About Scheduling: Goals: Max Throughput, Min delay, mean time to completion, mean response time | TRAPS come from CPU, INTERRUPTS from devices external to CPU
Timer interrupt: after some time, context switch to a new process
Turnaround Time:Tcompletion-Tarrival (From submission to end)
Response Time:Tfirstrun-Tarrival - Interval between first request& first response
Non-Preemptive Scheduling Pros & Cons (run 1 process to end)
Low scheduling overhead, high throughput,easy. But bad response time, bugs can freeze machine, not fair.
Preemptive Scheduling Pros & Cons: (stop 1 process to another)
Good response time, fair, good for real-time/priority scheduling
Complex, possible bad throughput, high-overhead
Real Time Scheduling – Critical operations must happen on time
Batch: Only care about maximizing throughput
Time sharing: care about fast response times to interactive programs and equal share of the CPU 1.FIFO, long process makes avg turnaround time high non-preemp
2.SJF, different process come diff times, non-preemptive
3.PSJF,preemptive shortest job first bad response good turnaround
4.Round Robin, runs a job for a time slice aiming for equal CPU/delays/scheduling. Wait time better than FIFO more expensive context switches but all processes get a chance.
5.Oracle, all knowing to know how to process in advance
6.MLFQ, uses multiple ready queues and feedback to determine priority. If P(A) > P(B), run A. If equal, run Round Robin. When a job enters, highest priority, if uses all its time reduce priority. After some time move all jobs to top of queue.
Benefits: Good performance for short interactive jobs, fair for long intensive workloads. Predictable real time response.
How does Trap work? Interrupt -> Look up trap in trap table that's loaded on start -> execute subroutine pushing PC/PS on stack -> return to user
Number associated for each interrupt.1st lvl handler sets up execution for kernel will pick 2nd level handler that actually deals with the interrupt and restores things.
About Process? Processes have states running, ready, and blocked that are used to coordinate. The OS has a process list to hold info about all processes in the system and Process Descriptor (or PCB) which holds info about a process such as state.
Types of Processes? Orphan – when a parent dies before child.
Zombie -A process that ended but has entry in process table.
Why Copy-on-write? If there's a big data area, a copy is expensive. Set up this mode and if the resource is about to be modified, copy it & write to it. Efficient.
What are some Process API's? Each process has a PID. Fork is a system call that creates a new process almost identical, but it has its own registers/PC/PID. Non-deterministic for which one runs unless we use waitpid to synchronize but waiting for child to finish. Exec, load's new code/info for new program after forking.
Memory Mechanisms
Internal Fragmentation: wasted space inside fixed size blocks, Average waste 50%.
Splitting: find a free chunk of memory for the request, split it in two where the 1st goes to caller, 2nd remains on list new start region.
Header: kept in memory containing size of allocated region. When searching for free chunk, it is size of N + size of Header
Strategies/Policies for Memory:
Fixed Partition Allocation: Pre-allocated partitions that are fixed for processes, but not good for sharing memory. Internal fragmentation.
Dynamic Partitions: variable sized chunks but can relocate & expand, subject to external fragmentation.
Best fit: search smallest that's biggest |Worst fit: find largest chunk possible |First Fit: First chunk you find that is big enough, fragments | Next Fit: Pros of first & worst
Buffer Pools: For popular sizes have special pools to manage that size
How does a resource return to free pool? The client does by calling free or close, etc. If the resource is shared increment/decrement reference count. GC issue is if you scan you might not be able to tell if a program will use that resource. Defragmentation goes deeper more frequent garbage collection basically.
Relocating processes in memory? Address Translation changes virtual address from instruction to a physical address. HW interposes on each memory access, enables transparency.
1.Dynamic Relocation, Base register to transform virtual into physical and bounds to ensure address is in bounds.
Physical = virtual + base , internal fragmentation possible.
How do we manage Free space?
1.Paging, for a process's address space divide logical segments (code, heap, stack) into fixed-sized units called pages. Physical memory similarly divided into fixed-size slots called page frames. Utilize a page table per process to record where each virtual page is placed in physical. We use VPN to look up the PTE to find the right PFN. Within the page table, map virtual addresses to physical. Use valid bit to indicate whether translation is valid. Virtual address goes to TLB, if hit then to physical, else to page table and if there we've found the frame install to TLB and retry Otherwise, page fault find it in the disk.
2.Segmentation, pieces of the address space are relocated into memory variable sizes. They have 1 base & bound pair in our MMU for code, stack, and heap allowing us to place each segment independently in physical. Use the offset added to the base to map back to actual location in physical memory. Sharing memory segments like code using protection bits to specify level of protection.
How to speed up Paging? Translation-Lookaside-Buffers: part of MMU, hardware cache of popular virtual to physical address translations. For each memory reference HW checks TLB to see if the desired translation is there and if so, it's performed quickly without page table. If miss, we raise exception to trap handler to look at table and update TLB.
Mechanisms to Disk: We can swap pages to disk if we keep track of location and check a present bit to see if it's in disk. When we look at a PTE and not there, page fault and OS handles through disk I/O and updates page table. During this, process is blocked. Performance sucks if fault and context switch is high.
Page Replacement Policy: Goal: minimize cache misses.
AMAT = Tm(cost of access mem) +(Td(cost disk access) * Pmiss)
Compulsory Miss- cache empty to begin with
Capacity miss- miss bc cache ran out of space, had to evict
1.Optimal,replaces next page furthest in future by delaying page fault but uses the oracle.
2.FIFO,Random trash can't determine importance of blocks.
3.LRU, the idea is the near future is similar to recent past so if we haven't used a page recently there's a good chance, we won't use it soon. Utilizes locality(special and temporal), look at history to see what's important. Replace least recently used page.
4.Most frequently use and most recently used but trash.
Dirty Page: modified page means it must be written to disk to evict
Clean Page: unmodified so eviction is simple without additional I/O
Demand Paging: OS brings pages into memory when accessed "on demand". Prefetching is when you bring a page early but should have reasonable success. Clustering is grouping pending writes.
Working Set: Set of pages used by a process in a time interval, there's a sweet spot for min page faults and working set size.
Implementation: Page stealing, between processes if one has a smaller set and needs more you steal from the other that needs less.
*Ideal mean-time-between-page-faults = time slice length
Thrashing: When the sum of working sets for each process exceeds available memory, constant swapping of pages and no one will have enough pages in memory.
Coordination of operations with other processes:
Goals: simplicity, convenience, generality, efficiency, robust
Flow Control:make sure fast sender doesn't overwhelm a slow receiver.
1.Coordination of operations with other processes-Network connections, IPC can be lost, limited throughput & high latency. Complex.
2. Exchange of data between processes, pipelines
Shared Memory: OS arrange processes to share read/write segments but only within local machine. However, bug can wreck.
Out of Band Signal we have a reserved channel for important requests enabling possibility for preemption for queued operations.
Threads: Each has their own stack, PC, registers. Same address space. TCB states.
Kernel Threads: in kernel mode.For User implemented level threads, kernel doesn't know anything about them. As a result, when a user-mode thread issues a system call that blocks, the whole process is blocked and less context switch time. Not so for kernel mode threads and context switch takes more time & uses HW.
Synchronization--enforce--things to happen in proper order, consists of a critical section(resource shared by multiple threads that can be a counter variable as EX) serialization & notification of asynchronous completion. We enforce mutual exclusion(1 thread at a time with resource) w/ locks. Condition Variables are also used to do signaling between threads.
Race Condition: When multithreading, shared memory(crit sect) can be accessed from different threads and change it at the same time. Conflicting updates.
Locking Goals are, mutual exclusion/correct, fairness(no starving), & performance.
How to synchronize Avoid shared data, can be inefficient if you need to do it.
Disable Interrupts, simple but too much trust & ignoring important updates bad.
Mutual Exclusion on CS, atomic instructions all-or-nothing but CPU lvl so Mits tricky.
Software locking, protect CS with DS.
Approximate counter you have local counters and you can access them w/o contention and then periodically update a global counter. Should interval to update global counter is important. Locking Types: 1)Test & Set, test value & set mem location to new value. 2)Compare & swap, compare the value and swap it atomically. 3)Spin lock, can use one of the locks above and constantly run it. It will burn cycles but enforces mutual exclusion pretty well. Acceptable if contention is rare or awaited op will happen soon. To work

right, preemptive scheduler to relinquish thread. 4)Yield and spin, check a few times then yield and later reschedule. Performance, context switches & still wastes cycles. 5)Completion events, can't lock then block & have os wake when available using condition variables With multiple wait we use wait list for each event w/ signals.

Semaphore uses integer counter and fifo waiting queue, using P to consume/V to produce. Should be used for signaling as a way to implement locks. Can have binary semaphores 0/1. Semaphores can order events. Shared across threads.

Producer/consumer problem: Producer generates data into buffer, consumer takes from it. We need to synchronize. Solution: 2 condition variables & more slots to hold.

Mutexes: locking mechanism about object level locking, has low overhead & general.

Enforced Locking Guaranteed to happen no need to enforce cooperation.

Advisory locking cooperate locking scheme w/o enforcing.

Priority inversion: low priority process acquires resource that a higher priority process needs then a medium priority process preempts, leaving the high priority process blocked on the resource while medium priority finishes.

Convoy is the scenario of a long-line like a bottleneck.

Improving lock performance: 1) Reduce contention by eliminating CS and use atomic instructions, 2) eliminate preemption during CS, 3)less time in CS, 4) spread requests out, so we can have either coarse/fine-grained locks.Coarse-bottleneck Fine -complex

Deadlocks: 2 or more processes holding resources waiting for 1 another's, never ends 1)Commodity resource, needs an amount of something and happens when you overcommit. 2) General resource, client needs an instance of something. Complex dependencies can result in deadlocks, encapsulation too.

Deadlock Conditions 1) Mutual Exclusion, resource used 1 at a time

2) Incremental Allocation proces/threads allowed to ask for resources whenever

3) No- Pre-emption , when you have a resource, can't take away

4) Circular waiting, a waits on b, which waits on A

Non-Deadlock Bugs 1) Atomicity-Violation(a code region inteded to be atomic, but atomicity not enforced) Solve by adding locks around shared references. 2) Order-Violation Bugs(A should come before B, but order aint enforced), solve by adding condition variables to synchronize.

Livelock 2 or more programs continuously change state, both making no progress.

How can we avoid deadlock?

1)Total Ordering: always get a before B, partial ordering 2) acquire locks all at once 3) lock free approach w/ HW. 4)Advance reservations, if you over subscribe you have chance to handle. If we fail we want to keep running. We can do health monitoring, use of internal monitoring agents and external testing to verify no dead-lock. If issue, cold(restore last saved state)/warm(restart from scratch))/reset & reboot. Monitors are an object that has a semaphore automatically acquired on any method invocation locking object.

Event-based concurrency wait for events to occur, check what type, do work as they arrive. Use event handler to process each event. Polling will check for incoming I/O, if there's 1 thread asynchronous I/O to return control asap.

Devices Memory bus transfer data & addresses from main memory to CPU.

Canonical Protocol Status Register: read to see status of device(POLL) , Command Register: tell device to perform a task, Data register: to pass data to device or get

-Lower CPU overload with interrupts to avoid polling, after a request you can sleep and device can interrupt back to interrupt handler to enable overlap of computing & I/O. Not always ideal if you have an interrupt that takes longer than the task.

-Hybrid poll, wait a bit and then interrupt or coalesce interrupts.

Programmed I/O: When CPU is involved with data movement.

DMA a device within system that makes transfers between device & main mem w/o CPU intervention. OS tell DMA what,where, and how. Large transfer.

Communicating w/ Device: 1) Explicit I/O instruction (privileged) 2) Memory mapped I/O , Hardware makes device register available as if they were memory locations. Smaller transfer done for this.

How to store & access data on disk Large # sectors, a platter w/ 2 surfaces where data decoded there on concentric circles called tracks. Each disk head per surface and arm to move across surface. To get data.

Flow: Seek to get right track, wait for rotational delay to occur and finally transfer.

Write back caching Acknowledge write completed when it put data in its memory

Write through Acknowledge after write to disk

Device Driver Each device has own driver, abstraction/classes. Clients of disk driver is Block I/O. Clients of file systems are system calls.

Double buffered input multiple reads queued up,filled buffers waiting until app asks

Scatter/Gather IO scatter read from device to pages, Gather write from page to dev

Disk Scheduling 1) Shortest seek time: order queue of I/O by track, pick request on nearest track to complete first, but starvation. 2) Nearest block first, schedules request with nearest block address next. 3) SCAN: Sweep back & forth across tracks, favors middle. 4) F-Scan, freezes the queue to be serviced when sweeping. 5) C-scan, sweep outer to inner & reset at outer track again, but ignores the rotation. 6) Shortest position time first, considers seek and rotational latency.

RAID use multiple disks to build better disk system. Performance, capacity and reliability all go up. When you do I/O you have to calculate where to go & many I/O's.

RAID 0 : Uses stripes(block in same row), RR spread blocks of array across disks.

Small chunk =many files get striped & increase parallelism, position time increase. Large chunk size will have the opposite benefits, most use large.

*Use if you care about performance, N*B useful capacity, reliable, all disks used.

RAID 1: Mirroring, make more than 1 copy of each block on seperate disks. Not good with capacity, reliability good,logical writes need to wait for physical writes to complete so worst case seek & rotational delay. USE if random I/O perf & reliability.

RAID 4 Parity : Parity adding redundancy to a disk using less space suffering performance. For each stripe, have parity block and can recompute value for recovery. Parity disk is a bottleneck under small writes, full stripe write most efficient. RAID 5: Rotating parity, rotates parity blocks across drives to remove bottleneck 4 parallelism, replacing 4. USE if you want capacity and reliability as main goals doing sequential I/O and want to maximize capacity.

File Systems A file is a linear array of bytes each which we can R/W & each has inode #. Directories also have inode # w/ list of pairs(user readable name, low lvl name)

-Each disk divided into blocks, there's a data region for info about files(inodes) stored in inode table. Each inode has metadata about file type, size, reference to location of data(direct/indirect). There's a bitmap to indicate block status, superblock contain info about file system like # inodes , when we mount OS reads superblock to init.

Creating new file Search superblock for free inode, and pick it to use.

-Ideal to store file with contiguous blocks of memory for performance.

-I/O traffic high, read inode bitmap, write inode B-map, one to inode, one to data, one to directory inode.

-To help with performance of I/O in FS we use DRAM to cache important blocks.

DOS FAT divides space into clusters(fixed size). Boot block(never used), Super block w/ meta, FAT- split into blocks where each block points to next FAT block in file system with a # representing something.

*Use of pointers means file dont need to be in contiguous blocks of memory. Has internal fragmentation. Pros (No external frag bc pointers, no need to allocate space before & can grow, keep creating files) , CONS (Slow O(N),loss of sequential access, slow bc of pointers disk must keep finding next block)

Crash Consistency: FSCK linear search to check for inconsistencies. JOURNAL, write to the structures to note what do. If crash, you know what problems occurred. Flow, journal write to log, journal commit, checkpoint(write actual stuff). Replay transaction when recover. To verify, read after write.

LFS write buffers for efficiency. Writing to disk sequentially is the heart of it.

DATA integrity: Silent failures use checksum, if the compute value doesn't match original its corrupt. For misdirected writes just add physical ID's.

Distributed Set of Machines to build system that'll rarely fail run over the network with many computers teamworking. Want it to be Scalable, reliable, ease of use. When server sends msg, receiver sends ack. If no ack retry. There is DSM, enables processes on diff machines to share large virtual addy space. RPC, goal to make the process of executing code straightforward like on local even though over network. Has stub(pack fcn args) & run-time lib(RPCS made synchronous, when asynchronous RPC issued, RPC sends request and returns asap. Hard to synchronize because spatial & temporal separation.

LEASES get from resource manager, exclusive access and they'll expire.

MapReduce divide large probs into pieces, each performed on separate nodes . Divide data into disjoint pieces, perform same fcn distributed(MAP), combine result(reduce).

Security-Authentication(who,ID),authorization(allowed)

ACL:for-each-object, maintain list to see who can come in. Very easy to see who can access resource revoke, hard to change rights.CHMOD, in inode.Capabilities(DS): each entity keeps a set of data items that specify allowable accesses. easy to determine wat objects a subject can access, faster. Cryptographic hash: enforce integrity by checking for tampering w/ checksum. Symetric key & PK rely on secrecy of key.

Symmetric crypt: used for encrypting messages, computationally cheap, key distrib hard. Flow: Asymmetric to send key, then use symmetric crypto.

Asymm crypt: used to establish *session key,* auth sender and receiver with key pair-sender encrypts with rcvr pub key and sndr priv key so only sndr could've signed, only rcvr can decrypt (not used to encrypt messages bc computationally expensive)

HDD(magnetic disk)/**SSD(memory): SSD no Random access penalty. Fast I/O is goal