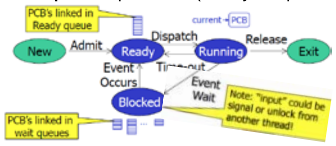
Jason Pham
phamhoa5
1003078144
CSC369 Aid Sheet

General

OS: (1) *virtual machine* - convenient for the user by simplifying hardware interfaces to APIs (2) *resource allocator* - proper and efficient use of (3) *control program* - protect, secure, authorize (esp. I/O devices)
Roles: synchronization, scheduling, memory management, inter-process communication, exception handling, FS, device drivers, networking
Themes: virtualization, concurrency, persistence
Limited direct execution: mode bit is set on privileged instructions: enable/disable interrupts, writing to devices, performing DMA, halting the CPU.
Interrupts caused by H/W or S/W trap or exception, signals error or OS intervention, CPU jumps to pre-defined interrupt handler routine. OS is event-driven. When OS done: returns to app, set-up registers, MMU, mode, and jumps to next application instruction
Hardware interrupts allow device controller to signal CPU of event, jumping to pre-defined interrupt handler
Software interrupts are signals or requests for OS services from a user program (traps/exceptions).
Interrupt steps: (1) OS fills interrupt table at boot time (2) CPU exec loop: fetch instruction at PC, decode, execute (3) Interrupt occurs (4) CPU change mode, disables interrupt (5) PC value saved (6) IDTR + interrupt number used to set PC to interrupt handler (7) execution continues (saves addition state as step 1)
Bootstrapping: H/W stores small program in non-volatile memory. When power supplied, program executes, loading OS into memory

Process:

an OS abstraction for program in execution.
PCB: state, PC, priority, registers, page tables, open files/networks & resource use info, kernel context
fork(): creates PCB, new address space, copy parent stack, initializes kernel resources (e.g. open files), puts PCB to ready queue. Returns PID to parent, 0 to child.
exec(char *prog, char *argv[]): stop current process, load *prog* into address space, initialize H/W context, put PCB on ready queue (no new process created).
Processes are **created by** fork(), **transformed** into other programs by exec(), **controlled** by system call interrupts/state changes, **deleted** by exit() and signals, turning PID to a **zombie** that must be wait()'ed for.
System call: A function call that invokes the OS.
In C: **`syscall(syscall_no, arg1, …, arg6)`**
The register %eax always contains the syscall number.
System call dispatch: kernel gives numbers to calls, kernel initializes syscall table, user process sets up num and args, user runs int 0x80, H/W switches to kernel mode, invoking kernel interrupt handler, kernel looks up table, invokes function, returns by running iret
State queue of processes (usually one per state).


PCB's linked in Ready queue
current
PCB
New
Admit
Ready
Dispatch
Running
Release
Exit
Event Occurs
Time-out
Event Wait
Blocked
Note: "input" could be signal or unlock from another thread!
PCB's linked in wait queues

Block reasons: lock, I/O, sleep, signal, quanta expiry
Context switch: switch CPU to another process by: saving state of old process, loading saved state for new process (context switch time is pure overhead). Happens if (1) yield() called, (2) makes system calls and is blocked, (3) timer interrupt handler does switch
Kernel module: insmod to load, rmmod to unload.
In C: module_init(myinit); module_exit(myexit);

Concurrency

Thread is a control flow with its own stack frame
Kernel threads can share address space to reduce context switches; are general to support all languages
User threads (100x faster) represented by small TCB, creation through procedure calls and invisible to OS and controlled by runtime (user-level) library.
Critical section piece of code using shared resource that must be accessed by only one thread at a time
Race condition is when two threads try to manipulate a shared resource w/o synchronization, making the outcome depend on order in which the accesses occur
Properties of solutions: > mutual exclusion
> progress: only waiting threads can influence which thread enters next; choice cannot be delayed forever
> **bounded waiting**: no starvation
Software solutions:
> Dekker's algorithm: int turn /*0 or 1*/; int f[2]={0,0}; work(i){f[i]=1;turn=i;while(turn==i && f[1-i]);/*cs*/ f[i]=0;}
> Petersen's algorithm: Dekker's for N processes.
> Bakery algorithm: upon entering, each thread gets # and thread with lowest # served next. tied #'s broken by thread ID, which is unique and totally ordered
Hardware support:
> **uniprocessor:** disable interrupt, multiproc cannot
> **TestAndSet(*m) {bool old=*m; *m=1; return old;}**
bool m=0; lock(*m){while(TAS(m));} unlock(*m){*m=0;}
Or, using yield: lock(*m) { while (TAS(m)) yield(); }
> **Swap(*a,*b):**
bool m=0;… key=1;while(k)swap(m,k);/*cs*/swap(m,k);
> **CompAndSwap(*m,a,b)**{int r=*m;if(r==a)*m=b;ret r;}
bool m=0; lock(m){while(cas(m,0,1)==1) /*spin*/;}
> **FetchAndAdd**(*v) { int old=*v; *v=old+1; return old;}
ticket = turn = 0; lock(m) { myturn = FAA(m->ticket); while(m->turn != myturn);//spin} unlock(m){m->turn++;}
> Machine instruction problems: busy waiting, starvation is possible, deadlock via priority inversion

Semaphores

Wait (P) decrements —, block until sem is free
Signal (V, a.k.a. post) increments ++. unblocks thread both P and V need mutex/spinlock to implement.
Implement mutex: initialize cnt = 1; P(); /*cs*/ V();
Implement semaphore with mutex: int c=0; mutex m;
  P() { lock(m); ++c; unlock(m); }
  V() { lock(m); while(c<=0) {unlock(m);sleep();lock(m);}
        --c; unlock(m);

Condition Variables

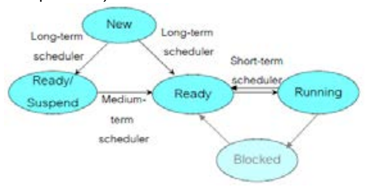
ADT encapsulates pattern of "release mutex, sleep, reaquire mutex" with internal queue of waiting threads.
- **cv_wait(cv, m)** – unlocks m, waits, locks m b4 return
- cv_signal(cv) – wake one enqueued thread
- cv_broadcast(cv) – wakes all enqueued threads
Unlike V(), if none waiting then signal/bc has no effect.
Usage: lock(m); while(condition not true){wait(cv,m);} /* do stuff */ signal(cv) /* or broadcast(cv) */; unlock(m);
Implement semaphore with lock+CV: init(v){val=v;}
P(){lock(m); while(val<=0)wait(cv,m); val--; unlock(m);}
V(){lock(m); val++; signal(cv); unlock(m);}

Monitors

ADT with restriction that only one process can be active at a time within the monitor. > enforces mutex
> local data accessed by only monitor procedures
> process enters by invoking one of its procedure
> other processes' attempts at entering are blocked
Hoare: signal() immediate switches from caller to waiting thread. Condition that waiter was blocked on is guaranteed to hold when waiter resumes. Need another queue for the signaler if signaler was not done using the monitor. Usage: if (empty) wait(condition);
Mesa: signal() places waiter on ready queue, but signaler continues inside monitor. Condition not necessarily true when waiter resumes; must check condition again. Usage: while (empty) wait(condition);
Trade-offs: Hoare makes it easier to reason about program, but Mesa easier to implement, more efficient, and supports additional operations like broadcast().
Monitor made up of: - initialization code
- one lock: for access to the monitor
- local data: here, buffer, inpos, cnt
- **condition vars:** here, nonfull and nonempty
- conditions 1..m: here, "cnt < SZ" and "cnt > 0"
- procedure 1..n: here, add() and remove()
General steps in any function: (1) acquire lock. (2) call wait() inside while loop (Mesa). (3) whenever condition waited on might've changed from F to T, signal corresponding condition variable. Signaller need not know prev state, only that curr is T. (4) release lock

Scheduling

Goals (all systems): (1) fairness - threads get fair share of CPU
(2) avoid starvation (3) policy enforcement – usage policies should be met (4) balance - all parts of system should be busy
Goals (batch systems): (1) throughput - maximize jobs completed per hour (2) turnaround time – minimize time between submission and completion (3) CPU utilization – keep CPU busy all times
Goals (interactive systems): (1) response time – minimize time between receiving request and *starting* to produce output (2) proportionality – "simple" tasks complete quickly
Goals (real-time systems): (1) meet deadlines (2) predictability
When to schedule: (1) I/O interrupts, (2) signals, (3) thread creation, (4) fixed signals
Long-term scheduling (admission control): in batch systems, not common today. **Medium-term:** happens infrequently, decides which process swapped to disk.
Short-term (aka dispatching): happens quickly, needs to be efficient (fast context switches, fast queue manipulations).



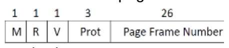
On dispatch: (1) select next thread from ready queue, (2) save currently running thread state (unless current thread is exiting) (3) restore state of next thread (register, OS control structs, switch to user mode if needed, set PC to next instruction in thread)
When to schedule: (1) thread enters *Ready* state due to thread creation (*admission* in batch systems), I/O interrupts, signals) (2) thread blocks (or exits) due to system call or signals (3) at fixed intervals
Scheduling policies:
Pre-emptive: Forces a context switch upon events (quanta expires, higher priority process unblocks, etc.) Suitable for real-time, time-shared, interactive systems
Non-pre-emptive: waits for running process to yield CPU (blocking/exiting) before scheduling new process Suitable for batch, single-app, some real-time systems
> **first come first serve** (non-pre) uses ready queue; **convoy effect**: small threads wait for big, OS is slow
> shortest-job-first (non-pre) pick job causing least wait time for others. Optimal w.r.t avg wait time.
> shortest remaining time: (pre-empt version of SJF)
> **round robin** (pre): keeps circular ready-queue. Each thread runs for time quantum q before pre-empted and moved to back. As q→inf, RR→FCFS, As q→0, RR→processor sharing. Want q large w.r.t. context switch time.
> **priority scheduling** (can be pre or non-pre) priority p given to each thread. Max(p) job selected from ready queue. Risk of starvation and **priority-inversion** (low priority job keeps resource from high priority job).
> feedback scheduling: Adjust criteria for picking a thread based on past history. **Aging** (boost priority of older processes to avoid starvation); prefer threads who do not use full quantum; boost priority after input
> multi-level feedback queue: combination of above. Distinct queues, each assigned different priority level. Each queue has multiple ready-to-run jobs. The scheduler always chooses to run jobs in queue with highest priority. Jobs start in the highest priority queue.
feedback: If a job uses an entire time slice, its priority gets reduced. If gives up CPU, remains same priority.
> fair sharing scheduling: group threads, ensuring each group gets fair share of CPU. Priority of thread depends on its own priority and past history of group.
lottery variant: Each group is assigned tickets based on share. Hold lotteries to find next thread to run.
> Unix: reduces to RR for CPU-bound jobs; more CPU accumulated → lower priority (negative feedback)

Memory Management

[Fast,$$$]Registers,L1/L2 Cache, DRAM, Disk[Slow,$]
Virtual memory: each process needs to think it has its own privatized physical memory
Goals: (1) Efficiency – transfer to/from DRAM and disk (2) Transparency – illusion of more memory in DRAM (3) Protection - should not access memory of OS or other processes, except when sharing is desired
Fixed partitioning divides memory into regions with specific and usually equal bounds.
Dynamic partitioning: execution time binding of addresses; similar to fixed but bounds are set on by-process basis using max process size determined at load time. Uses "base" and "bound" registers.
Placement heuristics: some algorithm to manage freelist: first bit, best-fit, worst-fit, quick-fit
Fragmentation: Internal fragmentation occurs in fixed partitioning; sometimes process is not as large as the process allocated to it. External fragmentation occurs in dynamic partitioning when process is removed and the hole remaining is too small for any process to use.

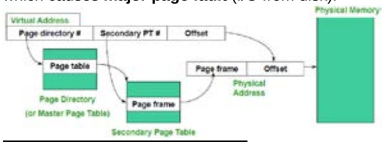
Paging

Each fixed sized (usually 4K) virtual page is mapped to a physical page frame. Hardware **MMU** converts VA into PA using the OS-managed **page table**. If page not in memory according to the table, causes a **page fault**. Then, the OS must load page from disk. OS policies take advantage of temporal and spatial locality.
Advantages: Fills physical frames without needing to be contiguous blocks. Paging reduces external frag & limits internal frag to at most one page per process.
Protection is enforced by VM, as we avoid leaked info from deallocated pages. OS ensures that newly alloc'd pages are "zero-ed" out with copy-on-write.
Translation requirements: (1) Relocation to different region of memory due to swapping processes in/out (2) logical organization – to map between 1D machine byte array and programmer's code module (3) physical organization – flow of data between DRAM/disk levels
Page info: Assuming page size is 2^x bytes and addresses are y bits. The offset is x number of bits & the page # will be the rest of the bits (y – x bits). To calculate the maximum number of pages use 2^y.
Example: Addresses are 16 bit. Pages are 1024 bytes. Offset = 10 bits. Page # = 6 bits. Max # pages = 64.
Page table entry: maps vaddr to PFN;



stored in OS memory, attached to process structure where it is protected and user process can't access.
(M)odify/dirty bit: set when write occurs. (**R)eference bit**: set when a read/write occurs. (**V)alid bit**: Says if a PTE can be used; checked on each use of vaddr.
(Prot)ection: Marks whether r/w/x allowed on page.
Memory overhead solutions: bigger pages? (no, wasted space) instead try (1) hierarchical page tables (2) hashed page tables (3) inverted page tables
Multi-level page tables: Virtual addresses 3 parts: master page #, 2ndary page #, and offset. Master table maps VAs to 2ndary page table. 2ndary table maps page # to physical frames. Offset selects byte in frame.
Hashed page tables: Hash fn() maps VPN to bucket in fixed-size table; search that bucket for queried VPN
Inverted page tables: 1 page table for system; 1 entry per physical page frame; entries record which virtual page # stored in frame and process ID. Less space but slower lookups. Can use hashing to increase speed.
Demand paging is when pages moves DRAM<->disk, transparent to program. Evicted pages go to **swap file**.
Page faults: Evicted page accessed. (1) On eviction, OS sets PTE invalid and stores location of the page in the swap file in the PTE. (2) If process accesses page, invalid PTE causes trap (page fault) which (3) run the OS page fault handler which (4) uses invalid PTE to locate page in swap file. (5) reads page into physical frame, updates PTE to point to it (4) resumes process.
Where put? Each eviction is slow! OS keeps free pages pool around so allocations don't always evict.

Translation lookaside buffers is hardware that translates virtual page # into PTE in a single machine cycle.
Who places translations in TLB? Hardware MMU (e.g. Pentium) knowing PTBR; or OS (MIPS R2000).
TLB on process context switch: entries invalidated, or old PTE must be evicted on TLB miss using policy.
Summary of cases: (1) TLB lookup in MMU. If hit, MMU reads physical address data to CPU. All done in H/W. (2) If TLB miss, then there is a **minor page fault** (no I/O needed) either (a) MMU loads PTE from page table in memory. H/W manages this step since OS setups the page table so H/W can access it directly. (b) trap to OS which loads page table's PTE into TLB (software step). (3) If PTE has protection fault, traps to OS (software) where either (a) r/w/x error - OS reports fault, or it might be protecting for CoW/mapped files. (b) invalid – seg fault, or page not in physical memory which causes **major page fault** (I/O from disk).

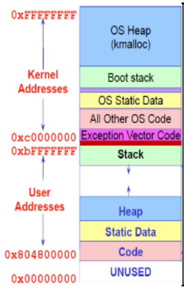


Advanced Memory Management

Thrashing: when more time spent by OS in paging data back and forth from disk than executing program, causing system to be overcommitted or oversubscribed. No time doing useful work.
Solutions: (1) swapping – write out all pages and suspend it (2) OOM killer daemon (3) buy more RAM
CPU Utilization: % of time CPU busy. This is high for compute-intensive processes, low for I/O heavy processes. If low, should we increase the degree of multiprogramming? No: typically decreases CPU util since less memory available memory per program => higher page fault rate. Decrease multiprogramming.
Fixed space allocation: each process given limit of pages, when limit reached, replaces from own pages (local replacement) **Variable spaced:** set of pages grows and shrinks dynamically (global replacement)
Working set WS(t,Δ) of a process is set of pages P such that P was referenced in time interval (t, t - Δ). Want working set to be in memory to avoid faulting.
Sharing: Use shared memory to allow processes to share data. Have PTEs in both tables map to same physical frame. Each PTE can have diff protection values. Must update both PTEs when page is invalid.
Copy on write: When parent forks, don't copy data immediately, copy when child tries to write in hopes of avoiding any copying. Instead of copy, create shared map of parent pages in child's virtual address space.
Mapped files enable process to do file I/O using loads and stores. Bind a file to a virtual mem region (PTE maps virtual address to page frame holding file data). Initially all pages mapped are invalid → OS read page from file when invalid page accessed, write when file evicted unless not dirty.
Page buffering - Most of these algorithms are too costly to run on every page fault. Instead:
> Maintain a pool of free pages (free pages list).
> Run replacement algorithm if pool gets too small.
> Free enough pages all at once to replenish pool.
> On page fault, grab a frame from the free list.

Page Replacement Algorithms

First-In-First-Out (FIFO) - Good because oldest page isn't used anymore, bad because maybe it is. Suffers from "**Belady's anomaly**" where the fault rate may actually increase given more memory.
Random: Pick random page to evict.
Least-recently used: Evict page used longest time ago. Can be costly (need to either time stamp every reference or use stack). Doing true LRU is too costly.
Second chance algorithm (Clock) – Approximates LRU with FIFO, but inspect reference bit when page is selected. If bit = 1, then set to 0, reset arrival time to current time. If bit = 0, replace the page. If page is used again, then set bit = 1 again. Pages used often enough to keep bit set won't be replaced.
Least frequently-used (LFU): Heavily used pages stick around even when not needed. **MFU:** favors new pages. **LFU** and **MFU** are uncommon and bad.
Belady's algorithm (OPT/MIN): replace page that will be accessed furthest in the future. Leads to fewest number of misses overall.



```
    0 | 0000 |     1
    1 | 0001 |     2
    2 | 0010 |     4
    3 | 0011 |     8
    4 | 0100 |    16
    5 | 0101 |    32
    6 | 0110 |    64
    7 | 0111 |   128
    8 | 1000 |   256
    9 | 1001 |   512
 A 10 | 1010 |  1024
 B 11 | 1011 |  2048
 C 12 | 1100 |  4096
 D 13 | 1101 |  8192
 E 14 | 1110 | 16384
 F 15 | 1111 | 32768
```

Files Systems

There are two tables to keep track of open files: a system-wide table and per-process table (which points to the system-wide table). Supported file operations: 1) creating 2) writing 3) reading 4) deleting 5) seeking within a file 6) truncating (erase parts; keep attributes**)**

Access methods: (1) Sequential Access: byte-by-byte; in order. (2) Direct access: given block/byte #.

Directory Benefits: File organization for users. Convenient naming interface for OS and logical file placement. For all: store file info.

Inode: Data structure representing a FS object (file, directory, etc.). Contains metadata: attributes, data block pointers, but no filename (stored in dir_entries)**.**

```
    struct ext2_inode {
      unsigned short i_mode;  // file mode
      unsigned short i_uid;      // low 16 bits of owner uid
      unsigned int     i_size;     // size in bytes
      unsigned int     i_atime, i_ctime, i_mtime, i_dtime;
      unsigned short i_gid;      // low 16 bits of group id
      unsigned short i_links_count;
      unsigned int     i_blocks; // # blocks in disk sectors
      unsigned int     i_flags, osd1;  // flags/os dependent
      unsigned int     i_blocks[15];
      unsigned int     i_file_acl,i_dir_acl,i_faddr,extra[3];
    };
```

i_block[1..12] are direct block points. [13] is singly-indirect pointer. [14] and [15] are doubly and triply. Data blocks store data if inode is regular file, dir_entry if inode is a directory, path name if inode is a symlink.

Directory: Linked list of (name, inode) mappings:

```
    struct ext2_dir_entry {
      unsigned int       inode;    // inode number, root is 2
      unsigned short   rec_len;   // dir_entry size in bytes
      unsigned char    name_len;  // length of name
      unsigned char    file_type;
      char                   name[]; //up to EXT2_NAME_LEN
    };
```

Hard links: multiple file names mapped to the same inode, only remove inode when i_links_count == 0.

Soft (symbolic) links: pointer to file, containing path.

Path name translation of "/somefile":
(1) use superblock to find inode for "/" and read inode into memory. inode lets us to find data block for dir entry "/" (2) Read "/", search list for "somefile" giving location of inode for "somefile" (3) Read inode into memory; The inodes says where first data block is on disk (4) Read block into memory to access data in file.

File Systems (FS Layouts)

VSFS: 64 blocks of size 4KB. Total disk size: 256KB. One superblock, one inode bitmap block, one data bitmap block, five inode blocks, 56 data block.

E.g address of inode 32: 12KB + 32*128B = 16K.

Types:

> Linear list: simple list of file names and pointers to data blocks. Linear search to find entries. Easy to implement, but slow (and not really used).

> Hash table: Hash filename to pointer to list entry.

Layouts:

> Contiguous allocation: Similar to memory; Inflexible; eventual fragmentation and needs compaction. (Good for small chips).

> Linked/chain structure: Each block points to the next, directory points to first block. Good for sequential access, bad for everything else.

> Indexed Structure: An index block contains points to many blocks. Handles random better; still good for sequential. May need multiple index blocks (linked together).

Multi-level pointers: To support bigger files with 15 blocks, instead of pointing to user data, we point to block containing more pointers to data.

Extent-based: extent is a pointer plus length in blocks. Instead of pointer to every block, just need pointer to every several blocks (every extent). **Pros***:* less metadata per file, and file allocation is more compact. **Cons**: less flexible than pointer approach.

Link-based: inode has just one pointer to first data block which points to other blocks. **Pros**: in-memory file allocation table (FAT) to find blocks faster. **Cons:** works poorly for accessing the last block of a big file.

File Systems (Protection)

Protection system checks if a given Action performed by a given Person on a given File should be allowed.
> Implemented by flags: [r]ead; [w]rite; and e[x]ecute.

Access Control Lists (ACL) maintain a list of users and their permitted actions PER file. Pros: Object-centric; easy to grant/revoke. Cons: Issues with heavily shared objects (use groups).

Capabilities List for each user (group): maintain a list of objects and their permitted actions. Pro: easier to transfer. Cons: To revoke permissions, have to keep track of all users (hard).

Files Systems (File Buffer Cache)

System wide cache of inodes, dir_entries, disk blocks for "hot" files (even whole file if small). Cache used and shared by all processes. Reading from cache makes a disk perform like memory.

Static partitioning: allocate fixed-size cache at boot in memory (~10% of total memory), used in early FS.

Dynamic partitioning: integrate virtual pages and FS pages into a unified page cache, so pages can be flexibly allocated for either memory or file. Used by modern systems. *Policy:* typically LRU.

Read-ahead: via predicting next block; take advantage of locality and compliment on-disk cache. Big-win for sequentially accessed files unless blocks scattered across disk (usually prevented in file system allocation)

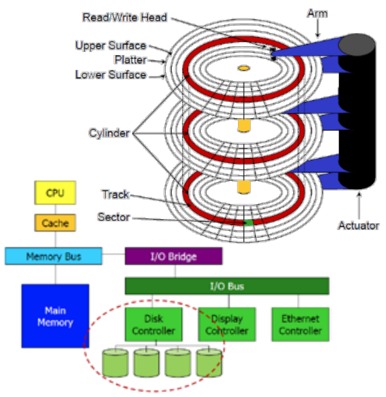
Trade-off vs. VM: competes with VM, replacement algorithms needed (LRU usually used). limited size**.**

Buffering: Combine multiple writes into one (e.g. updating multiple bits of inode bitmap). Write back to disk may not be *synchronous.* Can schedule lazy updates to be sequential, and avoid some writes.

Buffering approaches: (1) Delayed writes: how long? (2) asynchronous "**write-behind**" (maintain queue of uncommitted writes), periodically flush, but unreliable (3) write-behind with queue in battery backed-up NVRAM for reliability; expensive (4) log-structured FS to write contiguously at the end of the previous write.

Trade-off: Speed vs. durability. Buffered data lost during crash. But syncing more often => worse speed.

Physical Disk Structure and Optimization



Seek: moving disk arm to correct cylinder (1-15ms) depending on distance (avg 5-6ms), slow improving

Rotation: waiting for sector to rotate under head, depends on rotation rate of disk (rpm). Avg latency of half rotation (~4ms for 7200rpm SATA). Not changed.

Transfer: transferring data from surface to disk controller/host; ~100MB/s, avg sector transfer time ~5us. depends rapidly improving density (~40%/year).

Track skew: If arm moves to outer track too slowly, may miss sector and have to wait for a whole rotation. Skew track locations so we have time to re-position.

Zones: outer tracks larger by area; holds more sectors

Track buffer: Small 8-16MB chip part of HDD. Unlike OS cache, aware of geometry. On sector read, may cache whole track to speed up future reads on track.

Software interface layers: Each layer abstracts: [Program] <File name, offset> [File system (database)] <Partition, Block #> [Device Driver] <Disk #, Sector #> [I/O Controller] <Cylinder, Track, Sector> [Disk Media]

Disk Interaction: Older disks require OS to know all this. Modern disk provides SCSI – exported as a logical array of blocks [0..N] mapped to cylinder, surface, track, sector. Only need to specify the logical block # to read/write. But parameters hidden from OS.

Goals: (1) *Closeness* of related data: reduce seeking, benefits in factor of 2 range. (2) *amortization* of delays by grabbing lots of useful data; factor of 10 benefits.

Strategies: (1) Scheduling requests to reduce seek time. (2) Allocating related data "close together".

Disk Scheduling

FCFS - Reasonable when load is low.

SSTF (Shortest seek time first) – Minimizes arm movement, maximizes request rate, favours middle blocks, can lead to starvation.

SCAN (elevator) - Read one direction then reverse.

C-SCAN (typewriter) - SCAN, but one direction for fairness to fewer number of sectors on the inside.

LOOK/C-LOOK - Like SCAN/C-SCAN but only goes as far as last request in each direction (as opposed to hitting beginning and end of disk).

In general: Unless there are request queues, scheduling has low impact. Important for server, so-so for PC. Modern disks do their own scheduling.

Fast File System (FFS)

Problem with original Unix FS: poor utilization of disk bandwidth; aging files systems have data blocks scattered across disk. Fragmentation causes seeking. Also going back from inodes to data blocks causes seeks in path traversal, manipulating files/directories

FFS - Inodes: fixed size structure stored in cylinder groups. Metadata updates are synchronous ops. FS operations affect multiple metadata blocks:
- Write newly allocated inode to disk before its name is entered in directory.
- Remove directory name before inode is deallocated.
- Write deallocated inode to disk before its blocks are placed into the cylinder group free list. If server crashes between any of these synchronous operations, then file system is in an inconsistent state.

Solution - fsck: post crash recovery scan of FS structure and restore consistency.
- All blocks pointed to by inode or indirect block must be marked allocated in bitmap.
- All allocated inodes must be in some directory entry
- inode link count must match

Pros: Fixed low bandwidth utilization, and larger max file size (function of increased block size 1K→4K). Replicates the superblock avoiding media failures. Parameterize according to device characteristics.

Cons: Internal fragmentation (fix: allow 1K fragments)

Log-Structured File System (LFS) and Redundancy

Traditional FS changes in place: reduce seeks, avoids fragmenting, keep locality → reads perform well.

LFS: Reads will hit in memory as it becomes cheaper. Assume writes are bigger I/O penalty: treat as circular log. **Pros:** improved write throughput, simpler crash recovery. **Cons:** Memory might not grow, so reads may become much slower. Garbage collection tricky.

LFS writes FS data in continuous log. Each segment has a summary block containing pointer to next, along with imap and inode. Need a fresh block? Clean existing partially used segments (garbage collection).

Finding inodes: Not easy: inodes scattered over disk (1) Read the superblock ("checkpoint region") to find index file. (2) Read index file (linear search on inode blocks). (3) Use the disk address in inode to read the block of index file containing the inode-map (imap). (4) Get the file's inode. (5) Use inode as usual.

Metadata journaling "write-ahead logging":
When doing update, write down what you plan to do somewhere else on disk. Upon crash, go back to journal and retry writes. If crash happens before journal write finishes, not a problem since write has not happened, so nothing is inconsistent. Journal entry:

TxBegin (TID=1)	Updated inode	Updated Bitmap	Updated Data block	TxEnd (TID=1)

(1) Write all but TxEnd. (2) Write data (3) Write TxEnd.

Comparison vs. fsck: fsck is slow, must scan entire disk to recover small errors. Fsck only ensure integrity, cannot do anything about lost data.

Redundant Array of Independent Disks (RAID):

> Redundant/mirroring: Keep multiple copies of same block on different drives in case one drive fails.

> Parity info: XOR each bit from 2 drives, store the checksum in third drive.

Solid State Disks (SSD): Battery backed, NAND flash

Pros: faster; can erase! **Cons:** Expensive, wear-out.

> Wear-levelling by always writing to new location

> **Garbage-collection** by reclaiming stale pages

Deadlocks

Common non-deadlock bugs: atomicity violation (no mutex enforced in region) and order-violation bugs (desired order between memory access flipped)

Deadlock definition: Permanent *mutual* blocking of a set of processes or threads that (1) compete for resources, or (2) communicate with each other

Conditions: (1) mutual exclusion (2) hold & wait: Process may hold allocated resources while waiting. (3) No pre-emption: Resources cannot be removed from process holding it. (4) Circular waiting: closed chain of processes exist such that each process holds at least one resource needed by the next process in chain. CW implies H&W, but H&W is a policy decision.

Root causes: Resources (h/w, data, sync objects) are finite. Processes wait if needed resource unavailable. Resources may be held by other waiting processes.

Deadlock Prevention

Break one of the necessary conditions above:
(1) lock-free data structures using H/W support like CompareAndSwap(). Complex/not always feasible.

```
  atomicAdd(int *v, int a):
      do { int old = *v; } while ((CAS(v, old, old+a)==0);
```

(2) process can request all resource at once (all-or-nothing). Inefficient, starvation, unrealistic, etc.
(3) trylock(), grab lock if available: possible livelock. Not feasible or highly complex to safely achieve.
(4) Assign linear ordering to resource types and require process holding resource of one type R, to only request other resources that follow R in the ordering. Lock ordering – user must take special care of this.

Deadlock Avoidance

Allow (1) to (3) but ensure circular wait cannot occur. Requires knowledge of future resource requests to decide what order to choose. Depends on algorithm.

Strategy 1. Don't start process if max required + max needs of all process exceeds total system resources. This is pessimistic, and assumes processes need all resources at the same time.

Strategy 2. Don't grant individual resource request if future resource allocation "path" leads to deadlock.

Restrictions on Avoidance:
A. Processes must know max resource needs upfront*.*
B. Processes must be independent.
C. Must be a fixed number of resources to allocate.

Safe state: Exists a sequence of process executions that does not lead to deadlock, even if process requests their max allocations immediately

Unsafe state: no such sequence, but not yet deadlocked. If all processes requested their max resources at this point, there would be deadlock.

Banker's avoidance algorithm:
For every resource request:
1. Can request be granted? If not, block until we can.
2. Assume granted: Update state assuming granted.
3. Check new state is safe: If so continue, if not restore old state and block until safe.

Deadlock Detection and Recovery

Prevention and avoidance are costly. Instead, let them occur and check periodically (on every allocation request, fixed periods, or when system utilization drops below threshold) – when it happens, try to break it. Use a cycle finding algorithm on the **resource allocation graph** (directed) to find a circular wait.

Drastic: Kill all deadlocked processes.

Painful: backup & restart deadlocked processes; hope non-determinism stops deadlock reoccurring

Better: selectively kill processes until cycle is broken, re-run detection algorithm after each kill

Tricky: selectively pre-empt resources until cycle is broken. Processes must be rolled back.

Ostrich algorithm: All previous strategies costly in terms of CPU overhead or resource restriction. Most OS ignore problem and hope it doesn't happen often. Works because modern OS virtualizes most physical resources, eliminating finite resource problem.

Appendix

Bakery Algorithm

```
bool entering[1…nthreads]; int num[1..nthreads];
lock(i) { entering[i] = 1;
    num[i] =  1 + max(num[1], …, num[nthreads]);
    entering[i] = 0;
    for j = 1 to nthreads { // wait until j gets its num
        while (entering[j]); //spin
        // wait until all threads with smaller or equal
        // num (but higher priority) finish their work
        while ((num[j]!=0)&&(num[j],j)<(num[i],i)); //spin
    }
}
unlock(i) { num[i] = 0; }
```

Bounded producer/consumer (Semaphore):

```
Semaphore fillcnt=0, emptycnt=N, mutex = 1;
```

```
add(int v) {
  P(emptycnt);
  P(mutex);
  put(v);
  V(mutex);
  V(fillcnt);
```

```
int remove:
  P(fillcnt);
  P(mutex);
  int v = get();
  V(mutex);
  V(emptycnt);
```

Bounded producer/consumer (Monitor):

```
void add_to_buf(int v) {
  lock_aquire(mutex);
  while (elements==N) {
    cv_wait(notFull, lock);
  }
  data[inpos] = v;
  inpos = (inpos + 1) % N;
  num_elements++;
  cv_signal(notEmpty);
  lock_release(mutex);
}
```

```
int remove_from_buf() {
  lock_aquire(mutex);
  while (elements==0) {
   cvwait(notEmpty, lock);
  }
  int v = data[outpos];
  outpos=(outpos+1)%N;
  num_elements--;
  cv_signal(notFull);
  lock_release(mutex);
}
```

Reader/writer problem (Semaphore):

```
int rdcnt=0; Semaphore mutex = 1, w_or_r = 1;
```

```
Writer() {
  P(w_or_r); //lock others
  Write();
  V(w_or_r); //up for grabs
}
```

```
Reader() {
  P(mutex); //protect rdcnt
  If (++rdcnt==1) //1st read
    P(w_or_r); //sync writers
  V(mutex); //unlock rdcnt
  Read();
  P(mutex); //protect rdcnt
  If (--rdcnt==0) //last read
     V(w_or_r);
  V(mutex);
}
```

Hippos (Semaphore):

```
int marbles = M, scores[] = {0,0,…,0};
Semaphore mutex = 1, out = 0, done = 0;
```

```
void eat(int id) {
  P(mutex);
  scores[id]++;
  if (--marbles == 0) {
     V(out);
     P(done);
  }
  V(mutex);
}
```

```
int restart_game() {
  P(out);
  print_and_clr_scores();
  marbles = M;
  V(done);
}
```

Hippos (Monitor):

```
void eat(int id) {
  lock_acquire(mutex);
  while (marbles == 0)
     cv_wait(done, mutex);
  scores[id]++;
  if (--marbles == 0)
     cv_signal(done);
  lock_release(mutex);
}
```

```
int restart_game() {
  lock_aquire(mutex);
  while (marbles > 0)
     cv_wait(done, mutex);
  print_and_clr_scores();
  marbles = M;
  cv_broadcast(out);
  lock_release(mutex);
}
```

Multi-level translation with TLB & error checking:

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
if Success(TlbEntry = TLB_Lookup(VPN))              // TLB Hit
   if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Data = AccessMemory(PhysAddr)
   else: PROTECTION_FAULT
else:                                                // TLB Miss
   PDIndex = (VPN & PD_MASK) >> PD_SHIFT   // first get PDE
   PDEAddr = PDBR + (PDIndex * sizeof(PDE))
   PDE = AccessMemory(PDEAddr)
   if ~PDE.Valid: SEGMENTATION_FAULT
   else: PTIndex = (VPN & PT_MASK) >> PT_SHIFT   // get PTE
          PTEAddr = (PDE.PFN << SHIFT)+ PTIndex*sizeof(PTE)
          PTE = AccessMemory(PTEAddr)
          if !PTE.Valid: SEGMENTATION_FAULT  //or check swap
          elif !CanAccess(PTE.ProtectBits): PROTECTION_FAULT
          else: TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
                RetryInstruction()
```

Shared stack using CompareAndSwap:

```
void push(int i) {
  do { node *old = top, *new_top = new node(i, old);
  } while (CAS(top, old_top, new_top) != old_top);
}
```

Second-chance eviction algorithm (Clock):

```
while (coremap[clk].pte->frame & PG_REF) {
   coremap[clk].pte->frame &= (~PG_REF); //refbit=0
   clk = (clk + 1) % memsize;
}
int res = clk; clk = (clk + 1) % memsize; return res;
```