

OS Overview / Processes

Services of the OS	- Protect: isolates running program - Facilitate: provide resources
Problems do OS solve	- manage hardware - allocate resource to running programs - controls access to resources
Types of OS	- Library OS: library of services (small) - Multitasking OS: multiple processes existing - Multi-user: protection for diff users
What does kernel do	Part of OS that responds to system calls, interrupts and exceptions - System Call: how a user program interacts with OS - Interrupts and exceptions: how hardware interacts with OS
User Space Vs Kernel Space	All programs that run outside the kernel. (All Program) Kernel space is the region of memory where kernel runs
Types of kernel	- Monolithic: Entire OS in kernel - Microkernel: Only absolutely necessary are in kernel - Hybrid: in between
Whats a process	An environment which application runs, is isolated from others.
Kernel Mode vs User Mode	Kernel Mode access privileged features, allows to protect itself and isolate processes from each other
Caller-save vs callee save	Caller-save not preserved through subroutine calls callee-save preserved across subroutine calls
Trap Frame VS Switch Frame	Trap frame contains all general purpose registers and status registers, used during interrupt or exception Switch frame just contains callee-save registers of switchstack, used only during context switch
What do thread shares and not	thread shares code, global variable, heap and file descriptors but has its own stack, register values
Types of Concurrency	- Timesharing: appear to run at the same time by rapidly switching - Parallelism: run at the same time because of hardware
Define context switch, yielding, blocking, preemption	Context Switch - switching one thread to another Yielding - thread voluntarily stops running Blocking: waiting for I/O or resource Preemption - involuntarily stops running

Scheduling Quantum	In timesharing, each thread gets a small amount of time (SQ)
State of threads	Ready -> running (dispatch) Running -> ready (Preempt), -> blocked (waiting for IO) Blocked -> ready (io complete)
User Threads vs Kernel Threads	User Threads - user manage Kernel threads - kernel manage
Diff Techniques	- User-level threads, deadlock - Kernel threads (1 : 1), use less system resource than creating process but expensive as each is a syscall - n:m threading, hard to match to maximize parallelism

Synchronization

Define Critical Section, synchronization, race condition	Critical Section - part of concurrent program where shared object accessed Synchronization - coordination of access Race condition - program results dependent on order of execution
Cache Line	unit of data transfer between main memory and core's cache
Cache Coherence	Ensuring that in a multicore or multiprocessor environment changes in one cache must be propagated to the other
Sequential Consistency	1. Maintain program order on individual processor cores: If one memory operation appears before another, first must complete first 2. Ensure write atomicity: all subsequent reads will see same result
Write Buffer	Small and fast memory that holds data to be written to RAM
Sequential Consistency impedes what	- Impedes hardware optimization: Make cache coherence, write buffers more expensive - Impedes compiler optimization: software pipelining, code motion, loop blocking
Atomic instruction	An instruction that cannot be interrupted
Data Race Desired Solution	- Mutual Exclusion (Only one thread at crit sect) - Progress: If no thread is currently in crit sect, one of the thread will eventually get in - Bounded waiting: Once a thread tryign to enter, there is a bound
Semaphore	Used to enforce mutual exclusion requirements 1. Binary: Semaphore with a single resource, like a mutex but dont track ownership 2. Counting: Semaphore with arbitrary # of resource

Mutex VS Sempahore	- Semaphore can start with count 0 - Thread will wait until semaphore count is greater than 0 - Semaphore no owner - Sem_post doesnt have to follow sem_wait
Conditional Variables	Busy waiting is inefficient so more efficient to inform scheduler using a CV
Lockset Algorithm	For each memory location keep a lockset Keep a vector If lockset becomes all zero, no mutex in memory loc
Spinlock	repeatedly test the lock availability in a loop until the lock is acquired. Uses busy-waiting. Interrupts are disabled on the processor that holds the spinlock. <pre> Spinlock_Lock() { Critical_Enter() while (atomic_swap(&lock->lock, 1) == 1) lock->cpu = CPU } Spinlock_Unlock() { ASSERT(lock->cpu == CPU()) atomic_set(&lock->lock, 0) Critical_exit } </pre>
Issues when multithread processes share memory resource	1. Memory Consistency: Out of order reads 2. Cache Coherence: There are stale copies of cache Cache is divided into slices called cache line
Cache Line states	1. Shared (1+ have copy) 2. Modified (Dirty) 3. Invalid (Marked as stale)
Deadlock	1. Limited access 2. No preemption of resource 3. Multiple independent request 4. Circularity in graph of requests
Semaphore Impl	<pre> Semaphore_Wait(Semaphore *sem) { Spinlock_Lock(&sem->sem_lock) while (sem->sem_count == 0) { WaitChannel_Lock(sem->semwchan) Spinlock_Unlock(&sem->sem_lock) WaitChannel_Sleep(sem->sem_wchan) Spinlock_Lock(&sem->semlock) } --sem_count Spinlock_unlock } </pre>

	<pre>Semaphore_Post(Sem) { Spinlock_Lock(&sem->semlock) ++sem_count WaitChannel_wake(sem_wchan) Spinlock_unlock(semlock) }</pre>

VMEM

Two types of addresses	Physical - provided by hardware Virtual - provided by kernel to processes Kernel's job: - The hardware MMU translates the addresses. The kernel's job is to manage the mapping rules (page tables) that make that translation possible - part cop - protect one process address space from another - hide physical addr space - Kernel stays asleep during normal memory access. It is only woken up via a Page Fault when the hardware runs into a problem it cannot solve on its own (like loading data from disk or catching security violations)
Load time linking Issues	Linker patches addresses of symbols. - cannot enforce protection - how to move once in memory (must adjust all ptrs) - What if no contiguous region fits
Base + Bounds Tradeoffs	two privileged registers : base, bound $\text{physical address} = \text{virtual} + \text{base} \ \&\& \ 0 \leq \text{virtual} < \text{bound}$ Advantages: Inexpensive hardware only 2 registers, add and compare is cheap Disadv: - Growing process expensive - NO way of sharing
Whats the MMU	Memory Management Unit - One per core, translates virtual to physical addr

Segmentation	Process make base and bound registers for text, data, stack addr space built up of segments Each process has own segmentation table, vaddr divided into segment number and offset Advantages: - Allow multiple segments per process - Allows sharing - Don't need to store entire process in memory Disadvantage: - Require addr translation hardware - need n-byte segment of physical memory - Fragmentation
Fragmentation	inability to use free memory because its in too small spaces 1. Internal fragmentation a. Memory is in fixed-size slices, no unused space between the spaces but unused space inside them 2. External fragmentation a. when memory is allocated in variable sized slices, small holes between the slices as memory gets freed then filled by successive requests
Paging	Divide memory into small fixed-sized slices pages in physical mem is called frames, virtual is virtual pages each address has page offset and page number kernel map virtual page number to physical using page table each process has page table and each row is called PTE
Process of lookup page in MMU	1. Determine page number and offset of vaddr 2. look up PTE in current page 3. check PTE is valid 4. combine frame number with offset to get physical
Single level page table can be very large how fix?	Split page table into multiple levels MMU page table base register point to level 1 table address is p₁p₂...p_n offset index I₁ with p₁ ... index I_n for p_n Goal is to reduce size of individual page tables
Difference Hardware vs software managed MMU	- Software: - Manages MMU on addr space switches (context switch) - Creates and manages page table

	- manages physical memory - handles exception - Hardware: - Translate vaddr to phys - raise exception if necessary
Address translation using a page table adds at least 1 memory instr for execution, how fix?	Use a TLB, small fast cache to translate in MMU fault handler must be very eff Software managed TLB: if (hit) return f else find frame; add frame; return frame Hardware managed: - MMU handles TLB miss and page table lookup, replacing - If page fault, generates excep
Hardware Manged VS Software Managed TLB / MMU	TLB: - Hardware managed: Handles miss, automatically walks page table - Faster miss handling, predictable performance - difficult implementation, increase silicon area - Software: TLB miss triggers a trap (exception). OS walks page table retrieves mapping and updates - Simple, easier to upgrade - slower, unpredictable performance MMU: - Hardware: Works independently to handle addr translation and page table lookup - Fast Efficient - Hardware complexity, less flexible - Software: OS handle translation
Kernel Paging	Kernel is unpagged, it uses different translation method. Mapped to a high VM. Has exclusive access in the range 0x8000000
Types of Paging	- Demand paging: only load if needed - BSS page allocation (dont need to know proly) - Copy on write: Share reference rather than copy, if write happens make copy
What additional info need to be tracked to use secondary storage	Track present bit to determine if in memory Virtual pages in physical memory is the resident set
Page fault	accessing a page not in memory
How to reduce page fault	- be smart with replacement policy - 80/20 rule: 20% of mem gets 80% of access - prefetch pages to hide latency What to fetch? - Fetch surrounding pages if process exhibits spacial locality

What to do if there are more pages in L2 cache than there are TLB entries	have two TLBs
Naive Paging	Uses 2 IOs per page fault, swapping in and out Can reduce I/Os on critical path by keeping a pool of free frames
Whats thrashing? and why does it occur	When an application constantly swapping pages in and out, preventing much progress to be made. Caused by: processes requiring more memory than the system has - memory access patterns dont exhibit temporal locality (80/20 rule dont apply) - Each process fits individually but too many process - Hot memory doesnt fit in physical memory
Two types of page allocation policy	Local: Allocates within process using 2 steps - determining how much memory process use - Use page fault frequency plot, ensure within threshold - then separate policy to determine what to evict Global: Does not consider page ownership, doesnt protect against memory hogs
Working Set Model	According to 80/20 rule, heavily used is called working set set of pages in mem is called resident set
Eviction Policies	1. FIFO: evict oldest 2. Optimal Page Replacement: Theoretical 3. LRU: a. Stamp PTEs with timer value (takes time to find oldest) b. Keep doubly LL (costly) c. Approximate LRU 4. Clock Algorithm (FIFO but skip accessed pages) has clock hand: - if bit = 1: set to 0 skip else evict Memory could be an issue as pages are referenced in long intervals so add second clock hand to evict A = 0

File Systems

What are the layers of the	1. Local File System (Higher level APIs programmer
----------------------------	--

file system	sees) 2. Virtual file system (Abstract of lower level, presents different underlying files systems (HDD, SSD) to user as one) 3. Physical file system (How files are stored on physical media)
Hard Link vs Soft Link	Hard Link: Direct reference to physical data, allow multiple filenames to point to the same file. Creates an association between file name and i-number (Requires referential integrity, if link exists, then file refers to it must also) Symbolic / Soft Link: Pointer to another directory. Association between file name and path name
What does mounting do	Mounting creates a single namespace that combines the two namespaces of the file system. The new namespace is temp and only exists until FS is unmounted.
What needs to be stored persistently and not	Persistent - File data, metadata, system meta-data, dirs and links Non-Persistent - Per process: Descriptor Tables (tracks which file descriptor has open, which file it refers, position of file) - System ide: - Open File Table: files that are currently open - INode Cache: (in memory copies of inodes) - Block Cache: (in memory copies of data/indir blocks)
How are RAM and drives accessible	RAM - byte addressable Drives - Sector addressable
Metadata block of FS	inodes: each inode contain metadata and to map files to their data blocks. File i-number = index to inode table i: bitmap to store which inode in use d: to store which data block in use Superblock: contains metadata about entire FS
Since inode is fixed size how can it point to all data blocks of the file?	Smaller File: Pointers in inode sufficient (Direct block pointers) Larger Files: Indirect block pointers - Points to a block of full direct pointers
When we write to a block when do we need a read	Partial block: must read first Entire block: No read required
How to track file's data?	- Extent-based (Allocate files like segmented memory) - When creating prespecify length and loc - Simple, fast but fragmentation - Linked Files (Each block has ptr to next) - Easy dynamic growth, no frag

	- Non continuous so poor access times, poor non-seq access, pointers take up room - FAT (LinkedFiles but links in a file allocation table) - Reliable by creating copies of FAT - Bootstrapping by setting root at predetermine - Index Files: Same way as multilevel page tables (direct ptr, indirect, double indirect)
Cwd	Each processes file system context
Give scenario where cross allocation occurs	When truncating part of one file then appending it to another file, then crash - New file reuse block from truncated file - inode of truncated file may not be updated before crash resulting in cross-allocation
Ordering of Updates	Consistent order - Write to inode before creating dir entry - Remove dir name before deallocating - Clear inode before updating free map
To avoid crashing after "bad" subsets of writes. What rules in ordering updates to follow?	1. Never write a pointer before initializing the structure that it points to a. Never write directory entry before init inode 2. Never reuse a resource before nullifying ALL pointers to it a. Never mark inode as free in bitmap before removing it from dir 3. Never clear the last ptr to live resource before setting the new one a. When moving file from dir to dir, never remove it from first before adding it to other
Deadlock due to rules: how fix?	Soft updates. We write blocks in any order but keep track of dependencies. When writing a block, temporarily roll back any changes you cannot yet commit to disk. Structure of each updated field: - Old Value - New Value - List of updates this update depends on
What dependencies are better handled by postponing rather than rolling back updates	When freeing a block (file truncated), postpone marking the block free in bitmap until after block ptr is cleared.
What are some operations that require soft updates that occurs when creating file	- Link Addition (Directory entry -> bitmap, inode) - Write inode and bitmap to disk before directory entry - Link removal: (Bitmap, inode -> directory entry)

	- Remove dir entry before inode, bitmap - Block deallocation: (Free block bitmap -> blockptr) - Write the cleared block pointer to disk before freeing block bitmap - Block allocation: (block ptr -> free block bitmap, data block) - write data to block and updating bitmap before pointer in inode
Optimizing fsck	Splitting it into foreground and background - Foreground completed before mounting file system - Ensure FS info make sense - Recompute free block / inode counts from bitmap - Background does traditional fsck operations - Reclaim free space - Applications can use FS while this is happening
Problematic writes to metadata.	Write-ahead log to journal metadata. - Preserve portion of disk for log - write metadata op first to log then make change on disk - If crash, replay the log - Can group multiple ops into one log entry (Clear directory entry, clearinode, update free map) Performance: log is consecutive, multiple ops into one save disk bandwidth Challenge: find oldest relevant log but can use checkpoints; find end of log (uses circular buffer)

Devices

What's a bus and what are the two types of bus?	Communication pathway between components of computer - Internal bus: Between processor and main memory, fast - Peripheral: Allow devices in computer to communicate
Three types of device registers	- status: tells about device current state - command: issue command to device by writing val to register - data: used to transfer large blocks of data to/from device or combinations: status and command, data buffer
If write in progress and another write attempted what happens	UB

Two methods of interacting with devices	- Polling: Kernel driver repeatedly checks device status (use processor time) - Interrupts: Kernel does not wait for the device to complete command. Device request completions is taken by interrupt handler
How can device driver access device registers	- Port-Mapped I/O (device registers have port numbers) - Memory-mapped I/O (using LW, SW to device) - Program controlled I/O (Driver move data between memory and buffer, use processor to transfer data) - DMA (block data transfer between devices)
How data transfers between devices	1. Processor initiates data transfer (issues DMA request to disk controller), giving it starting addr and # of data 2. Controller directs data transfer 3. Controller interrupts processor when transfer complete
components of hard drive	1. Stack of platters (disk) that spins (Each has 2 surfaces to store data). Each surface is broken into: a. Track: single concentric circle b. Cylinder: collection of tracks that all in the same relative location 2. Disk arm (actuator arm) rotates around a pivot, has read and write head (Sense and record data along one cylinder at a time)
Two methods of HDD	- Cylinder head sectors CHS - location of each block specified by cylinder, head (surface), sector - all cylinders have same sectors - Logical block addressing - inner cylinders have less blocks than outer ones - disk moves arm to specific cylinder and keeps it there
Seek phases	1. Speedup (accelerate arm to max speed until half way point) 2. Coast (at max speeds (for long seeks) 3. Slowdown (stops arm near destination) 4. Settle (adjust head to desired track) Disk controller keeps a table of pivot motor table. It maps seek distance : power and time needed to move there. Table is set by periodic thermal recalibration avg seek time = $\frac{1}{3}$ of max seek
Disk controller properties	maps logical location to physical locations - Zoning: 2.7x more blocks on outside cylinders - Track skewing: sector 0 pos varies by cylinder to improve sequential access time - Sparing: flawed blocks remapped to new locations
Cost Model for Disk I/O	1. Seek time: time to move arm to cylinder

	2. Rotational latency: time it takes for desired block to get read write head 3. transfer time, data to be transferred to r/w head
Formulas	Bytes / track = surface capacity / num tracks Bytes per sector = bytesPerTrack / num sectors per track maxRotLatency = 60s/min / RPM avg seek = maxSeek / 3 average rot latency = (maxRotLatency - 0) / 2 transfer time = maxLatency / num Sectors per track
Disk Scheduling techniques	- First come first serve: process disk requests in order they are received - Pros: easy to implement, good fairness - Cons: cannot exploit locality to minimize seek time; increase latency, decrease throughput - Shortest positioning time first: always pick request with shortest seek time - Pros: exploit locality - Cons: Starvation, blocks far away from request never get processed; System doesn't know which request is fastest - Aged SPTF: Give older request older priority (better) - Elevator Scheduling (SCAN): Sweep across disk and service all request as you approach that track - Pros: Take advantage of locality, provides bounded waiting - Cons: Tracks in middle get better service as they are serviced in both directions, might miss some locality in SPTF - C-SCAN: sweep in one dir (better)
Flash Memory. Level that read/write occur at and overwriting/delete occur at	- Read or write occur at page level - Overwriting and deleting done in block level
How writing and deleting works in flash memory	slow: - read whole block into RAM - reinit block - update block in RAM and store to SSD fast: - mark page to be deleted as invalid - write to unused page - update translation table - requires garbage collection
SSD Limitations	- Each block of SSD has limited number of write cycles before read-only - SSD controller performs wear leveling, distributes writes evenly across the blocks so that block wear down at an even rate
types of flash memory	- single level cell

	- multi level cell - triple level cell - quad level cell each cell holds a charge. More levels = cost/GB cheaper but slower access to data and wears out more.

Scheduling

Performance metric of scheduler	1. response time ($s_i - a_i$) arrival to start 2. turnaround time ($f_i - a_i$) arrival to finish
Strategies	- First Come First Serve - Round Robin: (Preemptive FCFS), each job runs for 2 units of time - Shortest Job First, minimizes average turnaround time but starvation may occur - Shortest Remaining Time First (Preemptive SJF), select job with shortest remaining time and arriving jobs that are shorter
Processor Scheduler Objective	- Responsive: Thread gets to run regularly - Fairness: Sharing of processor, each thread makes progress - Efficiency: Account for the fact that there is cost to switch
Approaches to schedule with priority	1. Schedule the highest priority thread 2. Weighted fair sharing ($p_i / \sum p_j$)
What's Multi-Level Feedback Queues (MLFQ)?	Key Idea: Give higher priority to threads that block frequently (interactive ones) so they run whenever ready n round-robin ready queues with priorities on each. Higher level = higher priority. Higher level = smaller quantum. - Preempted thread put at the back of next low priority queue - When thread wakes after blocking, put at highest priority queue Interactive threads block frequently so stay in higher priority and vice versa.
If higher level threads are never empty, then do threads in	To prevent starvation, all threads are periodically placed in higher priority queue

--	--