UNIT I OVERVIEW OF OPERATING SYSTEMS

Introduction - overview of operating system concepts - Process management and Scheduling, Memory management: partitioning, paging, segmentation, virtual memory, Device and File management.

Definition
A program that acts as an intermediary between a user of a computer and the computer hardware

Functions
- OS is a resource allocator
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a control program
  - Controls execution of programs to prevent errors and improper use of the computer

Computing Environments
- Traditional computing
  - PCs connected to a network with servers providing file and print service
  - Companies implement portals, to provide web accessibility to their internal servers
  - Network computers - wireless networks - firewalls
- Web based computing
  - Handheld PDAs, cell phones
  - load balancers
- Embedded Computers
  - Run embedded realtime OS
  - from car engines to VCRs / ovens

Interrupt Handling
- The operating system preserves the state of the CPU by storing registers and the PC
- Determines which type of interrupt has occurred:
  - polling
  - vectored interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

Interrupt timeline
STORAGE-DEVICE HIERARCHY

Performance of Various Levels of Storage

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>registers</td>
<td>cache</td>
<td>main memory</td>
<td>disk storage</td>
</tr>
<tr>
<td>Typical size</td>
<td>&lt; 1 KB</td>
<td>&gt; 16 MB</td>
<td>&gt; 16 GB</td>
<td>&gt; 100 GB</td>
</tr>
<tr>
<td>Implementation technology</td>
<td>custom memory with multiple ports, CMOS</td>
<td>on-chip or off-chip CMOS SRAM</td>
<td>CMOS DRAM</td>
<td>magnetic disk</td>
</tr>
<tr>
<td>Access time (ns)</td>
<td>0.25 – 0.5</td>
<td>0.5 – 25</td>
<td>80 – 250</td>
<td>5,000,000</td>
</tr>
<tr>
<td>Bandwidth (MB/sec)</td>
<td>20,000 – 100,000</td>
<td>5000 – 10,000</td>
<td>1000 – 5000</td>
<td>20 – 150</td>
</tr>
<tr>
<td>Managed by</td>
<td>compiler</td>
<td>hardware</td>
<td>operating system</td>
<td>operating system</td>
</tr>
<tr>
<td>Backed by</td>
<td>cache</td>
<td>main memory</td>
<td>disk</td>
<td>CD or tape</td>
</tr>
</tbody>
</table>
**Figure 1.2** A modern computer system.
**Types of Advanced OS**

- **Distributed Operating Systems**
  - OS for a network of autonomous computers connected by a communication network
  - The users view the entire system as a powerful monolithic computer system
  - When the program is executed in a distributed system, the user is not aware of where the program is executed or the location of the resources accessed

- **Multiprocessor Operating System**
  - A set of processors that share a set of physical memory blocks over an interconnection network
  - The users view the entire system as a powerful uniprocessor system
  - User is not aware of the presence of multiple processors and interconnection network

- **Database Operating Systems**
  - A database system must support the concept of transaction, operation to store, retrieve and manipulate large volume of data efficiently; primitives for concurrency control and system failure recovery.
  - To store temporary data and data retrieved from the secondary storage, it must have a buffer management scheme

- **Real-time Operating Systems**
  - In RTOS, jobs have completion deadlines.
  - In soft real-time systems, a job should have completed before its deadline to be of use.
  - In hard real-time systems, a job should have completed to avert a disaster.
  - The design should take care of scheduling jobs in such a way that a maximum number of jobs satisfy their deadlines.
**PROCESS STATES**

- Process - a program in execution, changes its state during execution
- The state of a process is defined in part by the current activity of that process.
- Each process may be in one of the following states:

<table>
<thead>
<tr>
<th>No</th>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>New</td>
<td>The process is being created</td>
</tr>
<tr>
<td>2</td>
<td>Running</td>
<td>Instructions are being executed</td>
</tr>
<tr>
<td>3</td>
<td>Waiting</td>
<td>The process is waiting for some event to occur (such as an I/O</td>
</tr>
<tr>
<td></td>
<td></td>
<td>completion or reception of a signal)</td>
</tr>
<tr>
<td>4</td>
<td>Ready</td>
<td>The process is waiting to be assigned to a processor</td>
</tr>
<tr>
<td>5</td>
<td>Terminated</td>
<td>The process has finished execution</td>
</tr>
</tbody>
</table>

- It is important to realize that only one process can be running on any processor at any instant.
- Many processes may be ready and waiting, however.

![Diagram of process state](image)

**Figure 3.2** Diagram of process state.

- The scheduler determines the process to be executed from the list

<table>
<thead>
<tr>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>Scheduler</td>
</tr>
</tbody>
</table>
## PROCESS CONTROL BLOCK

<table>
<thead>
<tr>
<th>pointer</th>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>process number</td>
<td></td>
</tr>
<tr>
<td>program counter</td>
<td></td>
</tr>
<tr>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>memory limits</td>
<td></td>
</tr>
<tr>
<td>list of open files</td>
<td></td>
</tr>
</tbody>
</table>

### CPU Switch From Process to Process

<table>
<thead>
<tr>
<th>process $P_0$</th>
<th>operating system</th>
<th>process $P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>executing</td>
<td>interrupt or system call</td>
<td>idle</td>
</tr>
<tr>
<td>idle</td>
<td>save state into PCB$_0$</td>
<td>executing</td>
</tr>
<tr>
<td>idle</td>
<td>reload state from PCB$_1$</td>
<td>idle</td>
</tr>
<tr>
<td>executing</td>
<td>interrupt or system call</td>
<td>save state into PCB$_1$</td>
</tr>
<tr>
<td>idle</td>
<td>reload state from PCB$_0$</td>
<td></td>
</tr>
</tbody>
</table>
**CONTIGUOUS MEMORY ALLOCATION**

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process.
- Thus, the degree of multiprogramming is bound by the number of partitions.

**Multiple-partition method**
- When a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- This method is no longer in use.

**Fixed-partition method**
- The OS keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a hole.
- When a process arrives and needs memory, we search for a hole large enough for this process.
- If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.
- At any given time, we have a list of available block sizes and the input queue. The OS can order the input queue according to a scheduling algorithm.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- It is used primarily in batch environments.

**Solution to dynamic storage-allocation problem**
- First fit. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough. Mostly Faster.
- Best fit. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- Worst fit. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

---

Initial Memory

| P1 | P2 |

Best Fit

| P1 | P2 | P3 |

Worst Fit

| P1 | P3 |

First Fit

| P1 | P2 | P3 |
**Steps to Handle Page Faults**

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

**Pure demand paging**: executing a process with no pages in memory

**Hardware support:**

**Page table**: This table has the ability to mark an entry invalid through a valid-invalid bit or a special value of protection bits.

**Secondary memory**: This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as swap-space.

A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.
WRITE SHORT NOTES ON VIRTUAL MEMORY

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- Programs can be larger than physical memory.
- This abstracts main memory into an extremely large, uniform array of storage, separating logical memory from physical memory.
- This also allows processes to share files easily and to implement shared memory.

Figure 9.1 Diagram showing virtual memory that is larger than physical memory.

- The instructions being executed must be in physical memory.
- In many cases, the entire program (in memory) is not needed.
  - Programs often have code to handle unusual error conditions (seldom used).
  - Arrays, lists, and tables are often allocated more memory than they actually need.
  - Certain options and features of a program may be used rarely.
- The ability to execute a program that is only partially in memory would offer many benefits:
  - A program would no longer be constrained by the amount of physical memory that is available (simplifying the programming task).
  - Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
  - Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.
Virtual address space, which is the programmers logical view of process memory storage
Virtual address spaces that include holes are known as sparse address spaces.
Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:

- System libraries can be shared by mapping them into the virtual address space of more than one process.
- Processes can also share virtual memory by mapping the same block of memory to more than one process.
- Process pages can be shared during a fork() system call, eliminating the need to copy all of the pages of the original (parent) process.
**RAID Levels**

(a) RAID 0: non-redundant striping.

(b) RAID 1: mirrored disks.

(c) RAID 2: memory-style error-correcting codes.

(d) RAID 3: bit-interleaved parity.

(e) RAID 4: block-interleaved parity.

(f) RAID 5: block-interleaved distributed parity.

(g) RAID 6: P + Q redundancy.
WRITE SHORT NOTES ON DEVICE AND FILE MANAGEMENT

Device Management
A process may need several resources to execute like main memory, disk drives, access to files, and so on.

If the resources are
- Available, they can be granted, and control can be returned to the user process.
- Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating system can be thought of as devices.

- Some of these devices are physical devices (for example, disk drives),
- while others can be thought of as abstract or virtual devices (for example, files).

Managed access to device
- A system with multiple users may first request the device, to ensure exclusive use of it.
- After we are finished with the device, we release it.
- These functions are similar to the open and close system calls for files.

Unmanaged access to devices
- Other operating systems allow unmanaged access to devices.
- potential hazard for device contention and perhaps deadlock

Device Operations
- Once the device has been requested (and allocated to us), we can read, write, and (possibly) reposition the device, just as we can with files.
- In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX, merge the two into a combined file-device structure.
- In this case, a set of system calls is used on both files and devices.
- Sometimes I/O devices are identified by special file names, directory placement, or file attributes.
- The user interface can also make files and devices appear to be similar even though the underlying system calls are dissimilar

System Calls Summary
- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

![Diagram](image.png)

**Figure 2.6** The handling of a user application invoking the open() system call.
File Management

- create and delete files.
  - Either system call requires the name of the file and perhaps some of the file's attributes.
- open and using
  - We may also read, write, or reposition (rewinding or skipping to the end of the file, for example)
- close the file
  - indicating that we are no longer using it.
- Same sets of operations for directories
- To determine the values of various attributes and perhaps to reset them if necessary.
- File attributes include
  - file name,
  - file type
  - protection codes
  - accounting information ...
- At least two system calls, get file attribute and set file attribute, are required
- Some operating systems provide many more calls, such as calls for file move and copy.
- Others might provide an API that performs those operations using code and other system calls
- Others might just provide system programs to perform those tasks.

Examples of file management functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating a file</td>
<td>CreateFile ()</td>
<td>open ()</td>
</tr>
<tr>
<td>Close a file</td>
<td>CloseHandle ()</td>
<td>close ()</td>
</tr>
<tr>
<td>Read from a file</td>
<td>ReadFile ()</td>
<td>read ()</td>
</tr>
<tr>
<td>Write to file</td>
<td>WriteFile ()</td>
<td>write ()</td>
</tr>
<tr>
<td>Position into a file</td>
<td>SetFilePointer()</td>
<td>seek ()</td>
</tr>
</tbody>
</table>
SCHEDULING ALGORITHMS

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

**First-Come, First-Served Scheduling**
- The process that requests the CPU first is allocated the CPU first.
- Implementation of FCFS policy is managed with a FIFO queue
- When a process enters the ready queue, its PCB is linked onto the tail of the queue
- When the CPU is free, it is allocated to the process at the head of the queue
- The running process is then removed from the queue
- Non-preemptive - CPU will not be released until the running process terminates or blocks with IO operation
- Advantage: Simple to write and understand
- Disadvantage: Average waiting time is often quite long
- Burst time (24, 3, 3)
  - Average waiting time is \((0 + 24 + 27)/3 = 17\) milliseconds
  - Convoy effect

<table>
<thead>
<tr>
<th></th>
<th>(P_1)</th>
<th>(P_2)</th>
<th>(P_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>24</td>
<td>27</td>
</tr>
</tbody>
</table>

**Shortest-Job-First Scheduling**
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- Shortest-next-CPU-burst algorithm because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
- Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.
- Process: \(P_1, P_2, P_3, P_4\)
  - Burst Time: 6,8,7,3
- Average waiting time is \((3 + 16 + 9 + 0)/4 = 7\) milliseconds

<table>
<thead>
<tr>
<th></th>
<th>(P_4)</th>
<th>(P_1)</th>
<th>(P_3)</th>
<th>(P_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>9</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

**Priority Scheduling**
- A priority is associated with each process
- CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order
- Priority scheduling can be either preemptive or nonpreemptive
- A major problem with priority scheduling algorithms is indefinite blocking or starvation.
- A solution to the problem of indefinite blockage of low-priority processes is aging
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order \(P_1, P_2, \cdots, P_s\), with the length of the CPU burst given in milliseconds:
Round-Robin Scheduling

- Designed specifically for time-sharing systems
- It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds in length.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- Context switch will be executed once the timer goes off or the process completes before the quantum.
- The average waiting time under the RR policy is often long.
- The RR approach is also called processor sharing.
- Smaller time quantum increases context switches.
- Turnaround time varies with the time quantum.
- Average waiting time with quantum of 3 ms.
  - P1 waits for 6 milliseconds (10 - 4),
  - P2 waits for 4 milliseconds,
  - P3 waits for 7 milliseconds.
  - Average waiting time is 17/3 = 5.66 milliseconds.

Multilevel Queue Scheduling

- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue, generally based on some property of the process.
- Each queue has its own scheduling algorithm.

Multilevel Feedback Queue Scheduling

- Allows a process to move between queues.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- A process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.
PAGING

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.
- Paging avoids external fragmentation and the need for compaction.
- Solves the problem of fitting memory chunks of varying sizes onto the backing store

Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called **frames**, and to divide a program's logical memory space into blocks of the same size called **pages**.
- Any page (from any process) can be placed into any available frame.
- The page table is used to look up what frame a particular page is stored in at the moment.
Hardware Support for Paging

- The TLB is very expensive, however, and therefore very small. (Not large enough to hold the entire page table.) It is therefore used as a cache device.
- Addresses are first checked against the TLB, and if the info is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated.
- If the TLB is full, then replacement strategies range from LRU to random.
- Some TLBs allow some entries to be wired down, which means that they cannot be removed from the TLB. Typically these would be kernel frames.
- Some TLBs store address-space identifiers, ASIDs, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Otherwise the TLB has to be flushed clean with every process switch.
- The percentage of time that the desired information is found in the TLB is hit ratio.

Fig 8.11 Paging hardware with TLB.

Transaction Look aside Buffer

- The page table can also help to protect processes from accessing memory that they shouldn’t, or their own memory in ways that they shouldn’t.
- A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or some combination of these sorts of things. Then each memory reference can be checked to ensure it is accessing the memory in the appropriate mode.
- Valid / invalid bits can be added to "mask off" entries in the page table that are not in use by the current process.
- Rather than waste memory by creating a full-size page table for every process, some systems use a page-table length register, PTLR, to specify the length of the page table.

Protection
Shared Pages

- Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames.
- This may be done with either code or data.
- Some systems also implement shared memory in this fashion

![Diagram of page table and page frames](image)

**Figure 8.12** Valid (v) or invalid (i) bit in a page table.

**Figure 8.13** Sharing of code in a paging environment.
SEGMENTATION

Basic Method
- Most users think of their memory in multiple segments, each dedicated to a particular use, such as code, data, the stack, the heap, etc.
- Memory segmentation supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.

Hardware
A segment table maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers.

Segmentation can also be combined with paging.

![Diagram of logical address space](image)

**Figure 8.18** User’s view of a program.

![Diagram of address translation](image)

**Figure 8.21** Logical to physical address translation in the Pentium.
Figure 8.19 Segmentation hardware.
PAGE REPLACEMENT ALGORITHMS

**Basic Page Replacement**

1. Find the location of the desired page on the disk, either in swap space or in the file system.
2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the victim frame.
   - Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
4. Restart the process that was waiting for this page

**Algorithms**

1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement
4. LRU-Approximation Page Replacement
5. Counting-Based Page Replacement

**FIFO Page Replacement**

- A simple and obvious page replacement strategy is FIFO, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim.
- Although FIFO is simple and easy, it is not always optimal, or even efficient
- Belady's anomaly, can occur in which increasing the number of frames available can actually increase the number of page faults that occur

**Optimal Page Replacement**

- Replace the page that will not be used for the longest time in the future
- Also called as OPT or MIN
- OPT cannot be implemented in practice, because it requires foretelling the future

**LRU Page Replacement**

- The page that has not been used in the longest time is the one that will not be used again in the near future
- LRU is considered a good replacement policy, and is often used
- Two simple approaches commonly used:
  - **Counters**
    - Every memory access increments a counter(taking care of overflowing), and the current value of this counter is stored in the page table entry for that page.
    - Page with the smallest counter value is the LRU page.
  - **Stack**
    - Whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack.
    - Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.

**LRU-Approximation Page Replacement**

- Many systems provide a reference bit for every entry in a page table, which is set anytime that page is accessed
• One bit of precision is enough to distinguish pages that have been accessed

**Additional-Reference-Bits Algorithm**

• Storing the most recent 8 reference bits for each page in an 8-bit byte in the page table entry
• At periodic intervals (clock interrupts), the OS takes over, and right-shifts each of the reference bytes by one bit.
• The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared.
• At any given time, the page with the smallest value for the reference byte is the LRU page

**Second-Chance Algorithm**

• A FIFO, except the reference bit is used to give pages a second chance at staying in the page table
• This algorithm is also known as the clock algorithm

**Enhanced Second-Chance Algorithm**

• Looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes
• First makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.

**Counting-Based Page Replacement**

• Based on counting the number of references that have been made to a given page

**Least Frequently Used, LFU**

• Replace the page with the lowest reference count

**Most Frequently Used, MFU**

• Replace the page with the highest reference count
ALLOCATION METHODS IN FILE SYSTEM

1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation

Contiguous Allocation

- Requires that all blocks of a file be kept together contiguously.
- Performance is very fast, because reading successive blocks of the same file generally requires no or very little movement of the disk heads.
- Storage allocation involves first fit, best fit, fragmentation problems etc.
- Problems arises when files grow, or if the exact size of a file is unknown at creation time:
  - Over-estimation increases external fragmentation and wastes disk space.
  - Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
  - If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable.
- A variation is to allocate file space in large contiguous chunks, called extents. When a file outgrows its original extent, then an additional one is allocated.
  - The high-performance files system Veritas uses extents to optimize performance.

![Contiguous Allocation Diagram]

**Figure 11.5** Contiguous allocation of disk space.
Linked Allocation

- Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.
- Allocating clusters of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.
- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.
- The File Allocation Table, FAT, used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.

Indexed Allocation

- Indexed Allocation combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.
- Entire index block must be allocated for each file, regardless of how many data blocks the file contains
- Linked Scheme
  - An index block is one disk block, which can be read and written in a single disk operation.
  - The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.
- Multi-Level Index
  - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
- Combined Scheme
This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed.

- The advantage of this scheme is that
- Small files up to 48K using 4K blocks, the data blocks are readily accessible
- Files up to about 4144K using 4K blocks are accessible with only a single indirect
- Huge files are still accessible using a relatively small number of disk accesses

![Indexed allocation of disk space.](image)

**Figure 11.8** Indexed allocation of disk space.

![The UNIX inode.](image)

**Figure 11.9** The UNIX inode.
HARD DISK SCHEDULING

**First-come, first-served (FCFS)**

Fair, but it generally does not provide the fastest service

**Shortest-seek-time-first (SSTF)**

Service all the requests close to the current head position, before moving the head far away to service other requests

**SCAN Scheduling**

The head continuously scans back and forth across the disk

**Circular SCAN (C-SCAN) scheduling**

Returns to the beginning of the disk, without servicing any requests on the return trip

**LOOK, C-LOOK Scheduling**

The arm goes only as far as the final request in each direction, **not to the end** of the disk

Queue = 98, 183, 37, 122, 14, 124, 65, 67; head starts at 53

Comments & Feedback

Thanks to my family members who supported me while I spent hours and hours to prepare this.

Your feedback is welcome at GHCRajan@gmail.com