Assembler

TSL L	Read L and set condition code if L=0
BNZ	jumps if Z is not set.
MOV #n,L	sets L to constant n

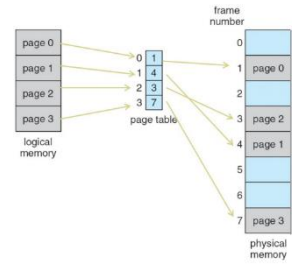
```
LOCK:   TSL L
        BNZ LOCK
UNLOCK: MOV #0, L
```

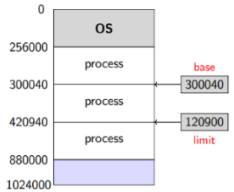
OS characteristics: Sharing, Concurrency, Non-determinism, Storing data. **OS structure**: Kernel – implements most executed functions of the OS. Executes in privileged mode. Has complete access to hardware. Always in memory. **Processes** - allow a single processor run multiple programs simult. Encapsulates code and state of a program. Turn single CPU into multiple virtual CPUs, each runs on a virtual CPU + provide concurrency, isolation (own address space), simplicity of programming, allow better utilisation of machine resources. Foreground processes (interact with users)/background pr. Process termination: normal completion, system call (exit() UNIX), abnormal exit, aborted, never (endless loop). 5 States: New, Ready, Running, Waiting/Blocked, Terminated. **Concurrent processes** – pseudo-concurrency – 1 CPU, process interleaving. Real concurrency – multiple CPUs (less than processes), still elements of pseudo concurrency. **Multiprogramming** – efficient use of resources. Processor used by other programs while waiting for I/O. Allows to perform diff activities at the same time. Requires **process/context switch** – CPU switches from executing process A to B, which cannot be pre-determined (events are non-deterministic). Caused by e.g. time slice of proc A expired, proc A blocked waiting for I/O or resource, proc has run to completion. No built-in assumptions about timing. When CPU switches between processes, info about proc A should be safely stored in a process descriptor or **process control block (PCB) –** data struct representing a proc in the kernel, stored in the **process table**. When a process is created (initialized or installed), the operating system creates a corresponding PCB. **A process virtual machine** allows a single process to run as an application on a host machine, providing a platform-independent programming envir by masking the info of the underlying hardware or OS. A process has its own virtual CPU, address space, open file descriptions, etc. **PCB stores:** process Ifs, state, priority (relat to other proc), program counter, context data (saved from registers), mem pointers, I/O status, file mgt, accounting info. 3 main categories: proc management, mem mgt, file mgt. **Process switch** – each IO class has an interrupt vector containing the address of the interrupt service procedure. 1) on interrupt, push registers, PC, PSW onto the stack by the interrupt hardware. 2) hardware jumps to address to service interrupt. 3) assembly lang routine saves registers to PB and calls the device-specific interrupt service routine. 4) C interrupt service. 5) Scheduler decides the next proc. 6) C procedure returns control to assembly code. 7) Assembly procedure starts up the new current proc. **Why avoid unnec context switches?** Direct cost: save & restore process state – PC, page table register, stack pointers, process mgt info (proc ID, parent proc, process group, CPU used, etc.), file mgt info (root directory, working dir, open file descriptors); indirect cost: disruption of mem caches, mem mgt registers, etc. **Threads** – smallest unit of execution. Threads within a proc share the same address space. Each has: execution state, saved thread context (if not running), execution stack, per-thread static storage for local variables, access to the memory and resources of its process – shared). Threads, as opposed to processes, are lightweight, fast creation/deletion, efficient comm between threads, activities can share data, reflect parallelism within application, where some activities may block. BUT one thread can overwrite another thr's stack, concurrency bugs, signals – which thread should handle a signal? For SIGALARM any thread. **Multi-threaded environments** – 1 process can have 1 or more threads. **User-level threads** (each proc manages its own threads) – process maintains a thread table and does the scheduling, kernel only manages processes -BUT blocking system calls stop all threads in the process – wasteful, difficult to implement pre-emptive scheduling**, kernel-lvl threads** – kernel can schedule multiple threads from the same process on multiple processors. Blocking system calls can be easily managed. BUT thread creation/deletion more expensive, requires kernel call – but can use thread recycling (thread pools). No application-lvl scheduler. **Hybrid –** bulk of scheduling and synchronisation of threads by the application Use kernel threads and multiplex user-lvl threads onto some kernel threads**. Process synchronisation**: **Critical section** – section of code where processes access a shared resource. Synchronisation mechanism required at entry/exit of CS – requirements: mutual exclusion (processes request permission to enter critical sections), progress, bounded waiting. **Solution to CS**: **disabling interrupts** – works on single-processor systems, buggy proc may never release CPU. Use CLI() (clear interrupt flag) and STI() (set interrupt flag), available to kernel code. **Busy waiting** – needed for strict alternation, continuously test the value of a variable. Wastes CPU time, should only be used when wait time is exp to be short. **Atomic operations** – sequence of statements that is/appears indivisible. **Lock variables** – L = 0 → lock open, L=1 → locked. **TSL** (test and set lock) instr – atomic instr provided by CPU, TSL(LOCK) atomically sets mem location LOCK to 1 and returns old value. **Spin locks** – locks using busy waiting – wasteful. **Lock**

page number	page offset
p	d
(m – n) bits	n bits

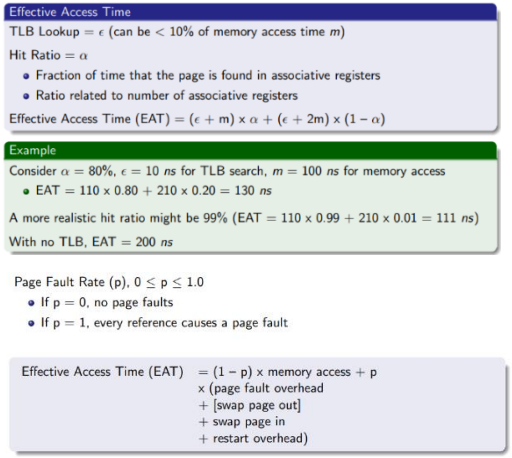
Per process items	Per thread items
Address space	Program counter (PC)
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Granularity – amount of data a lock is protecting. **Lock overhead** – memory space, initialisation, time required to acquire and release locks. **Lock contention** – measure of the num of proc waiting for a lock. **Finer lock granularity but understand trade-offs – release a lock ASAP, make CS small**. **Race condition** - when multiple threads/proc read and write shared data and the final result depends on the relative timing of their execution (interleaving). **Semaphores**– processes cooperate thru signals, incl. block or continue; sem. use special variables, accessible via atomic operations: down(s), up(s), init(s, i). 2 components: counter int (initial value specifies num of proc that can access shared data at the same time; queue of processes (FIFO). **Binary semaphores** (mutex) – counter initialised to 1. **Reader-writer problem** – when a writer needs access, mutex must be ensured. File can hold arbitrary num of items. Writer – can only write if no other proc is reading or writing. Reader – can only read if no other proc is writing, can read if others are reading. For writer – need mutex. For reader – keep count of the num of readers, let writer know is num of readers=0. **Deadlocks** – if each process is waiting for an event that only another process can cause. Necessary conditions: 1) mutual exclusion, 2) hold and wait (proc holding resources may request new resources), 3) no pre-emption (no res can be forcibly removed from the holding proc), 4) circular wait (proc >=2 in circular chain, waiting for the res held by the next proc). **Dining philosophers problem** – 5 philosophers and 5 chopsticks. **Managing deadlocks** – **Ignore it** – contention for resources low – deadlocks rare. **Detection and recovery** – after system is deadlocked, dynamically build resource ownership graph and look for cycles – when arc inspected, it is marked and not visited again. Detection: Resource allocation graphs – directed graph, directed arc from resource to proc means the proc is currently owning the resource, directed arc from proc to resource means the proc is currently blocked waiting for the resource; cycle in the graph = deadlock. Recovery: pre-emption – temp give the resource to another proc, rollback on deadlock to prev state, killing processes – select random proc in cycle and kill it. **Dynamic avoidance** – grant res only when it is safe; Banker's algorithm – check if there is enough resources to satisfy any max. request from some customer. A state is safe IFF there exists a sequence of allocations that guarantees that all customers will be satisfied. **Prevention** – ensure min. 1 out of the 4 necessary conditions can never hold. Dining philosophers' problem – request resources in specific order: proc request lower-numbered res first and only request higher-numbered res if lower res is already acquired. **Communication deadlock** – proc A sends message to B and blocks waiting for B's reply. B didn't get A's msg, so A is blocked, and B is blocked waiting for message. Solve by communication protocol based on timeouts. **Livelock** – proc/threads not blocked but they/the system as a whole does not make progress. **Starvation** – process waits forever to enter the critical section. Proc/thread is unable to acquire the neces res to execute its task, despite waiting for an extended time **Scheduler** – allocates processes to processors, select a ready proc from head of ready queue and move it to the running state. Invoked at every entry to the kernel. Batch systems – throughput, turnaround time, max CPU utilisation. Interactive systems – response time, meet user expectations. Real-Time systems – meeting deadlines (soft vs hard), predictability. **Non-pre-emptive** – proc runs until it blocks or voluntarily releases the CPU. **Pre-emptive** – let proc run for a max time – requires clock interrupt. **CPU-bound processes** – most time spent using the CPU. **I/O-bound proc** – most time spent waiting for I/O, tend to use CPU briefly before issuing another I/O req. **First-Come First-Served Scheduling** – runnable proc added to the end of the ready queue, non-pre-emptive – easy to implement and no indefinite postponement. BUT perhaps not the most efficient resource use. Pre-emption can help with the waste if CPU-bound proc are in a queue before I/O proc and take longer. **Round-Robin Scheduling** – pre-emptive, proc runs until it blocks, or time quantum exceeded. Fair - ready jobs get equal share of CPU, response time good for small num of jobs, avg turnaround time low when runtimes differ, poor for similar runtimes. **Quantum –** allowance of CPU time. RR overhead depends on the context switch time and RR quantum. Large quantum = less overhead, worse response time. Small quantum – larger overhead, better response time. Optimal=quantum average CPU time between I/O to minimise context switches, and much larger than context switch cost but provide decent response time. **Shortest Job First (SJF)** – non-pre-emptive, with run-time known in advance – pick the process with shortest CPU burst first. Optimal when all jobs are available simultaneously. **Shortest Remaining Time (SRT)** – pre-emptive version of SJF – runtimes known in advance. If new process arrives with execution time less than the remaining time for the running process, run it. Run-time is usually not available in advance – can compute CPU burst estimates, or use user-supplied estimates. Need to counteract cheating. **Fair-Share Scheduling** – users are assigned a fraction of CPU, scheduler takes into account the owner of process before scheduling it. **Priority Scheduling** – always run the job with the top priority, which can be externally defined or based on process-specific metrics. Priorities can be static (no change during execution) or dynamic. **General-Purpose Scheduling** – favour short and I/O-bound jobs – good resource utilisation, short response times. Quickly determine the nature of the job and adapt to changes (whether currently I/O- or CPU-bound). **Multilevel Feedback Queues** – (Mac OS X, Windows Vista/7) – one queue for each prio level, and each queue can use different scheduling algorithm (usually RR). Run job on highest non-empty priority queue. Need to determine current nature of the job, and worry about starvation of lower-priority jobs. Feedback – mechanism – job priorities recomputed periodically. Aging – increase job's prio as it waits. Inflexible – applications have almost no control, priorities make no guarantees. Does not react quickly to changes – often needs a warm-up period to get better results. Cheating to boost a priority is a concern. Cannot donate priority. **Lottery scheduling** – jobs receive lottery tickets for resources, e.g., CPU time, and each time one ticket is chosen at random. Num of lottery tickets translate to the p% of resource. Highly responsive because of % of lottery tickets received. No starvation, jobs can exchange tickets – allows prio donation. **User-level thread scheduling** – 1) kernel picks a process, 2) runtime system picks a thread, so only those process's threads will be executed. **Kernel-level thread scheduling** – 1) kernel picks a thread, so any schedule is possible. **Memory management**: binds logical address space (generated by CPU, seen by the process) to physical address space (address seen by the mem unit, refers to physical mem). They are same in compile- and load-time address-binding schemes, but different in execution-time address-binding schemes. **Memory-Management Unit (MMU)** – hardware device for mapping logical to physical addresses, needs to be fast. E.g., adds value in relocation register to every address generated by process when sent to memory. **Memory Allocation** – main mem split into 2 partitions: resident operating system (held in low memory with interrupt vector) and user processes (held in high memory). **Contiguous Memory Allocation** – with relocation registers, base register contains physical start address for process, limit reg contains max logical address for process. MMU maps logical address dynamically. Physical address=logical address+base. **Multiple-Partition Allocation** – holes (blocks of available mem) of diff sizes scattered throughout memory. When new user proc arrives, allocate mem from a hole large enough. OS keeps track of free and allocated holes. **Dynamic Memory Allocation** – first-fit (first hole that is big enough), best-fit (smallest hole big enough), worst-fit (allocate largest hole – produces largest leftover hole; worst speed and storage utilisation). External **Fragmentation** – memory exists to satisfy requests, but not contiguous. Reduce by **compaction** – shuffle mem contents to place all free mem together in 1 large block – leads to I/O bottlenecks. **Swapping** – swap processes temporarily out of memory to backing store, bring back later for contd. execution. Requires swap space and transfer time. **Virtual memory** – separation of user logical mem from physical mem – only part of a proc needs to be in memory for execution, logical address space can be larger than physical address space. Address space can be shared! More efficient process creation. **Paging** – Frames (fixed-sized blocks of physical mem), pages (same-sized blocks of logical mem). To run a program of size *n* pages – find n free frames & load the program. Set up page table to translate from logical to physical. Physical address space can be non-contiguous; process allocated physical mem when available, avoid external fragmentation and problems of variable-sized memory chunks. Translating logical address into physical – addr consists of: page num (used as index in the page table, which has base address of pages in physical mem) and page offset (defines physical mem address sent to mem unit, combined with base address), since address offset will be same in frame and page. Page num needs to be translated into corresponding frame address, take number of bits required for address (e.g. for a page of 64 bytes, need 64 addresses=6 bits ($2^6 = 64$). 10-bit virtual address=6 bits for page offset, 4 bits for page number (0-15), total 16 pages. **Internal Fragmentation** – allocated mem larger than requested mem, but size difference is internal to partition. Avg-case fragmentation: ½ frame size. Each page table entry takes memory to track, page size growing over time. **Page table** – kept in main memory,





Page-table base register (updated when context switching) points to page table, page-table length register indicates size → inefficient; solution: **Associative memory** – supports parallel search, uses special fast lookup hardware cache; is called: **Translation Look-aside Buffer (TLB)** – address translation – (p, d) – if p is in the associative register, get frame # out, else get # from page table in memory. TLB needs to be flushed after context switch – overhead! Page table size can grow to v. large sizes. **Hierarchical Page Table** – page-table broken up and paged if too large. **Two-Level Paging** – each part of the page table that is being paged must fit on a page. Num of entries on one page = $\frac{page\ size}{address\ size} = \frac{4\ KB}{32\ bits} = 2^{10}$, therefore, need 10 bits. Address bits left for top-lvl page table=32-10-12 (page offset) =10. **Hashed Page Table** – hash virtual page num into page table, which contains chain of elements hashing to some location, search for match of virtual page number in chain, and extract corresponding physical frame if match found. **Inverted Page Table** – one entry per physical frame; decreases mem needed to store page table, but increases time to search table when page reference occurs. **Segmentation** – separate address spaces for code, data, stack. **Segment** – corresponds to program, procedure, stack, object, etc., it's an independent address space from 0-max, which can grow/shrink and supports protection (read/write/execute). Mem allocation harder bc of variable size, but good for shared libraries. **Segmentation Address Translation** – one bit in table indicates whether segment is in memory, another bit indicates whether segment is modified. **Demanding Paging** – bring page into memory only when needed – lower I/O load, less mem needed, faster, support for more users. Page needed → reference it (invalid reference, not-in-memory – bring into memory (page fault)). Memory control bits – associated with a frame. Valid (1)/invalid (0 – page missing in physical mem – default initial) bit. Page replacement bits – indicate if page has been modified or referenced. **Page fault** – first page reference → trap to OS. If valid reference but not in memory → handle request (get empty frame, swap page from backing store into frame, reset tables, validation bit=1, restart last instruction). If no page free, **replace unused page** using a strategy – minimise bringing same page into memory several times, prevent over-allocation of memory, use modify bit to reduce overhead of page transfers. **Page replacement algorithms** – aim is to lower page-fault rate. **Optimal algorithm** – replace page that will not be used for the longest period of time – unimplementable, useful as a reference for other algos. **First-In First-Out Algorithm** – Replace oldest page, could replace heavily used one. Belady's Anomaly – more frames → more page faults. **Least Recently Used Algorithm** – each page entry has a counter – when page referenced, copy clock into counter. Expensive! – use approximations instead, e.g. use reference bit, or clock replacement policy. **Least Frequently Used (LFU)** – replace page with smallest count, dodge risk by resetting counter or using aging. **Most Frequently Used (MFU)** – replace page with top count, bc page with smallest count probably just brought in and will be used. **Locality of reference** – programs tend to request the same pages in space and time. **Working Set Model** – working set of pages W(t, w) – referenced by process during process time-interval (t-w) to t. **Working Set Clock Algorithm** – keeps track if page in working set. If r=0, calculate age. Working set size – should aim to reduce misallocation and reduce page fault numbers. **File system** support functions: logical name to physical disk address translation, management of disk space, file locking for exclusive access, performance optimisation, protection against system failure, security. **Space allocation** – file size is variable, space allocated in blocks (512-8192 bytes) – block size too large wastes space for small files (more mem needed for buffer space), block size too small wastes space for large files (high overhead for data management, high file transfer time). **Contiguous File Allocation** – place file data at contiguous addresses on storage device. Disadv: external fragmentation, poor performance if file grows/shrinks over time. **Block Linkage (Chaining)** – place file data by linking them together. Disadv: need to search from start of list to find data block, search process can be slow, and many seeks can occur, wastes pointer space in each block. **Block Allocation Table** – store pointers to file blocks. Directory entries indicate first block of file, block number as index in the block allocation table, determines location of next block. File allocation table – stored on disk but cached in memory, akin to BAT. Reduced num of seeks to access a record, but files become fragmented, and table can get large. **Index Blocks** – each file has one or more index blocks, which contain a list of pointers that point to file data blocks. File's directory entry points to its index block. Chaining – may reserve the last few entries in index block to store pointers to more index blocks. Adv: searching in index blocks themselves, if blocks near corresponding data blocks, then quick access to data, can cache index blocks for faster access. **UNIX- Inodes** – form of index blocks, structured as inode on disk; on file open, OS opens inode table. Includes: disk device num, inode num, num of processes with open file, major/minor device num. Max file size calculation using inodes: $Num\ of\ direct\ pointers * blocksize$ – direct pointers. $NumOfEntries * BlockSize = \left(\frac{BlockSize}{Size\ of\ pointer}\right) * BlockSize$ – single indirect pointers. $NumOfEntries^2 * BlockSize = \left(\frac{BlockSize}{Size\ of\ pointer}\right)^2 * BlockSize$ – double indirect pointer. $MaxFileSize = sum$. **Free Space Management** – by using a free list – a linked list of blocks containing locations of free xblocks. **Bitmap** – contains one bit in memory for each disk block - indicates whether block is in use. Can quickly determine available contiguous blocks at certain locations on secondary storage, but may need to search entire bitmap to find free block, resulting in execution overhead. **File system layout** – Fixed disk layout (with inodes), or Superblock. **File system directories** – directory maps symbolic file names to logical disk locations, ensures uniqueness of names. **Single-lvl file system** – simplest, stores all files with unique names in one directory. Performs linear search to locate file – poor performance. **Multi-lvl (Tree) directory structure** – hierarchical FS. Root indicates where the root directory begins, root directory points to various directories, each of which contains entries for its files. Name uniqueness within dir. **Link** – reference to dir/file in another part of FS, allow alternative names. Hard link – reference address of file (Unix), Soft/Symbolic link – reference full pathname of file/dir, created as directory entry. File deletion – either leave links and cause exceptions when used, or keep link count with file and delete file when count=0. **Mount operation** – combine multiple FSs into one namespace, allows reference from single root directory, support for soft-links to files. **Mount point** – directory in native FS assigned to root of mounted FS. FS manages mounted directories using mount tables. **Device management** – **Character devices** – delivers/accepts stream of characters, without regard to block structure; not addressable. **Block devices** – stores information in fixed-sized blocks, transfers are in units of entire blocks. Hardware controllers have registers used for communication with CPU – OS can write to these registers to command the device, and can read from these registers to learn about state of device, ready/notready. **I/O Software** – goal of device independence from device type or instance, uniform naming, device variations. **I/O Techniques** – **Programmed I/O –**




```
copy_from_user (buffer, p, count);      // p = kernel buffer
for (i = 0; i < count; i++) {           // loop on every char
  while (*printer_status_reg !=READY);  // loop until ready
  *printer_data_register = p[i];        // output one char
}
return_to_user ();
```

Interrupt Driven I/O –

```
copy_from_user (buffer, p, count);
enable_interrupts ();
while (*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler ();
```

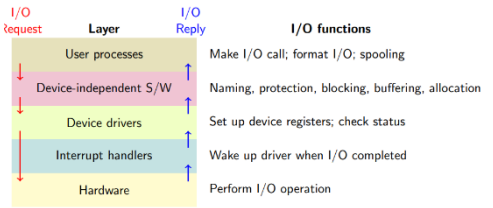
```
if (count == 0) {
  unblock_user ();
} else {
  *printer_data_register = p[i];
  count = count - 1;
  i++;
}
acknowledge_interrupt ();
return_from_interrupt ();
```

Direct Memory Access (DMA)

```
copy_from_disk (buffer, p, count);
set_up_DMA_controller ();
scheduler ();
```

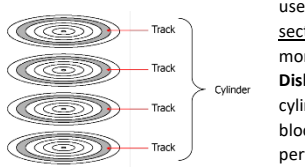
```
acknowledge_interrupt ();
unblock_user ();
return_from_interrupt ();
```

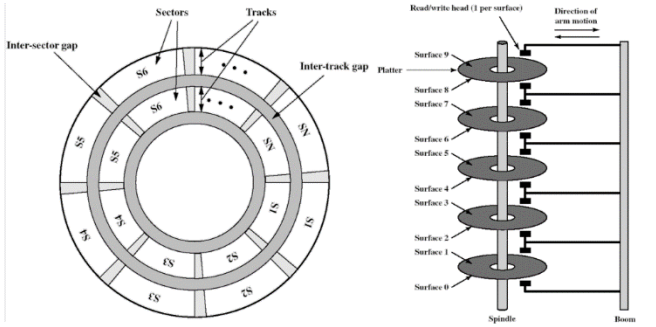
User-level software – most of the I/O software within the OS, some consist of libraries linked together with user programs, or **spooling** – a way of dealing with dedicated I/O devices in a multi programming system, where some devices can be used only by one proc at any given moment and cant be shared. **Spooled devices** – printer output saved to disk file, file printed later by spooler daemon – printer only allocated to spooler daemon, no normal proc allowed direct access. **Uniform interface for device drivers** – new drivers can be installed easily, the writer of the driver knows what to expect from the OS. For each class of devices, OS defines a set of



functions that the driver must supply. Each device has a major device num and minor device num. **Buffering – unbuffered input** – each interrupt can wake up user's process; **buffering in user space** – OS writes into user space's buffer and wakes up user's process once the buffer is full. **Buffering in the kernel followed by copying to user space** – OS writes into a buffer in kernel's mem space, then copies data to user's space. **Error reporting** – 2 classes: programming errors (solution – report back an error code to the caller), actual I/O error (solution up to the driver – try to solve or report back error code). **Device drivers** – device-specific code for controlling an I/O device, handles usually one device type. Part of kernel, positioned below the rest of the OS. Most Oss define a standard interface for block and character devices. **Interrupt handlers** –

Disk Addressing – physical hardware address (cylinder, surface, sector), although modern disks use logical sector addressing (sectors numbered 0..n). Disk sector layout – surface divided into zones. Outer zones have more sectors per track, zones hidden using virtual geometry. **Disk formatting** – low level format (disk sector layout, cylinder skew, interleaving), high lvl format (boot block, free block list, root directory, empty file system). Disk performance – given b- num of bytes to be transferred, N – num of bytes per track, r – rotation speed in revolutions per second: Seek time t_{seek}, Latency time (rotational delay) – $t_{latency} = \frac{1}{2xr}$, Transfer time – $t_{transfer} = \frac{b}{Nxr}$, Total access time – $t_{access} = t_{seek} + t_{latency} + t_{transfer}$. **Disk scheduling** can minimise seek and/or latency time, order pending disk requests with respect to head position. **First Come First Served** – no ordering or requests – random seek patterns, OK for lightly-loaded disks, fair scheduling. **Shortest Seek Time First** – order requests according to shortest seek distance from current head position – discriminates against inner- and outermost tracks, unpredictable and unfair. **LOOK** – only change direction when reaching outer-/innermost request in preferred direction – common, but long delays for requests at extreme locations. **C-LOOK** – services requests in one direction only – when head reaches innermost request, jump to outermost req. Lower variance of requests on extreme tracks, may delay requests indefinitely. **N-Step SCAN**- services only requests waiting when sweep began, similar to SCAN. Requests arriving during sweep serviced during return sweep. Doesn't delay requests indefinitely. **Solid State Disks (SSDs)** – no moving parts, low power, lightweight, writing takes more time but fast reading. $Latency = QueueingTime + ControllerTime + TransferTime$ **SSDs writing** – only empty pages can be written to, if no empty page, need to erase one first. Controller maintains a pool of empty blocks. Blocks have a finite lifetime – used often wear out quickly. **Flash Translation Layer** – layer between OS and NAND mem, maps block numbers from OS to physical page nums, controller can control where pages are stored. Copy on Write – when OS updates a page, write a new page rather than overwrite one, update mapping. Use Wear Levelling – distribute writes as evenly as possible across the NAND blocks.





1 KB = 2^{10} bytes = 1024 bytes vs 1 KB = 10^3 bytes = 1000 bytes
1 MB = 2^{20} bytes = 1024^2 bytes vs 1 MB = 10^6 bytes = 1000^2 bytes
1 GB = 2^{30} bytes = 1024^3 bytes vs 1 GB = 10^9 bytes = 1000^3 bytes