Introduction

Motivation – An OS is a piece of software that virtualises the computer; it provides a consistent and uniform view of underlying hardware and abstracts this complexity from the user, only the OS can access hardware directly.

Characteristics – **(1) Sharing:** share data, programs and hardware via time and space multiplexing. Resource allocation by ensuring fair and efficient use of resources, support simultaneous access to resources, provides mutual exclusion and protection against corruption. **(2) Concurrency:** support several simultaneous parallel activities, ability to switch activities at arbitrary times fairly, and ensure safe concurrency. **(3) Non-Determinism:** Never makes assumptions on events, as events arrive in unpredictable orders. **(4) Storing Data:** easy access to files through user defined names, enforce access controls, protection against failures, and storage management for easy expansion.

Structure – **Kernel:** core of the OS, implementing the most commonly executed functions of the OS, running in privileged mode (direct access to hardware), it is always in memory.

Monolithic Kernels – kernel is a single executable with its own address space, pushes parameters onto the stack and traps to execute system calls. **Advantages:** efficient calls within kernel and easy to write kernel components due to shared memory. **Disadvantages:** complex design and no protection between kernel components.

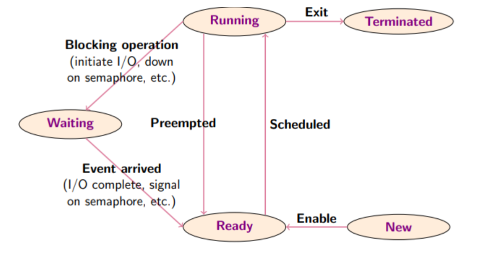
Microkernels – minimal kernel with majority of functionality in user space, engages in IPC with separate servers for I/O devices. **Advantages:** kernel not complex so less error prone, servers have clean interfaces and can crash and restart without bringing down the kernel. **Disadvantages:** high overhead of IPC communication.

Hybrid Kernels – Combines features of both monolithic and microkernels, it has a more structured design but offers a performance penalty for users.

Processes

Introduction – a process is the abstraction of running a program, it encapsulates the code and state of a program. It allows a single processor to run many processes simultaneously by virtualising the CPU. Processes can provide the illusion of concurrency, provide isolation of address space, a simplistic way of programming and allow better utilisation of machine resources.

Process Model – **(1) New:** process is being created. **(2) Ready:** process is runnable and waiting to be scheduled for the processor. **(3) Running:** process is currently executing on a processor. **(4) Waiting/Blocked:** process is waiting for an event to occur. **(5) Terminated:** process is being deleted.



Concurrency – Pseudo-concurrency is when a single hardware processor switches between processes over time by interleaving process execution, gives the illusion of concurrent execution. Real-concurrency is possible with multiple hardware processors, where each process is run on a separate CPU, still uses pseudo-concurrency as there are usually more processes than processors.

Multiprogramming – I/O is slow compared to computation time, so waiting for I/O is a poor utilisation of resources. Instead, multiprogramming allows the processor to be used by other programs while a program is performing I/O, allows the user to perform different activities at the same time.

Context Switches – switching between two processes due to an event (non-deterministic). When a context switch occurs, all process data is stored in a process control block, a data structure representing a process. PCB's are stored in a process table.

1. On interrupt, the PC, PSW, some registers are pushed onto the (current) stack by the **interrupt hardware**
2. Hardware jumps to address (PC from interrupt vector) to service interrupt
3. Assembly language routine saves registers to PCB and then calls the device specific interrupt service routine
4. C interrupt service runs (typically reads, writes & buffers data)
5. **Scheduler** decides which process to run next
6. C procedure returns control to assembly code
7. Assembly procedure starts up the new current process

Cost – Context switches are expensive due to the direct cost of saving/restoring process state and the indirect cost of cache disruption and memory management registers. The information that must be stored are: **(1) Program counter**, page table register and other internal registers, **(2) process management** information such as process ID, priority and CPU uses, **(3) File management information** such as root directory, open directory, and open file descriptors.

Threads

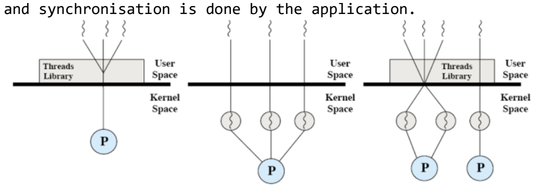
Introduction – Threads are the smallest unit of execution that can be allocated processor time, they share the same address space as the parent process. Each thread has an execution state, saved thread context, execution stack, static storage and access to memory of the parent process.

Thread vs Processes – Processes are heavyweight: expensive to create and destroy, diffcult to communicate between processes, blocking activites block entire application, and context switches are expensive. Threads are lightweight: faster creation and deletion, efficient communication between threads, activites can share data, and reflect parallelism within the application (blocking a thread does not block parent application). However threads can cause memory corruption and concurrency bugs.

User-Level Threads – Threads implemented by a software library in user space, kernel is oblivious to threads and thinks its managing the process only. The process itself handles thread scheduling and does not require kernel mode privileges. **Advantages:** better performance on creation and termination, switching, and synchronisation since it doesn't involve the kernel as it allows for application specific runtimes and scheduling algorithms. **Disadvantages:** blocking system call blocks all threads in the process (entire process blocks) since kernel is unaware of the existence of threads, defeats the whole purpose of threading. Page faults also block the entire process, and difficult to implement pre-emptive scheduling.

Kernel-Level Threads – Thread management is entirely done by the kernel. **Advantages:** Kernel can schedule multiple threads from the same process on different processors, page faults and system calls can be easily accommodated for and don't block the entire process. **Disadvantages:** thread creation/termination is more expensive as they involve kernel calls, but this can be mitigated using thread pools. Thread synchronisation is also expensive due to blocking system calls, same for scheduling.

Hybrid Approaches – Thread creation is done in user space which is multiplexed onto kernel threads, bulk of scheduling and synchronisation is done by the application.



User-level threads Kernel-level threads Hybrid

Synchronisation

Critical Section – Section of code in which processes access a shared resource, this involves reads and writes at memory locations that are shared between processes/threads.

Race Condition – Occurs when multiple processes read and write data and the final result depends on the relative timing of their execution.

Solution – **(1) Mutual Exclusion:** no two processes access the critical section simultaneously, must first acquire permission. **(2) Progress:** no process running outside the critical section stops processes from entering the critical section. **(3) Bounded waiting:** no process waits forever to access the critical section.

Disabling Interrupts – Using **CLI()** and **STI()** instructions to clear and set interrupt flags when a process enters the critical section. This only works on uniprocessor systems and does not guarantee bounded waiting.

Software Solution – Strict alternation between processes, does not guarantee progress. This is also called busy waiting, it wastes CPU time and should only be used when wait times are known to be short.

P_0 turn $\boxed{0}$ P_1

```
while (true) {
  while (turn != 0)
    /* loop */ ;
  critical_section()
  turn = 1;
  noncritical_section0();
}
```

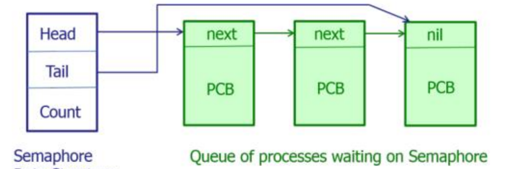
```
while (true) {
  while (turn != 1)
    /* loop */ ;
  critical_section()
  turn = 0;
  noncritical_section1();
}
```

Atomic Operations – Sequence of statements that appear to be indivisible. Examples of these are locks, which are set to 0 or 1 and prevent processes in accessing a critical section when it is busy. Lock granularity refers to the amount of data the lock is protecting. Locks that use busy waiting are called spin locks.

Lock Overhead – Measure of cost associated with using locks in terms of memory space, initialisation and time required to acquire and release locks. Lock contention is the measure of processes waiting for a lock (more contention means less parallelism). Coarser granularity means lower overhead and complexity but higher contention. Finer granularity means higher complexity and overhead but lower contention.

Semaphores – Processes cooperate by a means of signals: processes block waiting on a signal and continue if it has received a specific signal.

Semaphores have two private components
- A counter (non-negative) integer
- A queue of processes currently waiting for that semaphore – queue is typically first in first out (FIFO)



Semaphore Data Structure Queue of processes waiting on Semaphore

Deadlocks

Conditions – A deadlock is when a set of processes are waiting for an event that only another process can cause. **(1) Mutual exclusion:** resource cannot be used by more than one process at a time. **(2) Hold and Wait:** process already holding resources may request a new resource. **(3) No pre-emption:** no resource can be forcibly removed from a process. **(4) Circular wait:** two or more processes in a circular chain, each waiting for a resource held by the next.

Strategies – Ignore it, detection and recovery, dynamic avoidance and prevention.

Detection and Recovery – Detect deadlocks and recover after occurrence. Build a directed resource ownership graph and perform and DFS to detect cycles. If deadlock detected then either temporarily remove resource from a process to break cycle, rollback to state previous to deadlock, or kill a process in the cycle.

Dynamic Avoidance – Bankers algorithm.

SAFE

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

UNSAFE

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Safe state

Are there enough resources to satisfy **any** (maximum) request from some customer? assume customer repays loan, and then check next customer closest to the limit, etc.

A state is safe iff there exists a sequence of allocations that guarantees that all customers can be satisfied

Prevention – **(1) Attack mutual exclusion:** share the resource. **(2) Attack hold and wait:** require all processes to request resources before start (violated non-determinism). **(3) Attack no pre-emption:** force process to give up resource half way through, not good in case of printer. **(4) Attack circular wait:** force single resource per process, if a process wants another resource it must release it first (optimality issues), difficult to organise large numbers of resources.

Livelock – Processes/threads are not blocked, but the system as a whole does not make progress.

Scheduling

Goals – Ensure fairness, avoid starvation, enforce policy, maximise resource utilisation, and minimise switching overhead.

Batch Systems – maximise throughput and CPU utilisation whilst minimising turnaround time.

Interactive Systems – Prioritise fast response times, and meets user expectations (predictability).

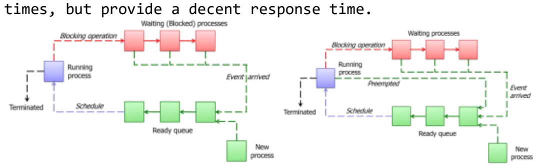
Real-time Systems – Meet hard and soft deadlines while being predictable.

Scheduling Types – Non-pre-emptive scheduling lets the process run until it blocks or voluntarily releases the CPU. Pre-emptive scheduling lets processes run for a fixed amount of time, utilise priority and clock interrupts.

CPU-Bound vs I/O-Bound Processes – CPU-bound processes spend most of their time using the CPU, whilst I/O bound processes spend most of their time using I/O and tend to use the CPU briefly in a short burst before issuing another I/O request.

FCFS Scheduling – Non-pre-emptive. **Advantages:** no indefinite postponement (all processes eventually scheduled), and easy to implement. **Disadvantages:** long jobs followed by short jobs drastically increase turnaround time (not fair), favours CPU bound processes over I/O and lower CPU utilisation.

Round-robin scheduling – Pre-emptive, process runs to completion or time quantum. **Advantages:** fair as all processes get same CPU time, good response times for small numbers of jobs. **Disadvantages:** low average turnaround time when run-times differ, and poor for similar run-times. Time quanta should be selected to be longer than context switch times, but provide a decent response time.



Shortest Job First – Non-pre-emptive with runtimes known in advance, optimal when all jobs available simultaneously.

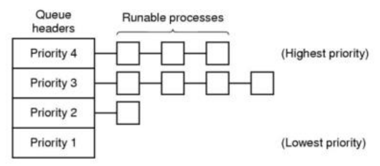
Shortest Remaining Time – Pre-emptive version of SJF but run-times need to be known in advance, short jobs get good service, but may cause starvation of longer processes if short processes keep getting scheduled.

Priority Scheduling – Jobs run based on their priority, which can be user defined or based on process-specific metrics. Priorities can be static or dynamic.

General Purpose Scheduling – favours short I/O bound jobs, good resource utilisation and short response times. Quickly adapt to changes in process nature.

Multilevel Feedback Queues – implements aging, which increases a jobs priority as it waits to prevent starvation. They are not very flexible and do not react quickly to changes, cheating is also a concern and cannot donate priority.

One queue for each priority level
- Run job on highest non-empty priority queue
- Each queue can use different scheduling algorithm
  - Usually RR – quantum may be varied, e.g. highest priority is I/O-bound with short quantum. Exceed quantum → move down level but get bigger quantum



Lottery Scheduling – Jobs given lottery tickets to access CPU, at each scheduling decision, a ticket is chosen at random and that job gets CPU. Number of lottery tickets are meaningful and fast response times, no starvation, processes can exchange tickets. However it has unpredictable response times.

Memory Management

Logical address space – generated by the CPU, address space seen by the process.
Physical address space – address space seen by the memory unit, refers to system physical memory.
Memory Management Unit – maps logical to physical addresses, value added to relocation register to generate physical address. Kernel memory is stored in low memory locations, whilst user memory is stored in high memory locations.
Contiguous memory allocation – base register contains physical start address of process, limit register contains maximum logical address of process. Physical address = logical + base.
Multiple-Partition allocation – Holes are blocks of available memory, process gets allocated memory to large enough hole. First-fit: allocate first hole that's big enough. Best-fit: allocate smallest hole that's big enough. Worst-fit: allocate largest hole.
Fragmentation – External fragmentation is when enough memory is available to satisfy the request, but isn't contiguous. Can be solved by compaction but has high I/O overhead. Internal fragmentation is when the allocated memory is larger than the memory needed, but size difference is internal.
Swapping – Number of processes are usually limited by the amount of memory. A solution would be to swap processes in and out of memory temporarily, bringing them back later for execution. It requires swap space on disk, and transfer time is a major issue. Entire Address space is swapped.
Virtual Memory – Processes can run even when they are not completely loaded onto RAM, giving the illusion that a process has more memory than available. Address space can be shared by processes.
Paging – frames are fixed size blocks of physical memory, pages are blocks of same size as frames of logical memory. Physical address space is non-contiguous, so processes are given physical memory when available which avoids external fragmentation. Some pages in address space on disk and mem.
Address Translation – Addresses generated by the CPU have a page number (index to page table) and page offset which defines the physical memory address sent to the memory unit (combined with base address). Page table entries have physical addresses as well as some control bits.
Page table implementation – page table in memory, PTBR points to and PTLR indicates size.
Associative memory – Hardware cache that allows pages to be accessed by content, a TLB entry has the page number and frame number, supports parallel search. TLB's are flushed during context switches but ASID can protect entries.

Effective Access Time

TLB Lookup = ϵ (can be < 10% of memory access time m)

Hit Ratio = α

- Fraction of time that the page is found in associative registers
- Ratio related to number of associative registers

Effective Access Time (EAT) = $(\epsilon + m) \times \alpha + (\epsilon + 2m) \times (1 - \alpha)$

Example

Consider $\alpha = 80\%$, $\epsilon = 10$ *ns* for TLB search, $m = 100$ *ns* for memory access

- EAT = $110 \times 0.80 + 210 \times 0.20 = 130$ *ns*

A more realistic hit ratio might be 99% (EAT = $110 \times 0.99 + 210 \times 0.01 = 111$ *ns*)

With no TLB, EAT = 200 *ns*

Hierarchical Page Table – page table is broken up into levels, it allows for efficient memory usage and supports larger address spaces but has indexing overhead and slower memory accesses.
Hashed Page Table – hashes virtual page number, page table contains chain of elements hashing to same location, search for match of virtual page in chain and extract corresponding physical frame if found. May be faster access times than hierarchical page tables but complex to implement.
Inverted Page Table – One entry per physical frame, decreases memory needed to store the page table, but increases search times.

For a machine with 32-bit addresses and a 4 KB page size (page offset needs 12 bits)

How do you break the page table up?

Each part of the page table that is being paged must fit on a page

- Number of entries on one page = $\frac{\text{Page size}}{\text{Address size}} = \frac{4\ KB}{32\ bits} = 2^{10}$
- No of bits required for 2^{10} entries = 10
- Address bits left for top-level page table = 32 − 10 − 12 = 10

Thus, logical addresses is as follows

page number		page offset
p_1	p_2	d
10	10	12

p_1 → index into the outer page table
p_2 → displacement within page pointed to by outer page table

Segmentation – Paging gives a one dimensional virtual address space. Segmentation gives an independent address space of some maximum size, which can grow and shrink independently. Unlike pages, programmers are made aware of segments, where the segment encapsulates everything in the program. Memory allocation is harder due to dynamic size, and may suffer from external fragmentation, however good for shared libraries. One bit signals if segment is in memory, and another signals if it's modified.
Demand Paging – pages are only brought into memory when needed, resulting in lower I/O overhead, less memory needed, faster response times and support for more users. If a page is needed and not in memory, it causes a page fault, which is a kernel trap that brings the page into memory from the backing store. Recall a page has protection bits, and a valid/invalid bit which denotes if the page is in memory or not.

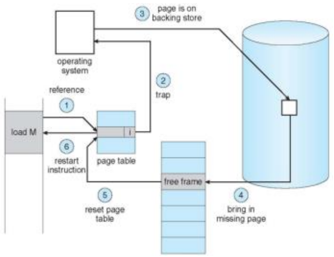
First reference (page fault) → trap to OS

OS looks at validity of request

- Invalid reference → abort
- Valid reference but not in memory → handle request

To handle valid request

- Get empty frame
- Swap page from backing store into frame
- Reset tables, validation bit = 1
- Restart last instruction

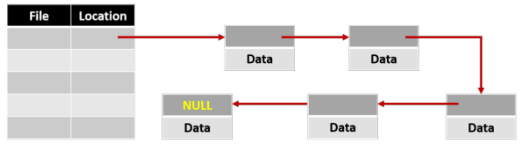


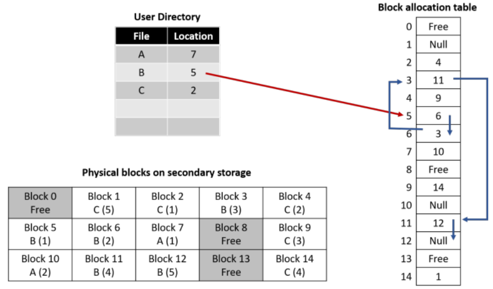
Effective Access Time (EAT) = (1 − p) x memory access + p x (page fault overhead + [swap page out] + swap page in + restart overhead)

Virtual Memory Tricks – **(1) Copy on write:** Allows parent and child processes to initially share same pages in memory, if either process modifies shared page, then copy page. Efficient process creation - copy only modified pages, free pages allocated from pool of zeroed-out pages. **(2) Memory-mapped files:** Map file into virtual address space using paging, simplifies programming model for I/O. **(3) I/O interlock:** Pages must sometimes be locked into memory, pages used for DMA from disk.
Page Replacement Algorithms: Optimal algorithm (un-implementable) replaces page which will not be used for longest period of time. FIFO, LRU and clock replacement (real LRU is expensive to implement), and MFU.
Locality of Reference - for a program to run efficiently, the system keeps a subset favoured pages in main memory, otherwise thrashing occurs (excessive paging activity) which results in poor processor utilisation. Locality of reference is when programs tend to request the same pages in space and time.
Working Set - set of pages being referenced in a time interval, OS temporarily maintains in memory pages outside the working set.
Local vs Global strategy – in local, each process gets fixed allocation of physical memory, need to pick up changes in working set size. In global, dynamically share memory between runnable processes, initially allocate memory proportional to process size, consider page fault frequency (PFF) to tune allocation and measure page faults/per sec and increase/decrease allocation.

File Systems

Introduction – a file is named collection of data of arbitrary size. Goals are: share information easily, allows concurrent access to data, organisation and management of data. File size is naturally variable, but space is allocated in fixed sizes called blocks. If block size is too large, it wastes space for small files and more memory is needed for buffer space. If block size is too small, it wasted space fort larger files, and introduces large overhead in terms of management of data and high file transfer time.
Contiguous File Allocation – places files at contiguous addresses on storage device. **Advantages**: successive records are physically adjacent. **Disadvantages**: external fragmentation, poor performance if file grows and shrinks over time, I/O overhead of relocating file if too large.
Block Linkage – places file data by linking them together. Need to search from start of list so many seeks, pointer space wasted in each block, and table can get very large.





User Directory

File	Location
A	7
B	5
C	2

Block allocation table

0	Free
1	Null
2	4
3	11
4	9
5	6
6	3
7	10
8	Free
9	14
10	Null
11	12
12	Null
13	Free
14	1

Physical blocks on secondary storage

Block 0 Free	Block 1 C (5)	Block 2 C (1)	Block 3 B (3)	Block 4 C (2)
Block 5 B (1)	Block 6 B (2)	Block 7 A (1)	Block 8 Free	Block 9 C (3)
Block 10 A (2)	Block 11 B (4)	Block 12 B (5)	Block 13 Free	Block 14 C (4)

Index Blocks/ Inodes – each file has one or more index blocks, where files directory entry points to index block, can be chained. Advantages over linked list are: searching can take place within the index block, index blocks can be cached, and index blocks can be placed close to corresponding data blocks, allowing for faster accesses. When a file is open, OS opens inode table and inode entry created in memory. Includes: disk device number, inode number, number of processes with opened file, major and minor device number, as well as direct, indirect, doubly indirect and triply indirect pointers to data blocks. Formula: **(B/P)^N * B** where B is block size, P is pointer size and N is level of indirection.
Free list – linked list of blocks storing locations of free blocks, newly freed blocks at the back, allocated blocks at the front. Low overhead for maintenance operations but higher file access times.
Bitmap – contains a bit in memory for each block, can quickly determine contiguous blocks at locations in secondary storage but need to search entire bitmap.
Directories – maps symbolic file names to locations on disk, flat systems are simple but no two files can have the same name, linear search results in poor performance. Hierarchical file systems organised as a tree, faster search times and names can be reused in separate subdirectories.
Links – reference to filke/directory in other part of file system, allows for alternative names. Hard links reference the address of the file while soft links reference the pathnames. Problems with file deletion and circular looping.
Mounting – combines many FS's into one namespace, allows for reference from single root directory. Mount point is directory in native FS assigned to root of mounted FS, FS's manage mounted directories with mount tables.

Device Management

Objectives – Fair access to shared devices, exploit parallelism of I/O for multiprogramming, and provide simple uniform view if I/O.
Character Devices – delivers or accepts stream of characters without regards to block size, not addressable (no seek).
Block Devices – Stores and transfers information in fixed block sizes.

```
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 ();
```

Code executed at the time the print system call is made

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

Interrupt service procedure for the printer

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

Code executed when a disk request is made

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

Interrupt service procedure

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

User-level I/O Software – **make I/O call, format I/O, spooling.** Some devices can only be used by one process at a time (printers), spooling will take print requests and add them to an intermediate disk to be dealt with later, allows for multiprogramming with dedicated I/O devices.
Device-independent I/O software – **Naming, protection, blocking, buffering, allocation.** Usually device independent, allows for uniform interface for a class of devices. 4 types of buffering: buffering in user space, kernel space, and kernel to user space and double buffering in kernel. When writes to kernel buffer, user process woken up and data copied into it. Circular buffering or double buffering if buffer is full. Actual I/O errors handled by driver. Block size is device independent to provide uniform view.
Device Drivers – **Set up device registers, check status.** Device specific code for controlling an I/O device. Positioned below the OS which defines a uniform interface for block and character devices.
Interrupt Handler - Wake up driver when I/O completed. Blocks and unblocks driver on I/O.

Disk Management

Addressing – modern disks use logical addressing, surface divided into zones, each track has numbered sector, done by manufacturer.

b – number of bytes to be transferred

N – number of bytes per track

r – rotation speed in revolutions per second

Seek time	t_{seek}
Latency time (rotational delay)	$t_{latency} = \frac{1}{2 \times r}$
Transfer time	$t_{transfer} = \frac{b}{N \times r}$
Total access time (t_{access})	$t_{seek} + t_{latency} + t_{transfer}$

Before a disk can be used, it must be formatted

- **Low level format**
  - Disk sector layout

Preamble	Data	ECC

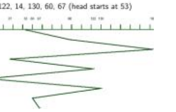
  - Cylinder skew
  - Interleaving
- **High level format**
  - Boot block
  - Free block list
  - Root directory
  - Empty file system

First Come First Served (FCFS)

No ordering of requests → random seek patterns

- OK for lightly-loaded disks
- But poor performance for heavy loads
- Fair scheduling

Queue: 98, 183, 37, 122, 14, 130, 60, 67 (head starts at 53)

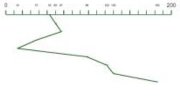


Shortest Seek Time First (SSTF)

Order requests according to shortest seek distance from current head position

- Discriminates against innermost/outermost tracks
- Unpredictable and unfair performance

Queue: 98, 183, 37, 122, 14, 130, 60, 67 (head starts at 53)

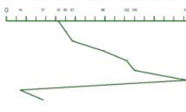


C-LOOK

Services requests in one direction only

- When head reaches innermost request, jump to outermost request
- Lower variance of requests on extreme tracks
- May delay requests indefinitely (though less likely)

Queue: 98, 183, 37, 122, 14, 130, 60, 67 (head starts at 53)



LOOK

Only change direction when reaching outermost/innermost request in preferred direction – variation on the SCAN (or elevator) scheduling algorithm

- Common scheduling algorithm
- Long delays for requests at extreme locations

Queue: 98, 183, 37, 122, 14, 130, 60, 67 (head starts at 53 and direction is towards 0)

N-Step SCAN

As for SCAN, but services only requests waiting when sweep began

- Requests arriving during sweep serviced during return sweep
- Doesn't delay requests indefinitely

Queue: 98, 183, 37, 122, 14, 130, 60, 67 (head starts at 53; direction → 0); requests 80, 140 arrive when head moving outwards