persistent)

Log structured FS : use entire disk as a log, buffer all updates (including metadata) in-memory, when segment is full, write to disk in long sequential transfer to unused part of disk, never overwrite existing data