processes single-CPU system sims parallel by do rapid context switch between resident processes / multiprogramming is pseudo-parallelism / a multi-CPU sys can do true parallel / single process or thread confined to run on single CPU // process model / process is a program in exec / physically is pages of mem contain progs data and instructions, hardware register vals that spec where these pages are, the prog counter val, vals in the reg that process is using / to switch from one proc to another, all of curr-exec proc’s reg vals must be saved (so can be restored later when proc resumes), and the saved vals from the ‘next’ proc must take their place before control transferred to new proc // process creation / in unix, all proc’s created by dup an existing proc, except one: to avoid the chick n egg prob, first proc created ‘by hand’ as part f sequence of steps that starts the kernel running (the OS [the kernel] not a process, it is overseer that supervises and coordinates other proc’s) / many of tasks OS needs to do can be handles by a process called init whose PID is 1 / some of procs init makes will spawn own childs leads to big fam tree of grandchilds / many procs init makes are fired up as system is initialized/ many are daemons that remain until sys shut down / they often sleep, wake only when services requested / sendmail demon handle requests on port 25 / finger demon port 79 / webserver demon port 80 / ssh demon listen to login requests port 22 / many network demons one-time proc’s exist only as needed / a single demon simult listened to mult ports and then called on specific service for activity on that port / xinetd the extended internet services demon, starts progs that provide internet services, instead of having such servers started at system initialization and be dormant until connection request comes in, xinetd is only demon started and listens on all service ports for services listed in config file, xinitd is referred to as a super-server / xinetd single proc that handles monitoring that multiple demons used to do, demons that are rarely awoken are good candidates for xinetd / batch processing system, OS creates new procs (based on user requests), when it determines there are sufficient resources to support an additional proc (typically upon complete of another proc) / procs can create child procs that work in conjunction with the parent/ ex when a web server is started, it will often clone itself several times so that there are many proc’s available to handle incoming web requests, they run identical code but each have unique PIDs: no two proc’s can have the same PID, since the PID in effect points to a slot in proc table (a struct in kernel that keeps track of running proc’s and contains state info unique to each proc) / The shell which we use to interact with unix spawns a child proc for almost every command you do in the shell / fork creates [almost] exact duplicate of orig proc, new proc will have same reg contents, point to same set of machine code instructions, same prog counter val ( this means new proc not start at beginning, starts from the return of the fork call just like the parent) / if parent open file for writing then child will also know same fd, both procs may end up writing to same file / diffs are child have own PID, fork() will arrange for curr data vals to be dup’d in the new proc’s data space (ie parent and child do not share same data, instead given separate copies of identical vals), except fork() will set retVal reg of child to zero, and set retVal of fork in parent to PID of child / exec() overlays one proc with a new prog, making it assume new identity / new prog begins executing normally (at beginning, unlike fork) / params to exec() list the pathname to the prog that should be exec’d, the args that prog should be given, and perhaps the environment it should inherit (such as vals of environ vars) / ex if exec() vi editor, will pay attention to $TERM, find out what terminal have, so know what character seq needed to clear screen, move cursor, insert chars etc / if lie to vi about term type, it will appear to be broken / ps proc status command / exec ps, ps command now has PID shell used to have / ‘-‘ in –tcsh indicates tcsh shell was login shell / a login persists until shell dies / so when exec ps occurs, the login shell gets overwritten by ps and even though ps may have the same PID the login shell is effectively gone so will get logged out / ps - l shows more info about proc’s such as PPID / ps –lw PPID will find info about PPID’s parent, can do this up to init // process termination / voluntary terminations, procs may voluntarily exit, either bc successfully complete task, or bc it determined some condition exists that makes impossible to cont (ie something gone wrong) / proc could do something offensive to OS (ex divide by zero), OS respond by killing proc / most OS provide mech for a proc to kill another proc (assuming sufficient privileges) / in unix a norm user may kill only proc’s with same user id, but super user [root] can kill any proc / unix convention, proc ret zero on ok and nonzero on not ok (typically diff vals reflect diff errors) / $status contains the exit value of the prev command, bash and sh store in diff var, $? / cd is not a prog, it Is an internal shell command, so no child is executed // process hierarchies / in unix, a process and all its descendants define a process group, can arrange for a sig sent to parent to be delivered to every process within the grp / (ex TERM sig to every child it has created, using killpg() / getty proc’s are spawned by init, jobs are to listen on a serial port for activity / bygone: job of getty, accumulate chars until carriage return then exec login proc over the getty proc (which had in turn been overlaid by the clone of the init proc after forking) / duty of login proc, collect passwd usr types, match with passwd of the usrname [that getty has already collected / if usr identity verified then login execs usr’s preferred shell / found in /etc/passwd / init, getty, and login proc’s owned by root [with uid 0] / login proc will change ownership to match UID of usr logging in / when init initially fork()s, the resulting getty proc will have a unique PID, this PID will be retained by login [since there is an exec not a fork()] and then by a usr shell / the login shell is thus an offspring of init / when usr logs out [by exiting shell], init will be notified of the death of this child; one of inits many tasks, once it notices that this PID was associated with a serial port, is to spawn [fork and exec] a new getty to monitor the port // Process states / conceptually proc’s hve three main states: running, rdy, blocked / a proc using CPU right now is running / a proc capable of running [ but is stopped bc some other proc is using the CPU right now] is rdy (ready) / a proc waiting for some external event is ‘blocked’ (blocked proc’s may be waiting for I/O, for some timer to go off, for some signal to arrive from another proc etc) / in unix: newly forked proc’s are in SIDL state, which lasts until all the needed resources and the parent context is duplicated, then becomes SRUN state [rdy to run] / a running proc is either running in usr mode or in kernel mode / blocked proc’s are in SSLEEP state / upon exit proc goes to SZOMB state, at this point loses all resources except slot in proc table / zombie can persist until parent takes note of its’ death / a parent can give notice in advance do not want to know about status of child (formally, by ignoring SIGCHLD sigs by setting handler to SIG\_IGN), in which case the child get immediately reaped, from the proc table upon death / mis-designed parents don’t ignore child status but then never acknowledge SIGCHLD, this causes zombies / when parent proc dies, unix reassigns PPID of its children to init which acknowledges each child sig thus preventing zombies / one and only way to banish zombies is to kill the parent proc / wait() a child that terms but has not been waited for becomes a zombie and the proc table could eventually fill and allow no more procs to be created / a read from a pipeline will receive EOF if both of following conds are met: 1 no data left in pipeline 2 the proc writing into the pipeline has closed its end and exited / !! repeats the last command / uniq : echoes each line, but only if line is diff from prev // Implementation of processes / process table is array of structs / each struct contains proc program counter val, state, ptr’s to open files, ptr’s to its’ blocks of mem, stack ptr val, priority/ parent, UID etc .. / when proc from proc table is running on CPU, most of struct info in special registers, but when time to do context switch, all critical info copied into the proc control block [the entry in the proc table] : proc management: registers, prog counter, prog status word, stack ptr, proc state, priority, scheduling parameters, proc ID, parent proc, proc grp, signals, time when proc started, CPU time used, childrens CPU time, time of next alarm ; memory management: ptr to text segment info, ptr to data segment info, ptr to stack segment info ; file management: root dir, working dir, fd, usrID, grpID <- all prev mentioned in proc control block (ie struct in proc table) / /strategies to managing peripheral devs: bad: have CPU poll each dev for pending requests, more efficient to have devices ask for assistance when they need it, asking is done with interrupt requests: I/O dev asserts a bus line leading to the interrupt controller which [ if it is not currently dealing with more important interrupt] asserts a pin on the CPU and puts the dev num on the bus for the CPU to examine, dev num is used by CPU to locate correct interrupt vector [ address to jump to in prder to begin running the service routine that handles this particular type of dev/interrupt], in response to asserted pin, CPU hardware saves the currently-running proc onto the stack [ to free up registers for use by the service routine] and then jumps to the instruction indicated by the interrupt vector / service routine will then do what ever is appropriate for this type of interrupt / ex disk controller signaled it completed copying contents of a disk block directly into main mem (DMA disks can do such direct memory access – non-DMA disks would merely be alerting the CPU that they have requested contents in their buffer and in effect invited CPU to suck this data off the bus and put it where it wants in mem), DMA was big improvement, since it handles this without much attention from CPU, still had to tell disk controller where in mem to put the data, but once it does that just waits for completed sig, rather than wait and supervise the bus transfer once the data is rdy, in the case of a disk drive interrupt, the service routine would be responsible for noticing which proc’s were blocked bc new disk data had been unavailable, and proc’s which were waiting for this particular event would have their states changed from sleeping to rdy / once service routine has finished its tasks, scheduler is called, so it can digest new state of affairs (ex may decide that newly unblocked proc deserves CPU next), then assembly routine restores registers of proc that was interrupted // Thread usage / a proc has its own prog counter, reg vals, mem space, open file handles, and other resources / a forked proc can share resource with parent (ex fd) but some resources inherently unsharable (prog counter) / parent and child each have own separate mem space / threads share mem space but are like multiple proc’s / / can create multiple threads within single proc the threads share common address space, but have own prog counter, reg vals for its curr working vals, and stack for procedures (ie threads have unique versions of things needed for them to execute independently, but share everything else with the other threads in the proc / not potential to harm stacks of threads / why use threads, allows to commandeer several CPUs simult / on single CPU may not speed things up everything would run sequentially and additional overhead for context switching, but there are instances where may be useful in single CPU (such as lots of num crunching and then search through large file, one thread reads the file while second does number crunching, even though reading thread is repeatedly blocked while waiting for I/O, num crunching thread continues / note two cooperating proc’s cannot use same trick bc even though reading proc can put file data in RAM, num crunching proc cannot access it since it virtually has no way to reference data outside its own address space (best two cooperating proc’s can do is create a pipeline) / Posix thread calls: pthread\_create, pthread\_exit, pthread\_join, pthread\_kill, pthread\_yield, pthread\_mutex\_init, and pthread\_spin\_lock // Classical thread model / threads less expensive to create [as opposed to forking a proc] , bc thread mostly sharing existing resources rather than needing to give an entirely separate set of resources / three server models: 1 threads, parallelism, blocking sys calls 2 Single-threaded proc, no parallelism, blocking sys calls 3 finite-state machine, parallelism, nonblocking sys calls, interrupts / threads in usr space vs kernel space / issue is whether function calls [eg pthread\_yield()] are system calls [like write] or just regular functions [like sin()] / write() is done after switching to kernel mode, whereas sin() done in usr mode / if implement threads in kernel space, pro: kernel knows about threads and can do things more efficiently, ex finding another thread to run if one is blocked, con: thread calls becomes system calls and require a switch from usr mode to kernel mode and back, makes things much slower / if implement threads in usr space, pro: less overhead, increase in speed , cons: at expense of keeping the kernel ignorant making other management issues less efficient // Interprocess communication / types interactions between proc’s: 1 communicating data 2 coordinating activities so things happen in correct order, 3 coordinating sharing of resources / 1 easy to do, 2 and 3 are not // Race Conditions / race condition: a situation where cooperating proc’s interact with some shared resource, incorrect outcomes occasionally arise, depending on order in which they interact / happens due to bad timing // Critical Region / critical region: part of prog where shared mem is accessed / good solution: any scheme that satisfies: 1 no two proc’s may be simult inside their critical regions 2 no assumptions may be made about speeds or the number of CPUS, 3 no proc running outside its critical region may block other proc’s, 4 no proc should have to wait forever to enter its critical region / 1 sufficient prevent race conditions but not enough ensure solution fair to all proc’s (ex starvation: situation where proc is indefinitely locked out of its critical region while other proc’s repeatedly allowed into their critical region / mutual exclusion no proc’s modifying same var at same time // Mutual Exclusion with Busy Waiting / Mutual Exclusion bad: disable all interrupts, including clock interrupts (thus preventing context switches, which is what usually causes the race condition), bad bc no way to ensure usr will give back control of resource when done / bad software sol: using shared lock var, proc’s agree not to manipulate shared resource with unless first obtain lock, look at lock val, if 1 then wait (some other proc has resource), if 0 then set lock to 1 and access resource, then set lock back to 0 when done; bad bc now lock subject to race condition and will violate condition 1 / bad sol 2: round-robin sequencing / arrange for two or more proc’s take turns, will prevent race conditions, but will violate condition 3 / busy waiting: shared turn var continually tested until other proc exits its critical region and changes state of turn var, proc burns up CPU cycles waiting and doing nothing/ spin lock: lock employs busy waiting strat; fails condition 4 / petersons solution meets all four conditions, but critical region must be short / modern hardware provides test set lock [TSL] instruction to ensure read and write steps are atomic [indivisible], XCHG instruction does this to / in one uninterruptible [atomic] sequence TSL locks the mem bus, reads, writes, then unlocks the bus / locking mem bus affects all CPUs // Mutual exclusion with sleep and wakeup / sleep() [self inflicted] system call puts self to sleep [block pending some external event called wake up] / wakeup is ex of interproc communication (takes one param, proc to wake) / producer consumer : producer creates items to place in ring buff, consumer removes these items, both proc’s encounter conditions under which they cannot proceed, if cant proceed then puts self to sleep and waits on other proc to wake it up / flawed sol: both proc’s use common var [count, records num items in buff] , to decide if should block, and to decide if should wake up partner, since they both have access to count, bad timing can result in race condition / one sol: protect shared var with mutal exclusion or record num wakeup calls sent / received (basically semaphores) // semaphores / semaphore is a generalization of count and wakeup and sleep (up() down() repect, sys calls), int value of semaphore is meant to reflect num saved wakeups / if down() called on semaphore that is equal to 0 then it blocks (still decrement but val does not ever go below 0, must get blocked until val is increased, which is some other proc’s job) / up() increments and checks for blocked proc’s and if so then unblock one / possible scenarios for up(): if val is already pos num, up() just increases it by one ; if val is zero but no proc’s are waiting on semaphore, increases it by 1 ; if val is 0 but some proc’s are waiting on semaphore, val remains at 0 but one of blocked proc’s is unblocked ; in all cases up() quickly returns, ie never gets blocked as a result of the call) / possible scenarios for down(): if val is already pos num, just decreases by 1, and returns immediately ; if val is zero, down() blocks proc, now depend on other proc to issue up() before has change to proceed / semaphore manipulations must be atomic or race conds could develop if two proc try access semaphore at same time / producer consumer solved with three semaphores: full counts num filled buff slots, blocks consumers ; empty counts num empty buff slots, blocks producers ; mutex(binary) semaphore to protect shared resource // mutexes / mutex can be programmed by user if TSL instruction is available / if cooperating threads use mutex\_lock, must do thread\_yield if mutex already locked: if instead do spinlock to wait for mutex, will wait for ever: spinlock will hog CPU and not let other threads run / futex fast usr space mutex, allows proc’s implement mutex that is handled in usr space and only switches to kernel if needed // Monitors / one stall bathroom attendant / programming lang construct / compiler responsible for doing coordination / wait() signal() [sleep() wakeup()] // Dining Philosophers / **starvation**: situation where some proc’s conspire to continually prevent a proc from acquiring resource it needs / starvation differs from deadlock: in deadlock proc’s are all blocked, in starvation proc’s are active but make not progress / philosopher needs two chopsticks to eat, main() start five children and each runs philosopher() with position parameter **solution one** - philosopher waits for the left chopstick to be available, waits for the right chopstick to be available, eats, put both chopsticks down, and thinks **problem one** - philosopher picks up left chopstick and hold it forever, all die of hunger **solution two** - puts left chopstick down when unsuccessful with right chopstick, good for ethernet protocol **problem two** - everyone picks up left chopstick, put it down, picks it up again, starvation **solution three** - protect everything but think() with a mutex **problem three** - only one philosopher eats at a time, not good for larger number **solution four** - protect only acquiring and releasing chopsticks because they are shared resources, critical regions do not have eat() or think(), mutex binary semaphore to block other from accessing shared resource, state[] array each philosopher is hungry, eating, or thinking, s[] array semaphores block philosopher until chopstick available **take\_forks(i)** - enter critical region, record the fact that philosopher i is hungry, try to acquire 2 forks, exit critical region, block if forks were not acquired **put\_forks(i)** - enter critical region, philosopher has finished eating, see if left neighbor can now eat, see if right neighbor can now eat, exit critical region **starvation** - processes are active but no progress **deadlock** - all processes are blocked **Readers and Writers Problem readers and writers problem** - multiple read simultaneously and write only when no other access, bad to have two writers, no write when readers are present because of inconsistent data, **solution one** - two binary semaphores, one mutex, one for signal to write to database, both initialized to 1, first have immediate access to critical region, rc function as semaphore because manipulated only within critical regions to count number of readers, first reader blocks access to writers, last reader reenables access for writers, priority to reader or writer **problem one** - starvation of reader or writer **solution two** - queue both readers and writers in order of made requests // Scheduling / batch processing: throughput [ num jobs completed per unit time] most relevant factor, so scheduler should attempt maximize this / ex having mix compute and I/O proc’s more likely to make efficient use of resources / if too many proc’s to fit in available mem, then excessive page faulting may occur and resulting overhead will drag throughput down / interactive system: response time [ time from when a command is issued until it is completed] most relevant metric [ analogous to turn around time in batch sys], on single usr pc scheduler not make much diff, but on multi usr interactive sys [like edoras] good scheduling policy greatly enhances performance / good response times requires clock interrupts and context switches whenever compute bound job exceeds allotted time slice: if proc were instead allowed to hog CPU until I/O occurred, other proc’s could be delayed unreasonably, giving poor response times / context switches req overhead [housekeeping work which OS performs in lieu of directly making progress toward completion of a usr proc / smaller time slice = larger overhead since cause more context switches / balance req optimum performance / curr time slice == 25 ms / preemptive scheduling alg: a sys that enforces time slices / sched alg goals: all sys: 1 fairness – giving each proc fair share CPU 2 policy enforcement – seeing stated policy is carried out 3 balance – keeping all parts of sys busy / batch sys: 1 throughput – maximize jobs per hour 2 turnaround time – minimize time between submission and termination 3 CPU utilization – keep CPU busy all time / interactive sys: 1 response time – respnd to requests quickly 2 proportionality – meet usr expectations / real time sys: 1 meeting deadlines – avoid losing data 2 predictability – avoid quality degradation in multimedia sys // Batch scheduling / schemes for batch sys: fifo , shortest job first[magic] , shortest remaining time next[also relies on unknowable metric] / three opportunities or scheduling: 1 admissions scheduler – decides which proc’s to inject in the sys and when to admit additional proc 2 CPU scheduler – chooses among jobs in rdy queue each time context switch 3 memory scheduler – monitors page fault situation and momentarily swaps out some proc’s to allow remaining proc’s have enough elbow room / admissions scheduler monitors various sys aspects to decide when let in new proc, does not add new proc if abnormal amounts of page faults or swapping / scheme coordinating these three called: three level scheduling // scheduling in interactive sys / round robin (rr) – proc’s queue up for CPU, take turn, then go end of line, downside: I/O proc’s shortchanged (sol: allow to make brief use of CPU before compute intensive proc’s hod) / priority scheduling (pq) -runs jobs based on priority, at each context switch high priority chosen run, downside: low-priority can get starve (sol: dynamically adjusting priority based on past behavior [ but can increase overhead]) / combine rr with pq to form priority classes – highest class [made up of procs] gets chosen, then proc’s in that class do rr / unix uses dynamic priorities / every proc starts with avg priority, which then adjust based on past behavior, if recently used CPU then less priority, if not used CPU recently (due to IO, semaphore blocking, low priority) then priority goes up / scheme allows I/O maintain high priority, heavy compute low priority but never starved, proc’s with same priority use rr / priority determined by: half: proc’s behavior in most recent time slice ; quarter: previous time slice ; eighth: one before that ; etc.. diminishing exponentially / sounds complicated but math is fast with: prev weighted avg plus current behavior then divide by two (ie one add and one right shift) / load avg: avg num jobs in run queue during given period of time, w and proc name or usr name (in shell) gives load avg // Deadlock / deadlock – if each proc in set waiting for an event only other proc’s can cause, ie all blocked / condition resource deadlock: 1 Mutual exclusion – each resource either currently exactly one proc or is available (resources can be held by at most one proc) 2 hold and wait – proc’s currently holding resources that were granted earlier can request new resources (once a proc is granted a resource it doesn’t have to give back, even if curr blocked and waiting for additional resources) 3 no preemption – resources previously granted cannot be forcibly taken away from a proc, must be explicitly released by that proc (resources cannot be forcibly taken away once allocated until the proc explicitly decided to release the resource) 4 circular wait – must be circular chain of two or more proc, each must waiting for resource held by next member of chain / ostrich alg – pretend does not exist (most reasonable response) / another strat – detect deadlocks and attempt to recover by forcibly yanking resource away from one proc in deadlocked set, or killing one of the deadlocked proc’s) / another strat: avoid/prevent deadlock, such as ensuring never meet all conditions of deadlock (ex ensure circular wait condition never occurs, do this by OS assigning num to each exclusive resource, then not grant requests for lower num resources if a proc has already been granted a higher num resource, ie procs must request resources they need in the order specified by OS, downside: will not allow out of order request (even if programmer needs it in his alg) // memory management / volatile – needs electric power to retain its contents / RAM volatile / hard drive not volatile / disk drives: slow but cheap and nonvolatile / cache – very fast but expensive and volatile / RAM fast, medium price, nonvolatile / mem manager keep track of proc’s using which part of mem, free areas to be allocated, and areas to be swapped between ram and disk // No mem abstraction / monoprogramming model: can run only one prog at a time / but to run mult proc’s pseudo simult, all need to be in mem at same time, scheme: divide RAM into partitions put one proc in each partition, would need hardware mod prevent one proc from accessing another proc’s mem / equal size partitions: downside: if a proc does not need all mem now waste of mem / diff size partitions: good bc less mem waste, bad bc which jobs allocate to which partition queue and one queue can become overloaded and if one proc larger than largest partition then cannot run on hardware / CPU utilization (% time cpu is busy) = 1 – p^n , p – time waiting for I/O complete, n – num proc’s in mem at once / help decide if need more mem, assuming know type of proc’s / ex 50% io model, total RAM = 512 Mb, OS = 256 Mb, each prog need = 128 Mb, means can run two proc’s simult, CPU util = 75%, add 256 Mb to total, now can fit 4 proc’s, CPU util = 94% , throughput = (94-75)/75 = 25% increase , this is desirable, but if do again: add another 256 Mb total mem, CPU util = 98%, throughput = (98–94)/94 = 4% , less desirable maybe not worth // Memory Abstraction: address spaces / relocation problem – partitions require large address locations offset from partition 1 / software sol: linker given start address of prog to be loaded and it mods each mem address accordingly (but not constants or opcodes), downside: resulting exe cannot be run in another partition, hardcoded for that particular spot in mem / hardware sol: (preferable) BASE and LIMIT registers, exe’s compiled as though run at mem address zero, then CPU mods each instruction by adding BASE to every mem reference / ex proc run at address 700k, hardware consults BASE reg and adds 700k to each mem address / pros: no changes to exe file needed, cons: every mem reference needs add time / both soft and hard schemes do not address protection issue [prevent one proc from accessing another proc’s mem] / since no neg address don’t worry about below BASE, but can grow too large so set LIMIT register / if pass LIMIT reg then mem fault, instruction not completed, proc aborts with fatal err / virtual mem makes this scheme obsolete // swapping / swapping – moving proc images in mid-execution between RAM and hard disk / use when too many proc’s in main mem, a proc evicted for new proc / issue: for sys that req running proc’s to be contiguously present in mem for entire existence, swapping can lead to gaps in mem, virtual mem solve this / issue made worse by progs that dynamically allocate mem, changing size of proc image (sol: over-allocate mem, allow room to grow, reduces overhead, issue: waste mem space) / OS keeps track of available holes, complicated bc images may move around / sol: holes tracked by bitmap , but searching for long runs of zeroes [free space] tricky due to word boundaries in bitmap / other sol: linked list of seq of proc’s and holes reflecting curr mem use, pros: search free mem easy, cons: maintaining list when proc’s are deallocated / both sol need alg for determining which hole new proc will fill (assuming new proc can fit in multiple holes) / algs: first fit, best fit, next fit, worst fit, and quick fit, none do good job against fragmentation, moot due to vm // Virtual memory / hardware views full prog as sequence of equal-size chunks called pages that contiguously form the proc’s image / due to size [spanning address space that far exceeds addresses available in physical mem], image cannot fit in mem all at once, and hardware not try to bring all in, at most a few starting pages before prog starts executing, when during execution an address is encountered that is not on a page that is currently in mem, page fault occurs and proc suspended until OS brings in necessary page / sys works using indirect addressing, OS keeps track of locations of pages in mem, takes each virtual addr and translates to corresponding physical addr, each page has own translation offset, so pages can be diff locations in mem [non-contiguous] / page: chunk of virtual addresses / page frame – chunk of physical addresses / if page fault occurs proc suspended until chunk of that proc image comes in from disk [ usually taking place of some other chunk that not been used for while] / prior to vm, address needed by CPU placed directly on bus verbatim and exact RAM address accessed / with vm, virtual address goes to MMU (memory management unit), translates to physical address then place on bus / MMU translation done via page table, one for each proc, uses page num of each chunk as inde to array that specifies physical address [page frame] of that page / vm pros: external fragmentation solved (at worst internal fragmentation where proc needs only of page but nevertheless assigned entire page frame, only happens on last page of the image), allows enormous address space to be used (no prog too big), mem protection built in (to ref physical memory must go to page table translations, if page frame is not in ‘your’ page table, no way to access it, so can’t peek at page frames no assigned to your proc), more efficient use of RAM (several usrs can run same program without having to copy machine code, can share code segment) / vm cons: overhead of page table translations, 32 bit addr need at least 4 Mb for a proc (fine with lots of RAM) but if 64 bit addr then each running proc need 30 million Gb, not feasible, solution: multilevel page tables // multi-level page tables / indirect addressing avoids needed so much page table data in mem, also alternative to going to huge page table to directly look up page frame, instead refs first fo to top lvl page table which refers it to second lvl page table then finds true location of page frame / cuts down num page tables / unix, top of mem holds environ vars and frame stacks, so second lvl page table for this area actively needed / lowest part of vm is code segment, where machine code instructions for the proc, need many page frames for that / data segment [statically allocated data] just above code segment / heap [where dynamically allocated data lives] above data segment / ex if pg frames are 4k , and lowest second lvl page table contains ptr’s to 4 Mb of data, if prog and data fit within this, then these may be only two second lvl page tables needed, in top lvl of page table only three present absent bits are set to 1 (what if need more ? ex proc recursively calls a routine and stack grows past boundary, OS will notice a ref to second lvl page table that is not yet present, causing a page fault, putting proc to sleep, appropriate second lvl page table dragged into RAM, and a page frame will need to be allocated and pointed to by the entries in the second lvl pg table before the proc is rdy to resume running, although overhead in fetching additional second lvl page tables, the scheme avoids clogging mem with huge amounts of page table data, bc most virtual addresses never get used by the proc // structure of a page table entry / most of entry is physical address of corresponding page frame, rest are flags: present/absent (indicates whether physical address corresponds to current reality or is bogus) ; protection (if page can be read/written/executed) ; modified (if change on page in RAM, new data have to be written back to disk [if no change then disk data already ok and page frame can be reclaimed and used for other purposes without write]) ; referenced (set to 1 with each ref to data within page frame, intended to decay over time to zero, so can tell when page not been used in while, helpful when OS needs choose which page to evict when needs room for new page) ; caching disabled (if page mapping an area that part of I/O dev’s mem, then do not want to look at in-RAM copy, instead request new cpy of dev’s mem each time we look, so when dev makes change will see it [rather than repeatedly looking at stale static version in RAM]) / when page kicked out of RAM and stored on disk, effectively forgotten, tables are for active pages in RAM, OS keep track of location of info on disk, so can brought in and put in page frame when needed // speeding up paging: translation lookaside buffers / page tables are enormous and if have multi lvl page tables then might take 2,3,or 4 mem accesses just find one piece of data / locality of reference: over small interval time same set of addresses tend be used over and over (most used set) / hardware only need handle between 16 and 64 most used addresses for large boost in performance / this hardware is: associative memory or a translation lookaside buffer (TLB) / 16 to 64 most commonly used pages / field of TLB match normal page table/ therefore no need to incur overhead of fetching info from RAM / thus if need addr MMU first looks to TLB / hardware conducts search in parallel / if match is found page frame of proper entry combined with offset and put on bus to be extracted from RAM / if no match then default to usual way of looking at page table, when found entry is placed in TLB / large TLB can make software solution to handling page faults viable / do this by: also maintaining a software version of TLB [software cache of page references to supplement hardware of TLB], needed bc TLB miss results in having to look up some data in the page tables, and possible needed page table is also not in TLB cache, generating further fault, sol: have one TLB entry always present in TLB which points to special page frame of recent TLB entries, software first looks there when there is a hardware TLB miss // inverted page tables / inverted page table – page table with one entry per physical frame, where entry lists particular page associated with frame / hash table can make finding any TLB misses relatively swift, hash table is hashed on the virtual page num / size of hash about as large as num frames, avg chain length is 1 // page replacement algs / attempt to make an intelligent choice without much overhead ( don’t want to waste a lot of time deciding what to do: every bad decision better than much delayed) // optimal page replacement alg / [magic] see into future and evict page that would never be used again or not used for longest time / is impossible but can be compared against real algs | Need to know all pages in advance. Take out furthest used or page not referenced again first. If page not ever referenced again remove first. / LRU lease recently used is good but requires special hardware / aging is approximation of LRU gives good performance / WS-Clock depends on ‘working set’ has good performance / NRU not recently used better than FIFO first in first out -FIFO - first in, first to go out( remember the first one to come in oldest one in buffer gets kicked first). \* Don’t manipulate order if no page fault. // the second chance – FIFO with R bit of oldest page, if 0 page is old so replace, if 1 clear bit and put age to end of list of pages (if all pages have been referenced then becomes FIFO) // the clock / same as second chance but instead of linked list and reshuffling, instead use round robin // Least recently used - take out page last referenced. \* manipulate order if no page page fault occurs. / maintain usage history, impractical and expensive to do in software / but with special hardware close to optimal (64 bit counter or maintain nxn matrix where n = num page frames) // simulating LRU in software / fast approximation to LRU / NFU (not frequently used) > NRU / NFU keeps track of num references , but suffers from remembering too much (ex page that had many refs in past but is not longer being used may never get evicted), instead want weighted avg / recent time slices account for half, previous time slice a fourth, one before that an eighth etc.. / aging accomplished by tracking whether access or not during clock tick, R bit of each page zeroed at start of time period, if page is ref’d between clock ticks R will be reset to 1, each page has record of what has gone on in past n clock ticks, current R bit inserted on left, then R bits are zeroed out again rdy for next cycle // working set page replacement alg / typical proc exhibits locality of reference, ie small nume pages of proc tend be used most often, most of image rarely referenced / working set – set of most used pages / proc’s may hae diff working sets as enter new phases of their processing / tracking each proc’s working set can greatly decrease num page faults / if cannot keep working set of each rdy proc in mem at same time, good indication something should be swapped to give remaining proc’s more room / when proc brought bac kin after swapping out, paging can do nothing and not preload needed pages, but this causes many page faults as working set slowly brought into mem as needed and causing context switch to another proc at each page fault, better solution to preload working set before allowing proc to resume running, avoiding this issue / each proc given own allocation of page frames, and must stick with exchanging own pages when there is a fault, does not get to steal page frame from some other proc / over time, if proc exhibits thrashing, OS may increase working set size for thrashing proc / tracking complete set of pages expensive, want set of pages have been ref’d in past k instructions, but this hard to implement, so settle for num ref’s during some time t, (where t is some small num clock ticks, represented by greek letter tau) , not wall clock time, instead virtual time: only counts time spent actively using CPU, each proc has own virtual time counter / page table still contains: R (referenced) bit, M (modified) bit, and now new field: approx virtual time page last ref’d (virtual time proc experienced page fault at point when R = 1), similarly to other algs, R and M set by hardware and software resets R each clock tick / when page fault occurs, ALL PAGES in resident set [the pages allocated to this proc] is examined, those with R = 1 updated so Time-of-last-use field reflects current virtual time, else fif R = 0 check if page still belongs in working set [ pages that have not been referenced within the last t [tau] time units are by definition not in working set, so as soon as one is found evict that page to make room for new one / pure [and expensive] version of this scheme, alg cont to process remaining page frames so all frames with R = 1 update virtual time / best case: find page in resident set that is not in working set / case: run across many pages R = 0 but age still less than t [tau], sol: remember oldest of such pages and replace that one / worst case: every page has R = 1, sol: pick page at random [ preferring non-modified pages ] to avoid writing to disk, if repeats a lot then indication resident set size too small for proc and OS should consider increasing size to prevent thrashing // WSClock Page Replacement Algorithm / more efficient version of working set alg, uses trick from clock alg to avoid look at every page in resident set to update Time-of-last-use field for pages with R = 1 / software now also updates virtual time stamp for those bits where R has been 1, so when page fault occurs only have to update time stamp on new page that displaces evicted one / ‘clock’ hand travels around resident set, looking for page with R = 0, if find one use time-of-last-use field to determine if still in working set, if not then not dirty and have ideal candidate to evict (unlike working set alg, don’t care if most ancient page: instead do eviction based on in/out of working set, which governed by less/greater than cutoff time Tau) / if page not in working set and is dirty, schedule the write, but keep looking since might find clean page outside working set to evict, possible clock hand will come full circle without finding ideal conditions, , means could not find page both outside working set and clean, three way this can occur: 1 every page is in working set and all clean, sol: replace page clock hand currently pointing to 2 every page is in working set and some are dirty, sol: replace one of clean pages (dirty pages already been scheduled to be written out, after which will be marked clean again, and will be candidates for replacement in future) 3 some pages not in working set, but the non-working set pages are all dirty and have been scheduled to be written to disk, sol: do not want evict something from working set when better choices, so keep clock hand moving until finds clean page to replace, if no clean pages then keep searching, eventually some non-working set page will be marked clean (note may not be first page we asked to have written: disk controller will order writes in most efficient way, which may be diff from order we asked, so keep searching rather than waiting for one we ‘think’ will finish writing first) / unlike WS scheme save time by not having visit every page in resident set, and save time by not having tracking ancient time stamp / when page clock hand pointing to is replaced, move on since guaranteed that new page have most recent time stamp // Summary of Page Replacement algs / Global allocation policy: if page fault results in most advantageous page being replaced without regard to which proc owns that page [ ie proc’s can steal page frames from each other] / Local allocation policy: allocated resident set for working set of a proc, and force page faults to only steal pages from curr resident set, it is best periodically evaluate resident set size of proc and make adjustments / when thrashing occurs, indication curr collection of proc’s don’t have enough collective RAM and some swapping should occur to allow more pages be assigned to remaining proc’s to reduce num page faults / larger page frames means page tables can be smaller, but larger pages means greater internal fragmentation (wasting RAM if small tail of prog placed in mostly empty page frame) / if architecture limits to a small address space [such as 16 bit machine] one way to double space is virtual addressing for the prog and data separately, allowing each hunk appear having 2^16 bytes, I-space (prog) and D-space(data) have separate page tables, doubling virtual space / pages can be shared (ex 12 users on vi, their page table point to same machine code), when a proc finished its pages should be released immediately rather than letting them age (unless being used by another proc) / hunting for free page as needed slows things down, better to have supply of ready to be replace frames always available, many sys use paging demon periodically inspect mem if too few page frames curr free, go find some pages to evict, good candidate for replacement marked as free else if dirty then scheduled for write back to disk, newly freed pages are not overwritten immediately, form a reserve so page faults can be handled quickly, possible a page marked free will be needed before handed off to some other proc to satisfy a page fault, in this case owner simply given back the page since contents still valid //a two handed clock implements this scheme, front hand moved by paging demon, and other hand points to page to be replaced, demon stay far ahead of other hand scheduling writes of dirty pages, when page fault occurs back hand advances until find next clean [and unreclaimed] page (more likely due to demon) when gets close to demon, demon wakes up and collects more candidate for free list If(R==0 && M == 0){evict and put new page in and set r bit to = 1. Then move the second hand (SECOND HAND ALTERS R BIT) page by page to the next best candidate (page with R==0 and M==0), setting the R bits for each page we traverse through to 0 as we go along} if(R==1 & M==0){second hand sets R bit to == 0 and moves along to next best candidate setting the R bits for each page we traverse through to 0 as we go along} if(R==0, M==1){don't evict go to next candidate changing r bits to 0 as we traverse} if(R==1 && M==1){don’t evict, set R bit to 0 and iterate through to next best candidate setting r bits to 0 as we go along}. If daemon hand is to close move it a few clicks scheduling writes // File Systems / File – unit of storage / file sys – method of organization // File naming / directory separation char: unix: / windows: \ , cannot be part of file name (or path cannot be parsed reliably) / some sys insist on extensions (ex require dot new end of name followed by up to 3 chars) unix doesn’t / unix does not allow / or null terminator to be part of filename but allows pretty much every thing else but not somethings not recommended (< , ! ,spaces etc) / windows file extension determines which prog is launched, few unix apps care about extension other than : rcs[the Revision control sys] requires repos to end in ‘,v’ and C compilers insist source code end in ‘.c’ / unix – mult file ext allowed (ex .h.tar.gz,v indicates rcs repo (,v) for series of C header files (.h) that have been rolled into a tape archive (.tar) and then compressed with gzip (.gz) / instead of relying on ext unix files tend have ‘magic number’ in first two bytes of most files (ex postscript file begin with asci code for chars: %! , and shell scripts begin with #!, in this case windows require .mpg ext) / both unix and windows regard files as sequence of bytes ,but old sys organized data into records // File Types / data files are termed regular files (to distinguish from specialized files) / Unix has: named pipes, block-special files and character special files / named pipe is FIFO queue that mult proc can use / char special files – essentially portals to various hardware (ex send data to /dev/tty job of unix dev driver monitor this filesys node to ensure data appears on screen / block-special files are nodes that serve as portals to block devices, such as physical disk drives / normal text file used to be either EBCDIC (IBM format) or ASCII / IBM format based on punch cards with holes digits one hole chars two hole, binary codes for chars not rational / ASCII more rational, letters assigned 26 contiguous binary codes, and designed with lexicographical order (dictionary order) / Line termination only non standard part of ascii / EBCDIC outdated / agreeing on ascii advantages: all editors can handle file, progs can pass data between them without encoding issues / binary file - files with complex internal structure / archives and machine executables are binary files / file transfer protocols that transport files between hosts have friendly feature that turns code for every linefeed [decimal 10] into code for carriage return [decimal 13], to allow ascii to be read on a host using diff line termination / this feature on binary would be bad / therefore progs like ftp have ascii and binary transfer modes / unix executable begins with magic num (ex if tried transfer exe from one CPU type to another, OS would note magic numer is inappropriate for the architecture and refuse to run it), header also contains: entry point which [when combined with text size info] specifies location of first machine code instruction to begin executing, data size, text size, BSS size, symbol table size, flags/ outside of header section includes: text, data, relocation bits, and symbol table / archive is collection of .o files / archives are wrapped up saying where each module starts and ends and generally stored in /usr/lib / once compiler has take main.c and created main.o linker takes the various .o files it needs from the archive and combines them to form an executable a.out file / archives ext = .tar (.tgx for compressed archive) [could contain any set of files not just .o files] / archives includes many pairings of: a header: module name, date, owner, protection, size ; and an Object module / strongly type file strat: file ext mandatory and need proper ext // file access / initially comps access info on storage devs sequentially no rewind must loop around/ then drums and disk drives avail and could access anywhere (random access) yes rewind // File Attributes / attributes: protection, password, creator, owner, read-only flag, hidden flag, sys flag, archive flag, ASCII/binary flag, random access flag, temporary flag, lock flags, record length, key position, key length, creation time, time of last access, time of last change, curr size, and max size / name of file kept only in dir, everything else in inode [index node] / inode fields: mode [2 bytes] (file type, protection bits, setuid, setgid bits), Nlinks [2 bytes] (num dir entries pointing to this inode), UID [2 bytes] (UID of file owner), GID [2 bytes] (GID of file owner), size [4 bytes] (files size in bytes), addr [60 bytes] (address of first 12 disk blocks, then 3 indirect blocks), Gen [1 byte] (generation number (incremented every time inode is reused)), Atime [4 bytes] (time file was last accessed), Mtime [4 bytes] (time file was last mod), Ctime [4 bytes] (time inode was last changed) / dir entries are ordered pairs of names and inodes, legal to associate diff names with single inode, done via ln command, creates a hard link, create new dir pair entry / soft links [ln-s] are indirect links to a file and do not change Nlinks hard count, soft link is a special type of file where its data is just the path to file which are linking to / implies file sys tree not actually a tree / inodes list locations of data blocks that comprise actual file / files are not stored contiguously (bad idea bc fragmentation, only would work with CDROMs[nothing is deleted so no holes can form]) / important inodes are all same size so can be indexed as array of records / first ten ptr’s in inode point to blocks of file data / eleventh ptr points to single indirect disk block that contains list of next 1024 data blocks / twelfth ptr points to double indirect block points to 1024 data blocks each one points to single indirect block which in turn points to 1024 data blocks / thirteenth ptr points to triple indirect allows for billion more blocks / linux curr has 12 direct pointers instead of ten / scheme works bc most files are small and thus only use first 12 direct ptrs if that, keeping inode size small and files are access efficiently / open files new file ptr to indicate where next byte is to be read, cannot keep file ptr in inode or if two proc access same file then both will access same file ptr and each only get part of the file, cant keep in file descriptor table or mult proc writing to same file would overwrite each other / sol: maintain fd tables for each proc and kernel keep track of list of attributes of each file (inode info, flags for reading or writing allowed, curr file ptr, which inode on which partition) in structure called open file description table / removing an entry in a dir does not mean file is deleted, could still have other names associated with it / can remove a reference to file using unlink sys call / so rm removes dir entry but not necessarily move files data blocks to free list for reuse ( only does this if Nlink drops to zero) (and each proc has closed the file) / every dir contains a dot [‘.’] which means “this dir”, the dir has a name and inode pair in the dir above it, so every dir has at least a link count of 2 / every dir has dot dot entry [‘..’] points to its parent, so if a dir has lots of sub dir’s then link count for that dir could be large / cannot hard link a dir, bc might conflict with what dot dot entry means // file operations / 11 common sys calls: 1 create (file created with no data, just announce arrival and set some attributes) 2 delete (tell OS can free associated disk space) 3 open (allow sys to fetch attributes and list of disk addr into main mem for rapid access on later calls) 4 close (when all accesses are finished, attributes and disk addr no longer needed so close to free up internal table space, sys’s encourage by imposing max num open files on a proc, closing file forces writing of files last block even if last block not entirely full yet),5 read (data usually bytes from curr pos in file, caller must specify how many data needed and a buff to put them in) 6 write (data written to file, usually at curr pos, if cur pos == EOF files size increases, if cur pos in middle of file then existing data overwritten and lost forever) 7 append (restricted form of write, only add data to end of file) 8 seek (random access, specify from where to take data, repositions file ptr) 9 get attributes (ex makefile examines mod times of all source and object files and arranges minimum number of compilations required to bring everything up to date, must look at modification time attribute) 10 set attributes (some are usr settable, most flags for ex) 11 name / unix uses stat() sys call for attributes instead of get attributes / chmod, chown, utime mod attributes (ie set attribute) / utime lets lie about last access or mod time, but third time stamp (Ctime) not so easly altered / delete done with unlink() // read() will regularly return exact buff size, but will return smaller num when EOF reached // File System Layout / most storage dev’s allow comp to boot from them via **MBR** [master boot record] stored in first 512 bytes / typical motherboard read from beginning of dev, put data in mem, and execute tiny prog called **bootstrap loader** [its job to bring in a larger prog, such as OS or more sophisticated prog that will then load OS] / for disk drives, partition info follows MBR, can be logically carved into pieces, each piece behaving like separate disk drive (so each has a 512 byte block at beginning) / **MBR** prog of disk drive examines partition info, locates partition that is marked active, drags little prog into mem and controls jumps to its entry point / little prog loads the OS associated with that particular partition / since independent diff partitions can have diff file sys / typical **disk layout**: MBR, partition table, then disk partitions which are made up of: boot block, superblock, free space mgmt, inodes, root dir, files and dir / for **unix file sys**: boot, multiple block groups, each block group made up of: superblock, grp descriptor, block bitmap, inode bitmap, inodes, and data blocks / boot is MBR (each block grp still has own boot record), superblock contains essential info about file sys, each file sys type (FAT, NTFS, Linux, BSD etc) identified by a unique byte num / MBR in process of being replaced by **GPT** (GUID Partition Table) and UEFI (Unified Extensible Firmware Interface) / outdate: cylinder head sector (CHS) addressing reflected physical locations of data blcks on a stack of spinning disk platters, limited to 31.5GB / logical block addressing (LBA) allowed 26 bit specifiers expand to 32 bits, but best can do with traditional MBR partition tables / drives now exceed this size so need new partition table standard, can handle ZB (2^73 bytes) range / **GPT** layout keeps same layout as MBR (ie MBR always stored in first 512 bytes of disk and contains machine code getting boot proc going: BIOS loads machine code into RAM and starts executing) for backward compatibility / GPT stands for GUID Partition Table. ... GPT allows for a nearly unlimited amount of partitions, and the limit here will be your operating system — Windows allows up to 128 partitions on a GPT drive, and you don't have to create an extended partition. On an MBR disk, the partitioning and boot data is stored in one place./Why are we switching to GPT standard?ANS: MBR is stored in the first 512 bytes of the partition. It contains machine instructions for booting the computer. It did not, however, have the ability to navigate partitions or file systems. It contained code to load a second-stage bootloader, which could. MBR is limited to disks up to 2TB and only 4 primary partitions. GPT, GUID Partition Table, however, replaces MBR. It starts by giving each partition a Globally Unique IDentifier for each partition. In theory, every GPT partition ever has a unique identifier, due to how long the string is. Drives can be super huge and the number of partitions can be much, much higher. Windows allows for 128 partitions. Is backwards compatible so it can be used on MBR machines. / **GPT** REPLACE **IDE**: GPT allows for more partitions and in a more flexible structure./ **GPT**, first stage bootloader code stored in MBR smart enough handle new GPT partitions, but still using BIOS to boot, modern sys **replacing BIOS** with **UEFI**, which doesn’t rely on boot sector / old first stage bootloader code in MBR could not recognize partitions nor navigate filesys [bc 512 bytes], instead contain code find code for second stage bootloader which could identify partitions and navigate filesys, and therefore was capable of finding OS and loading it into mem [ note if move location of second stage bootloader, must run utility to rewrite MBR with new cylinder/head/sector location] / **UEFI,** MBR not used, firmware already contains enough smarts navigate filesys and locate latter-stage bootloaders by parsing dir entries until finds designated set of data blocks / pathnames stored in firmware vars which can change and view after turning comp on / firmware interface can be queried over network after power on, even if sys has no OS (not installed or won’t boot), allowing diagnostics and recovery services to be performed remotely (this ability turned off after handing control to OS) / UEFI includes graphics support and other dev drivers that OS can use until loads its own device drivers over BIOS / UEFI has secure boot capability ensures only acceptable OS loaders (ie a windows laptop refuses anything other than windows) / in Unix file sys, **superblock** specifies num inodes present [ calc from indes array beginning and end], how many data blocks for use, and where list free blocks begins (if superblock ruined file sys no longer useable, which is why modern unix file sys keep replica of superblock / linux **ext2** file sys follows Berkeley fast file sys, both split partition into several mini partitions (also called blocks), so inodes be closer on disk to their corresponding file data blocks, which should reduce avg distance read head armature has to move to access file / new linux file sys (**ext3** and **ext4**) increased allowable size of partitions, files , and dir’s to million Tb, 16Tb files, and 64k subdirs and intro journaling / Berkeley fast file sys and linux, inodes all same size to allow easy access in an array [efficiency] / BSD allow variable length file names [up to 255 chars that fit into 256 buff bc null term], accomplished by: entry size field says how far to jump to get to beginning of next entry (so if subsequent file get deleted then entry size will change to point at next available file) / searching for file in dir involves comparing names in each entry, using linked list wont slow down bc not indexing, instead comparing / if do ls in thousand entry dir, response slow bc traverse one node at a time in linked list, but faster next time bc filenames are cached [kicking out less used path names] / new file sys (ext3 and ext4) use HTree where names are hashed, faster look up / don’t put inodes at beginning of disk bc head must travel to very edge of disk repeatedly, head is slow, not good performance / improvement: put inodes in middle so half disk blocks infront of them and half behind them / even better improvement: split inode array and puts inodes near relevant data blocks / BSD sol to internal fragmentation: two sizes, regular data block can be carved into 8 equal pieces, large data blocks for large files / BSD file sys speed incr: distribution of inodes throughout partition, and use of two block sizes / mascots (linux penguin, BSD demon w pitchfork, BSD fast file sys demon with pitchfork and track shoes) // Flash Tech and SSD / two types of flash mem: nand & nor / **Nor** used for things like BIOS since only small amounts required and byte addressable [write a single byte without affecting other bytes] / **nand** is page addressable, natural fit for file systems / nand can easily turn any 1 bit into a zero bit, but zero to 1 can only be done by erasing entire page by turning an entire page to ones / so writing to page requires erase and then add zeroes / magnetic media eventually wears out, but MTTF [mean time to failure] huge / flash mem, page not reliable after 100000 erasures / prob in norm file sys where blocks get overwritten many times wearing out flash / **wear-leveling** – distribute new writes across large set of currently unused candidates / **virtual layer** to even out the wear on the set of blocks. / sol: virtual layer over flash mem / (ex for magnetic disk drive: controller gave disk a standard platter/track/sector designation, on disk electronic provide virtual layer of fixed number of sectors per track, and translates to appropriate place / on disk electronics secretly remap bad sectors to unused sectors and outside world does not know / virtual layer for flash presents array of fixed pos blocks / on drive electronics on write moves to new block but same name ref (ex read from block 45, goes to block 45, write to block 45 then moves to next free block, say block 70, and writes there but block 45 is now associated with block 70, old block 45 designated as free data) / when write a new block is chosen from free queue and old block is put into free queue / can put standard disk file sys on top of this virtual layer / for best performance: always have blank pages available so writes can take place immediately, if make true SSD then need controller to handle this / garbage collection: when controller told to overwrite block, it remembers stale data from old block and when not have pending read/write to handle will blank the old block / meanwhile new data that written on blank block and idea of where old block is points to this new block / at logical level of file sys, many more stale blocks than SSD controller would know about, when delete file append data block addr to respective free lists, don’t actually go in and zero, controllers garbage collection routines unaware contents of these blocks no longer of interest to OS, maintains this useless data, disk driver for SSD make use of special TRIM command, alert SSD controller a particular block no longer contains valid data (marks as stale, ie target for garbage collection) / controller gremlins if block flash mem used a lot [controller keeps track of num times block erased] gremlins look for unused mem and secretly remaps the pair / standard disk file sys constructs optimally designed for constraints imposed by spinning magnetic disks (ex prefer write related data on adjacent tracks and all on one cylinder, since large time penalty if have to move head assembly, nonissue for flash drives, since no moving parts, instead worry about wear leveling // Log structured File sys / with out log file sys: writes include many small writes in unrelated places: if make new file, write info inode, update things like list of free inodes, update disk for free data blocks, file name must be submitted into a dir which means write to dir and the inode for the dir itself (such last mod time) / so even though write contiguously at high speed on disk, each change happens in diff places and effective rate **can** drop to 1% of the optimal write rate / in log sys, save those many writes in a log entry called a segment [writing contiguously at 100% efficiency] onto disk, little while later write another log seg just behind the prev one, and this almost only write ever to disk so writes are now fast / reads get little slower, but don’t do as many reads anyways [due to efficient caching] // issue: in **unix file sys**, inodes were put into array of fixed size structures, in log sys inodes are strung out all over the place, sol: keep a separate array [**inode map**, indexed by inode num], this array points to inode location on disk, inode itself has usual data block ptrs but now these ptrs pnt to blocks of data in various parts of log file / if change an inode, new version of inode written out somewhere in log file, update location in inode map [inode map kept on disk so not lost if power goes out] [inode map also cached to reduce disk accesses] / Dir data blocks retain same structure as in traditional sys / when get to end of disk, **cleaner thread** is woken periodically to do sort of garbage collection [ better thought of as compaction] / cleaner goes through log segment and sees what inodes and data blocks are still being used, if still relevant then send to RAM to be written out in next log segment as though they were new changes, if not relevant mark as unused / disk is one big ring buffer [producer/consumer] where producer writes out new log segments periodically and consumer is cleaner that reads old segments and occasionally gives producer some new things to write out / file sys use **journaling** to recover from crashes and power outages, log struct file sys is its own journal, if power failure find last block tried to write to and fix it everything is fine / log struct good for flash drives, **wear leveling** built in [virtual layer still good idea] but now gremlins have less work / not greatest scheme if drive almost full with relevant data, end up reading almost entire segment into RAM and write back out to new log segment, takes longer and will wear out drive much sooner since everything get rewritten frequently [bc cleaner will have trouble staying ahead of segment writer causing need for recycling segments a lot] , do not let SSD fill past 75% / goback utility allow access past snapshots of file sys / don’t need much extra space to keep track of many versions of a file sys and about 1 second needed to create one of these versions / do this by never throwing out old **inode map**, just keep writing out any new stuff to next log segment / so goback just points you at an older inode map [new changes still put in new log seg ] , works until old inode is overwritten / with log structured writes are very efficient bc all contiguous / ghost: block that was kicked out of cache but later discovered to be useful / minimize ghosts with pair of SSD drives to form a second level cache [blocks that have been removed from RAM instead transferred to the SSDs] / since SSDs are almost as fast as RAM but hold a lot more data than RAM, cache of immediately available blocks expanded dramatically / if want to read something on edoras, OS first checks RAM, then **SSD cache**, if both fail then context switch to fetch from sinning disk drives / result: read latency is an order of magnitude better [eg avg read time might be twenty times faster than without the cache] // NFS: network file sys / clients can mount entire file sys from other hosts [servers] so that appear as part of local file sys / if server goes down, files on NFS temp unavailable, but when server reboots, things pick up where left off with no adverse consequences / stateless protocol: works without servers knowing what files are currently open, where file ptr curr is [so when server crashes, cannot lose such info bc not in charge of such info] / motivation of NFS was high cost of disk storage / if have bunch of identical hosts[same type of hardware, and OS , not much point in having all sys binaries replicated on each host, instead all executables mounted read only from central server / boot hardware for diskless workstations began by issuing call for help over the network to initiate tftp: [trivial file transfer protocol] of executable images it needed to begin booting / as part of boot proc, diskless client mount its entire file system from NFS sys [usually in two pieces: one mount for the common read-only binaries, another mount for data, such as usr home dir’s] / NFS protocol specifies format of requests clients send to servers and nature of replies servers send in response / two types requests: mounting and data / request to mount client asks for some dir name on server [using pathname server curr has partition mounted], server consults database [usually in /file/exports] to find out if dir is allowed to be exported, and if particular host is on approved list of clients, if req granted server returns file handle for client to use / file handle: magic cookie that client later presents to server each time requests data from server / under orig protocol host snooping on network could intercept magic cookie and gain unauthorized access to server, sol: Secure NFS uses encryption / RPCs [ remote procedure calls]: after NFS file sys mounted, client uses RPC’s to request info from server / RPC for attribute info, read, and write supported / no RPC for open or close / server only responds to requests, does not try to track state info / proc’s on the client that wish to read file must locally req the file to be opened but this req not relayed to server, the client NOT the server responsible for tracking state info / unix can mount native file sys, windows file sys, NFS file sys, and many other, whole things supposed to be presented as one coherent file sys tree: proc’s do not need know what file sys type underlies data / unix kernel implements **Virtual file sys** [VFS] layer to disguise complexity, typical sys call [like open] just presents request to VFS, which then responsible for sorting out issues for particular file sys type / in case of NFS file sys, NFS client code translates req to RPC calls to server / when a proc opens a file, how does OS know which type of sys will have to interact with it?, ans: open specifies relative path name, as path is parsed kernel notices when it passes through mounting point, and switchs to dealing with type of file sys / so as path descends into an NFS file sys, NFS client code takes over as it travels through mount point now req data from a dir on server rather than local client / if usr req 100 bytes from NFS mounted file, client anticipates further requests and actually asks for 8k of data from server [more efficient transfer large blocks of data] , client buffers extra data so can avoid future network requests / sim strat on writes, usr buffers 8k worth of data to write then sends to server, or if proc closes file then sends data immediately[even if buff not full] // CD-ROM file sys / write once format / follows one continuous spiral / injection molded media has pits and lands which when laser focused on them, reflect dimly or brightly allowing to be read as sequence of zero/one and one/zero translations / original molded cds could not be changed / CD-Recordable [CD-R] created bright/dim via embedded ink, blank disk all bright spots, high power laser darken areas, giving pattern of zeros and ones, but no good way to erase dark spots, so effectively write once[ie append only: data written in mult seesions by finding end of previously written data] / CD-RW where can now overwrite previous data mult times // ISO 9660 file sys / standard format every dev could handle grew out of high sierra standard / each logical sector contains 2k data portion and 100 bytes of error correction and overhead / ex 75 blocks encode one second music , skipping one min of music jumps 60\*75 blocks, if few bytes are bad from music file nearly undetectable by human ears, not much error correction needed for music cds[way different for data cds] / ISO 9660 leaves 16 undefined blocks at begging of spiral, any software disks have bootstrap prog embedded there, and BIOS booting seq looks here to boot the comp from inserted CD-ROM / next block is primary volume descriptor, containing various identification strings, but most importantly containing block addr of root dir, which is starting point for locating other files and dirs In volume / iso dir entry: dir entry length [1 byte], extended attribute record length [1 byte], location of file [8 bytes], file size [8 bytes], date and time [7 bytes], flags [1 byte], interleave [2 bytes], CD# [4 bytes], L (basename dot ext ; ver) [1 byte], file name [4-15 bytes], then padding / file described by: two nums: starting addr and size of file, works bc files stored contiguously, not in chunks over the disk / advantage of write once media where can plan layout exactly before writing anything / since dir be written before data files, means really have to know where everything goes, before start writing: all fields in dir must be calc before write the dir / every param representing a num in dir [such as starting addr and size of a file], are stored twice, once in little endian and once in big endian, so every architecture can read params easily / ISO 9660 is three standards in one: level1 limits files names to 8 chars plus three char extension [only format ms-dos can handle] ; level2 allows file names be longer [up to 31 chars] ; level3 allows files be non-contiguous [strange bc can easily arrange an write once data to be contiguous] ; level3 useful if have files that have repeated chunks of data, only have to write that common data once, then just arrange for several files to repeatedly ref same place on disk / each dir entry has sys use field to allow extensions to basic ISO standard [rock ridge for unix, and Joliet for Microsoft] / rock ridge allows each dir to have additional fields, specifying things such as where three time stamps unix expects are to be stored / file sys nodes specified to be soft links, each file and dir given rwx permissions for user, grp, and other / in addition to 8+3 name, each file can also have long name to handle any legal unix name / ISO 9660 limits dir depth to 8 lvls, rock ridge has fields to relocate a dir to diff place in file sys to work around this / sys and dev’s that have not been programmed to recognize the extensions in the sys use field [rock ridge non aware] just see a normal CD [since all other dir fields remain unchanged], thus data in a unix file can be copied to CD, but things like file owner, permissions , and other attributes stored in inode might be lost / rock ridge aware sys can copy unix file sys to CD, and inode info will be encoded into the sys use field of dir entries, CD can then be taken to another nix host and file sys will be faithfully reproduced / Joliet aware sys do something similar / CDs all same size and shape [per ISO 10149 standard], DVD’s same dim so player can handle both / to mass produce CD: start with blank glass master, powerful laser bruns pits at right places along spiral of data, master can then us ecreate a negative image mold, then used to make the clear plastic bottom part of CD, with bumpy lands and pits on upper surface, then sprayed with mirror coating to make more reflective, then label on top of that / pits and lands are thus equally reflective reading laser shines through clear side, so pits [which are near label side], protrude down toward the laser and beam reflected back down to photosensor, measures intensity of light that comes back / question how can laser tell a put from a land if equally reflective? Ans: pits are 20 microns closer, happens to be fourth of wavelength of infrared laser beam, laser light is coherent light [meaning light waves all move in lock step, batch of waves that hit pits travel half wavelength further [fourth + fourth = half] tan batch of waves that hit surrounding surface], makes crest of some waves coming back to photo sensor to match up with trough of some of the other reflected waves and they tend to cancel out [interfere destructively], so less light seen and thus photosensor detect presence of put / it is transitions from pits to lands and vice versa that encodes data [helps fight wobbly disk] where roughly no change means a zero and a change means a one / data follows continuous spiral, start reading near hole in center proceed outward, if edges get scratched wont effect data near center, if develop small crack likely near edge, recall dir goes near beginning putting it in safest place [if dir cannot be read CD is useless] / unlike hard disks, rotational speed not constant, instead want keep linear velocity constant [ex takes 120 cm of data digitally encode one second of sound, so CD rotate so 120cm travels under beam every second, therefore must rotate at 530 rpm near center, but only 200 rpm near edge / almost three fourths bits on data CD for error correction / meaning of data determined by ISO / DVDs dual layer have semi reflective layer half way through plastic depending on where focus laser doubling capacity // CD Recordables / make CD in bulk to keep cheap, CD-R tech changed this / instead of pits, have actual light and dark spots / have thin chemical layer [near label, just below reflective coating], which is transparent until heated by a laser strong enough to write on the disk. Heated dye then permanently becomes dark [write once] / CD-R blanks contain spiral groove to guide much cheaper writing laser / put wiggles in groove to solve maintaining correct linear velocity [writing laser keep track how fast wiggles went past and give feedback to cheaper moter] / reads at low power writes at high power / since write once material to be written pre assembled [create correct dir entries, written all in one stream] / variant of CD-R format allows put more data at end of previous writing session / issue: first writing session creates dir that cant change, older CD drive cannot see new stuff CD-R smart enough to know where end of data supposed to be, then looks past that for new version of dir / thus when write new session on CD must also write updated dir describing contents of entire cd / now transforms from write once to more appropriately append only // CD-Rewritables / CD-RW technology, now can overwrite / instead of dye, use alloy for reflective surface / in natural crystalline state is reflective [lands] / if blast with high power laser becomes amorphous and non reflective [pit] / if heat less violently will reform structure to highly reflective crystalline form / need three laser intensities: 1 low power read 2 medium power to remelt [erase] 3 high power [write] // Principles of IO Hardware // I/O Devices / two cats :block oriented – handles data in fixed sizes [chunks], chunks can be accessed (read or written) independently, these type of devs called block devs include things like disk drives / other: character dev – input or output a stream of data, cannot seek an individual pos, [ ex mouse is char dev, supplies input stream of bits, must collect data in real time] / two cats help parts of OS be dev independent (ex unix kernel seamlessly cobbles together: tape drives, floppy drives, hard disk drives, CDROMS, various other block dev’s in single file sys, software using file sys does not need know type storage dev, instead standard set of sys calls used to access the file sys, kernels dev drivers handle device dependent details // device controllers / OS dev drivers talk to dev controllers, which is logical (electronic) portion of IO dev that controls mechanical portion / generic IO controllers are standard parts of motherboard / but not always case, past: disk controller was sep daughter card, placed in ISA slot on motherboard, could control two disk drives / drives were connected to daughter via 40 wire cables (some for parallel data transfer, others for sending commands to physical drive to tell was info send over data lines) / drives respond controllers req and deliver stream of bits over data lines / impractical have CPU handle unpredictable stream, so one of controller jobs assemble received data into coherent block of rapidly accessible data [ie bits stored in mem on daughter card, examined to make sure complete and error free before sig CPU data is rdy to be delivered / in past: CPU supervise transfer of data in controller buff into main mem one byte at time [reason why SCSI drives more expensive that IDE drives] DMA erased most SCSI advantage / modern controller faster / when OS ask block, controller fetch that block and several adjacent blocks and caches them in own internal mem buffs, so when req come in disk controller first check block already in buff, if true ACKs req immediately / cost of reading data from disk drive is dominated by seek time (moving head of assembly), followed by rotational latency / actual transfer time per block miniscule fraction of these amounts // Memory mapped IO /def-each control register is assigned a unique memory address to which no memory is assigned. These addresses are usually at the top of the address space. These registers are just variables in memory and can be addressed in C, thus requiring no assembly code to do reads and writes. | Part of the address space does not refer to bytes of RAM in the usually way, but instead is mapped to various controller-card registers. In this scheme writing to some “high” RAM memory address effectively routes data to a register on some controller./ b) Why is programming simpler on memory-mapped I/O? ANS: The registers are just variables in memory and can be addressed in C, thus requiring no assembly code to do reads and writes. An IO Driver can be written entirely in C. Why does memory-mapped I/O lead to complications with modern architecture?ANS: On older systems, the memory and CPU shared a bus. Today, the I/O devices can’t see the memory references on the high speed bus and would be left in the dark without a special device called a bridge./ How does today’s advanced bus architecture complicate memory mapped I/O? ANS: On modern systems the CPU and memory have their own high speed bus and I/O devices cannot ‘see’ the memory references whizzing by on the high-bandwidth bus, and would be left in the dark unless extra hardware allow them to ‘see’ the relevant part of memory. /each controller card has on-board few reg that are used to interact with CPU, ex CPU wants contents of particular block to be read off a disk, puts description of which block into one of these reg, controller fills another of its control reg with proper completion code when op has finished / two convention allow CPU get at data in controller card registers / orig scheme conceptualize reg as array of integers in special IO space (ex suppose first five things in array associated with various control reg on floppy drive, next few associated with various control reg on first IDE controller etc, if CPU ask read from first of these integers (called ports) received data sitting in first reg with in floppy drive controller, if wrote to sixth port result data stuffed into one of reg on IDE controller / in 70’s some architectures switched to memory mapped IO, part of addr space did not refer to bytes of RAM in usual way, instead mapped to various controller-card reg / in this scheme writing to some high RAM mem addr effectively routed data to a reg on some controller / pro of mem mapped IO: drivers can be written in C (since specific mem addr for dev, C can easily access data by writing to appropriate mem location, in contrast, if special addr space for IO ports, CPU must have special machine instructions referencing them, so have us e assembly routine) / con: added complexity, in order controller get info CPU puts in mem locations relevant to controller, controller must watch all mem writes put on bus and when finds some in range it has been assigned, then responds to req, non issue if mem, CPU and IO dev all on single bus / but CPU and mem have own bus IO dev cannot see whats on that bus unless have extra hardware (bridge) / job of PCI bridge chip filter addresses rom CPU to RAM and pass relevant one to other buses / modern sys cache recently used mem addr in special reg array, good for speed bad for mem mapped IO need hardware to disable mem caching of parts of mem used to control IO devs / if caching were allowed for parts of mem used to control IO dev, defeats point having control reg allowing CPU check these locations to discover when dev has completed action, if CPU keep getting answer from cache would be unaware the dev has made change to actual mem location, must disable caching for this part of mem and force CPU check actual RAM // DMA: Direct Mem Access / Def- The DMA controller is programmed with a physical address within a page frame, and then the CPU goes off and attends to other matters, trusting the DMA to carry out the transfer once the data is assembled in its buffer./ What is the benefit of DMA? The CPU can do other tasks while the DMA controller handles a slow task like reading from disk. /How was disk access handled before DMA? The CPU oversaw the entire disk transfer byte by byte. / regardless of how CPU setup to access control registers of IO controllers, ultimately must get data from devs and store in main mem / on a sys where CPU busy, making CPU oversee this byte by byte transfer of large amounts of data big drag on performance / DMA, extra hardware called DMA controller , for IDE drives could be located on each controller card, or can have single DMA IDE controller located on mother board (along with IDE controller), the one DMA IDE controller can work with multiple IDE controllers (each of which can work with 2 drives) / without DMA controller, after CPU sends req for data directly to IDE controller, IDE controller sends appropriate translated req to actual disk drive, which sends data in stream to IDE controller, which assembles it in internal buffer, and verifies for errors once transfer complete (if error , as for data again, in fixed num attempts tells CPU op failed), if no error IDE controller causes interrupt alert CPU data rdy for delivery, interrupt handler read the data from IDE controller buffer one word at a time (assuming word is bus width), upon finish sig IDE controller so it knows can now proceed with next req and reuse buff / With DMA controller, CPU in addition to sending req to IDE controller also programs DMA controller with which controller it req read from, how much data ask for (count), and where in mem this block of data should go (address), DMA controller not told which disk drive or where on disk drive this info coming from, its task is take what is in particular controllers buff and stuff into mem / when disk controller assembled and verified block of data in internal buff, does not bother CPU instead send ack to DMA controller that it is rdy, DMA controller then coordinates transfer (like CPU in old way) / While CPU doing more important things DMA controller send req to IDE controller, repeatedly asking transfer a byte from its buffer to proper location in main mem until all bytes transferred, once completed (count in DMA controller equal zero), DMA controller sends interrupt to CPU to alert req data now in RAM / DMA useful if have busy sys, but if CPU idle then DMA actually slows things down / when interrupt handler starts up, it does not have to copy the disk block to memory; it is already there / issue: DMA controller programmed (address field) with physical addr within page frame, then CPU goes off and attends other matters, trusting DMA carry out transfer once data assembled in buffer, while DMA controller waiting for data arrive from physical disk, other proc’s are running, perhaps causing page faults, possible that page frame which DMA controller supposed to transfer data to is chosen for replacement, so frame overwritten with new page that some other proc req, DMA knows nothing about this and transfers buffer to physical address, thus vomiting over some innocent page / sol: must be way of locking page frames until data transfer op complete / in non-DMA scheme this non issue bc interrupt handler supervise transfer of data to physical page frame and retained control of CPU while doing this , therefore no chance of activity causing page faults // interrupts revisited / Peripherals alert CPU when finished by issuing interrupt / in past: CPU decided how handle interrupt: process it immediately, possibly interrupting some less important interrupt handler, or delaying ack til later time / modern sys insert interrupt controller between CPU and peripherals, lessening burdon on CPU, job to field all incoming interrupts and present most important one to CPU (by asserting interrupt sig on bus), interrupt controller puts unique num on buss address lines so CPU knows with IO dev needs attention, unique num is idx to table of interrupt vectors, so CPU can jump to appropriate interrupt handler routine / to run interrupt handler, CPU make room for new routine to run: proc that got interrupted was using CPU reg’s, must save these reg’s so interrupt handler can use, when handler runs, sends ack to interrupt controller: which knows not issue another interrupt until receives this sig (to ensure curr interrupt handler is in sane state before it can itself be interrupted) / when curr proc interrupted, context must be saved so can be resumed later, each proc has stack that could be used but most reliable way have sep kernel stack [eg stack ptr for a usr proc might not even be valid at moment], unlike usr stack [which can grow with out bound], kernel stack can be fixed size [ie well defined pecking order for interrupts so worst case scenario of max sequence of interrupts be computed and exactly right amount of space set aside for kernel stack / issue: **pipelining** / CPU has three stages: 1 instruction being fetched from mem, 2 during 1, prev instruction being decoded, 3 previously decoded instruction being executed this happened during 1 and 2 / in superscalar architecture instructions not necessarily executed in order originally specified / in most designs once instruction enters pipeline must be executed, consider: if x>0 then branch else add 1 to sum ; branch and addition both in pipeline, so regardless if branch will still add next even if not supposed to ; sol: instead of emptying pipeline each time, more efficient assume one outcome then back up and undo effects if assumption is wrong / old: typical model handling interrupts: after each intr finish exec, microprog or hardware check see if interrupt pending, if so PC and PSW were pushed onto stack and interrupt service begin, after interrupt handler rand, reverse proc took place with old PSW and PC popped from stack and prev proc cont; issue: not right for pipelining / for best performance handle interrupts as quickly as possible / 1st attempt: fnished instr curr exe, save state and jump to appropriate interrupt vector, saving sate of the interrupted proc involves saving PC ; issue: highly unlikely to be addr of instr that was supposed to come after one just finished: that next instr already in pipeline and being decoded, one before that already been fetched and so PC probably points to one that has come after that / once interrupt finishes, OS has do analysis of state machine figure out how begin refilling pipeline so interrupted proc restarts in correct place / superscalar architecture makes things worse (ex y++; x = -y; really needs be executed in correct order, as super scalar CPU may be decoding these at same time and first one rdy is first to be exe / sol: superscalar CPUs must be designed leave things in less of a mess using precise interrupts which follow: PC saved, all instruct before one pntd to by PC been fully exe, no instr beyond one pntd by PC been exe, execution state of instr pointd to by PC is known ; trade off is in order save OS from figuring out this mess, hardware on CPU chip must rationalize things, leaving less space for other things on chip // Principles of IO software / DMA used more than just for disk drives (ex CPU oversees sending data to a printer one char at time (interrupt driven IO) / a DMA controller can take over this job: CPU can fill a buff with data for the printer and DMA controller responsible for handling actual interrupts from printer and feeding chars one at a time, relieving CPU of this chore / with out DMA interrupt per char, with DMA interrupt per buff // Disk Hardware / num surfaces determines num track (track is: circular stream of data on one platter) / num cylinders: num concentric rings that armature heads can read / data in cylinder much cloer together than data on one surface / sectors are individual blocks on given track, each sector contains same num bits / bits near center are jammed closer together than ones around edge, limit how close together can be / old: to get something off IDE disk, had ask for specific cylinder, track, and sector / sol: needed virtual layer to do this (in real life more sectors on outside tracks than inside one, in virtual layer evenly distributed) / to do this IDE controller remains ignorant (thinks virtual layer is actual layer) and IDE drive is smart (it knows the diff) / IDE (integrate drive electronics) drive has its own microcontroller, which can respond to higher lvl commands (has gremlins) / old: 16 cylinders, 4 tracks, 6 sector nums which gives 512 bytes per sector (31.5 GB in total) / Logical Block Addressing (LBA) treats drive as though simple array of blocks / issue: IDE drive and IDE controller has to speak this lang else wont work (ex if hook up 500GB drive up to old controller can only access 1/10 of total space) / SCSI designed to be addressed as array of blocks and thus many advantages over IDE drives, such as could handle 15 devs at once (IDE drive only 2) // Disk Formatting / format of sector: preamble (bit pattern allow hardware recognize start of sector) ; data determined by low level formatting prog / ECC (error correcting code) (not just parirty bit, extra 100 bits to 16 bytes ) / lose disk capacity for reliability // Error handling / make new disk, test new disk for errors on sectors, if enough flaws that ECC cannot correct then considered bad sector, put on bad block list and remapped to space sector, / sol1: remap bad sector to good spare sector / issue: sectors may no longer be contiguous adding extra rotational delay to reach correct sector [1 2 3 bad 5 6 7 4 ] / sol2: rewrite remainder of track , reassigning sector num’s to be in order // issue: takes additional time (once) to do this but it min rotational delays [1 2 3 bad 4 5 6 7] / controller keeps track of changes , drives have special commands dump list of all remapped sectors, but since microcontroller handle this so usr and OS don’t need know / ie bad block mapping one more layer of abstraction over virtual geometry / if storing huge image fie, most efficient read sectors on one track consecutively, can happen all in one rotation / after last sector on track is read, bad: move head to next track on platter, good: switch to reading head on platter below this / when do switch some sectors already gone by, sol: offset sector 0 from one platter to next platter / after reading entire cylinder must move head to adjacent cylinder / issue skew factor, which is an attempt to min rotational latency in data that spans more than track / for random disk access once heads in proper place. Still wait between zero and one rev for data spin under read head / non random seq of disk access (consecutive) big cut down rotational latency / **rotational latency** – time it takes for ½ a revolution ->rps -> multiply both num and denom to get rotation to (½)// Clocks / are timers /**soft timer** - Is a time-ordered list of scheduled events that is checked every time the kernel exits kernel mode, This gives a compromise between polling and interrupts. // Clock Hardware / old: track AC current oscillations (60 Hz) / issue: imprecise / sol: crystal vibrations / 3 main parts of clock: counter, holding reg, and crystal oscillator / decrement counter with each pulse of crystal, interrupt generated when reach zero / to begin counting, holding reg provide int as starting point, this value transferred to counter, then coutns down to zero / clocks op on square wave model / alternate mode: one-shot mode, holding val loading holding reg, transferred to counter, when zero time interrupt occurs clock does nothing further until explicitly programmed with another holding val / watch dog timers use one-shot mode / (ex floppy drives only spin when necessary, take time to speed up, no way of signaling CPU that rdy, sol: set timer in future that stops and lets CPU know floppy is up to speed / square wave mod ecan keep track of time of day / unix time is kept counting num seconds since jan 1 1970 (two phase proc, count clock ticks on one accumulator until full second gone by/ at which point add one to accumulator that is tracking seconds / comp on, sys does this, comp off internal battery powers small hardware that does this, when comp back on sys uses this until can sync with atomic clock over internet // clock software six major functions of clock driver: 1 maintaining time of day 2 calc end of each proc curr time slice 3 accounting for CPU usage 4 fielding alarm sys call 5 maintain watch dog timers 6 gathering statistics, profiling, and monitoring / 1. Unix counts seconds since jan 1 1970, storing result in signed 32 bit int (so can access time before jan 1) (milestones: 1 billion s, 1234567890 s) overflow in 2038 / sol: unsigned rather than signed, or store time in 64 bit sign / 2 calc end of each proc time slice, when rdy proc chosen by scheduler to run, reg representing duration of time slice is set, each clock interrupt causes counter decrement when reach zero (if proc not already kicked) scheduler put new proc on / 3 accounting for CPU usage, most precise way implement usr CPU time have reg associated with reg that is init zero when proc is created, and incr by one each clock tick, when context switch or interrupt reg is saved along with other reg so not get charged with time when not using CPU / other sol: (more common less accurate but less resource intensive) have field in on proc table for each process , nut now get charged during clock interrupt / fielding alarm sys call - alarm: set alarm clock for delivery of sig (SIGALRM) / proc can req warning after certain interval / proc implementing network protocols use a lot (ex browser req web page, does not wait forever bc alarm) / cannot have enough hardware clocks to solve this so must use virtual clocks, where pending clock req put in linked list, sorted so ones happening first put in front of list / 5 watch dog timers / timers for OS / kernel cant be delivered sigs like normal proc, instead use watchdog to interrupt / when goes off associated with entry point of a procedure which is then exe in kernel mode / 6 gathering statistic, profiling, and monitoring / profil – execution time prfile – prvides CPU-use stats by profiling amount of CPU expended by prog (creates a histogram) / every OS has this bc infeasible to check PC on every instr so instead check once in while (ie on every clock interrupt) //TOTAL WRITE TIME = seek time + rotational latency + (#bytes / transfer rate(bps)) / average overhead = TLB \* h + (1-h) \* reading from page table. / Faster RPM – faster latency / **Dup2** duplicates one file descriptor over another. (For redirection).-We need to close all the pipe ends in all of the places because once the parent forks, it AND its children has open file descriptors to the pipe. If the process tries to read from the pipe, and there’s nothing to read, the pipe will return EOF. / Name a system call that will cause the [non-writable] text area of a process to suddenly be different. -When you execvp() all the code from the parent is dumped and replaced with whatever program is execed. The text area is obviously changed because it must contain the executable for the new process. /Old cd drives assume the cd-r is only written to once New formats can continue writing incrementally post previously written data, but old drives do not know to check past the first set of data for new data. / Reads would be faster: buffer cache stores recent used blocks in Ram/ double the linear recording density then the drive capacity and transfer rates are doubled. The seek time and average rotational delay are the same. / standard output and standard error - separate, so standard output can be redirected without affecting standard error. In a pipeline, standard output may go to another process, but standard error still writes on the terminal. / Can a page fault ever lead to the faulting process being terminated ANS: It is possible. Ex) when the stack grows beyond the bottom page, a page fault occurs and the operating system normally assigns the next / Is it possible that with the buddy system of memory management it ever occurs that two adjacent blocks of free memory of the same size co-exist without being merged into one block? If so, explain how. If not, show that it is impossible. ANS: It is possible if the two blocks are not buddies. Two new requests come in for eight pages each. At this point the bottom 32 pages of memory are owned by four different users, each with eight pages. Now users 1 and 2 release their pages, but users 0 and 3 hold theirs. This yields a situation with eight pages used, eight pages free, eight pages free, and eight pages used. We have two adjacent blocks of equal size that cannot be merged because they are not buddies / Both Fred and Lisa have access to the file x in their respective directories after linking. Is this access completely symmetrical in the sense that anything one of them can do with it the other one can too? ANS: No. The file still has only one owner. If, for example, only the owner can write on the file, the other party cannot do so. Linking a file into your directory does not suddenly give you any rights you did not have before.