Contents

Data structures and Algorithms

A data structure is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

BigO Notation

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

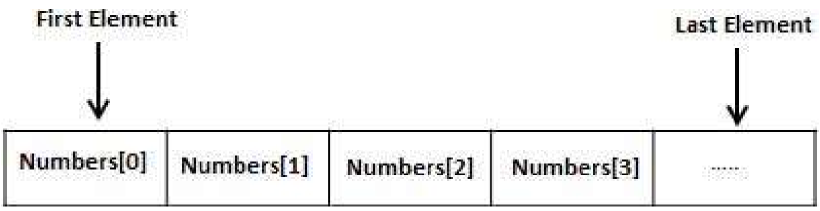
Sorting Algorithms	Space Complexity		Time Complexity	
	Worst case	Best case	Average case	Worst case
Bubble Sort	$O(1)$	$O(n)$	$O(n^2)$	$O(n^2)$
Heapsort	$O(1)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Insertion Sort	$O(1)$	$O(n)$	$O(n^2)$	$O(n^2)$
Mergesort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(\log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Selection Sort	$O(1)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
ShellSort	$O(1)$	$O(n)$	$O(n \log n^2)$	$O(n \log n^2)$
Smooth Sort	$O(1)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Tree Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Counting Sort	$O(k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$
Cubesort	$O(n)$	$O(n)$	$O(n \log n)$	$O(n \log n)$

Arrays

An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaration and array:

```
type arrayName [ arraySize ];
double balance[10];
```

Initializing an array:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Accessing an array element:

```
double salary = balance[9];
```

Iterations and Recursion

Iteration. This is a loop, in C that is done with a for, while or do-while loop.

Recursion is processing an element of a data structure and then calling the same function with the remaining elements or multiple times with different subsets of the remaining elements.

Linked list

https://www.hackerearth.com/practice/data-structures/linked-list/singly-linked-list/tutorial/

A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

Declaring a Linked list:

In C language, a linked list can be implemented using structure and pointers

```
struct LinkedList{
    int data;
    struct LinkedList *next;
 };
```

Creating a Node:

Let's define a data type of `struct LinkedList` to make code cleaner.

```
typedef struct LinkedList *node; //Define node as pointer of data type struct
LinkedList

node createNode(){
    node temp; // declare a node
    temp = (node)malloc(sizeof(struct LinkedList)); // allocate memory using
malloc()
    temp->next = NULL;// make next point to NULL
    return temp;//return the new node
}
```

typedef is used to define a data type in C.

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file `stdlib.h`.

sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to `malloc`.

Let's see how to **add a node to the linked list:**

```
node addNode(node head, int value){
    node temp,p;// declare two nodes temp and p
    temp = createNode();//createNode will return a new node with data = value
and next pointing to NULL.
    temp->data = value; // add element's value to data part of node
    if(head == NULL){
        head = temp;     //when linked list is empty
    }
    else{
        p  = head;//assign head to p
        while(p->next != NULL){
            p = p->next;//traverse the list until p is the last node.The last
node always points to NULL.
        }
        p->next = temp;//Point the previous last node to the new node
created.
    }
    return head;
}
```

This type of linked list is known as **simple or singly linked list**. A simple linked list can be traversed in only one direction from **head** to the last node.

The last node is checked by the condition:

```
p->next = NULL;
```

Here -> is used to access **next** sub element of node p. **NULL** denotes no node exists after the current node , i.e. its the end of the list.

Traversing the list:

The linked list can be traversed in a while loop by using the **head** node as a starting reference:

```
node p;
p = head;
while(p != NULL){
    p = p->next;
}
```

Stack

https://www.programiz.com/dsa/stack

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html

An array is a random-access data structure, where each element can be accessed directly and in constant time. A typical illustration of random access is a book - each page of the book can be open independently of others. Random access is critical to many algorithms, for example binary search.

A linked list is a sequential access data structure, where each element can be accessed only in particular order. A typical illustration of sequential access is a roll of paper or tape - all prior material must be unrolled in order to get to data you want.

In this note we consider a subcase of sequential data structures, so-called limited access data structures.

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

A stack is a recursive data structure. Here is a structural definition of a Stack:

a stack is either empty or
it consists of a top and the rest which is a stack;

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

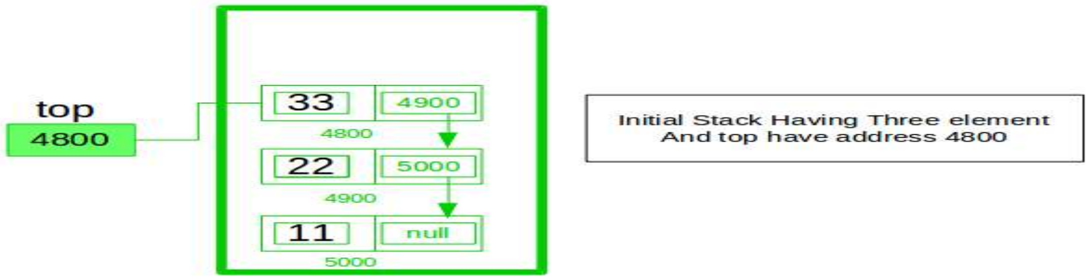
Mainly the following three basic operations are performed in the stack:

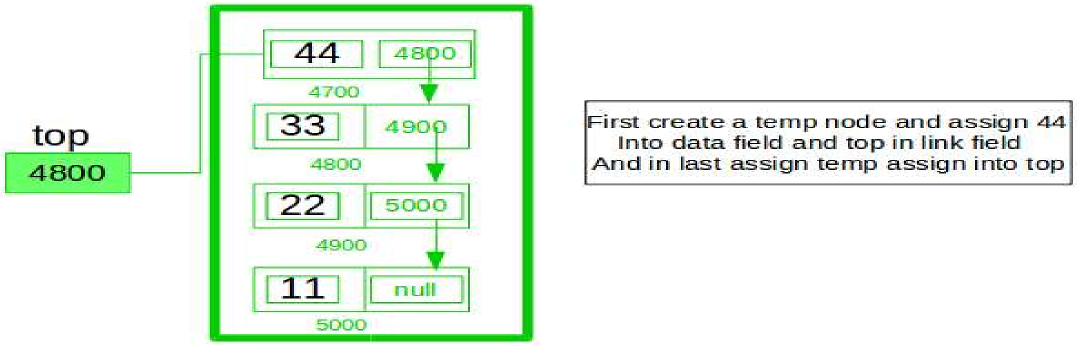
- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

Implementation:
There are two ways to implement a stack:
- Using array
- Using linked list

Stack Operations:
1. push() : Insert the element into linked list nothing but which is the top node of Stack.

2. pop() : Return top element from the Stack and move the top pointer to the second node of linked list or Stack.
3. peek(): Return the top element.
4. display(): Print all element of Stack.





Depth-First Search with a Stack

In depth-first search we go down a path until we get to a dead end; then we backtrack or back up (by popping a stack) to get an alternative path.

- Create a stack
- Create a new choice point
- Push the choice point onto the stack
- while (not found and stack is not empty)
  - Pop the stack
  - Find all possible choices after the last one tried
  - Push these choices onto the stack
- Return

Queue

A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Breadth-First Search with a Queue

In breadth-first search we explore all the nearest possibilities by finding all possible successors and enqueue them to a queue.

- Create a queue
- Create a new choice point
- Enqueue the choice point onto the queue
- while (not found and queue is not empty)
    - Dequeue the queue
    - Find all possible choices after the last one tried
    - Enqueue these choices onto the queue
- Return

Evaluating a Postfix Expression. We describe how to parse and evaluate a postfix expression.

1. We read the tokens in one at a time.
2. If it is an integer, push it on the stack
3. If it is a binary operator, pop the top two elements from the stack, apply the operator, and push the result back on the stack.

Consider the following postfix expression

```
5 9 3 + 4 2 * * 7 + *
```

Here is a chain of operations

```
Stack Operations              Output
--------------------------------------
push(5);                        5
push(9);                        5 9
push(3);                        5 9 3
push(pop() + pop())             5 12
push(4);                        5 12 4
push(2);                        5 12 4 2
push(pop() * pop())             5 12 8
push(pop() * pop())             5 96
push(7)                         5 96 7
push(pop() + pop())             5 103
push(pop() * pop())             515
```

Trees

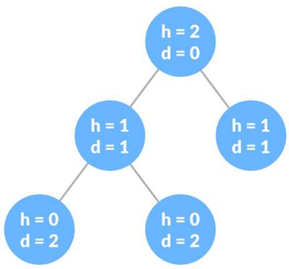
https://data-flair.training/blogs/binary-tree-in-c/

https://www.programiz.com/dsa/avl-tree

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.

Following are the important terms with respect to tree.

- Path − Path refers to the sequence of nodes along the edges of a tree.
- Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- Parent − Any node except the root node has one edge upward to a node called parent.
- Child − The node below a given node connected by its edge downward is called its child node.
- Leaf − The node which does not have any child node is called the leaf node.
- Subtree − Subtree represents the descendants of a node.
- Visiting − Visiting refers to checking the value of a node when control is on the node.
- Traversing − Traversing means passing through nodes in a specific order.
- Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- keys − Key represents a value of a node based on which a search operation is to be carried out for a node.



Height and depth of each node in a tree

Degree of a Node

The degree of a node is the total number of branches of that node.

Types of Tree

1. Binary Tree
2. Binary Search Tree
3. AVL Tree
4. B-Tree

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following

- Insert − Inserts an element in a tree/create a tree.
- Search − Searches an element in a tree.
- Preorder Traversal − Traverses a tree in a pre-order manner.
- Inorder Traversal − Traverses a tree in an in-order manner.
- Postorder Traversal − Traverses a tree in a post-order manner.

Tree Applications

- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

Working program

It is noted that above code snippets are parts of below C program. This below program would be working basic program for binary tree.

```
#include<stdlib.h>
#include<stdio.h>

struct bin_tree {
int data;
struct bin_tree * right, * left;
};
typedef struct bin_tree node;

void insert(node ** tree, int val)
{
    node *temp = NULL;
    if(!(*tree))
    {
        temp = (node *)malloc(sizeof(node));
        temp->left = temp->right = NULL;
        temp->data = val;
        *tree = temp;
        return;
    }

    if(val < (*tree)->data)
    {
        insert(&(*tree)->left, val);
    }
    else if(val > (*tree)->data)
    {
        insert(&(*tree)->right, val);
    }
```

```
}

void print_preorder(node * tree)
{
    if (tree)
    {
        printf("%d\n",tree->data);
        print_preorder(tree->left);
        print_preorder(tree->right);
    }

}

void print_inorder(node * tree)
{
    if (tree)
    {
        print_inorder(tree->left);
        printf("%d\n",tree->data);
        print_inorder(tree->right);
    }
}

void print_postorder(node * tree)
{
    if (tree)
    {
        print_postorder(tree->left);
        print_postorder(tree->right);
        printf("%d\n",tree->data);
    }
}

void deltree(node * tree)
{
    if (tree)
    {
        deltree(tree->left);
        deltree(tree->right);
        free(tree);
    }
}

node* search(node ** tree, int val)
{
    if(!(*tree))
    {
        return NULL;
    }

    if(val < (*tree)->data)
    {
        search(&((*tree)->left), val);
    }
    else if(val > (*tree)->data)
    {
```

```
        search(&((*tree)->right), val);
    }
    else if(val == (*tree)->data)
    {
        return *tree;
    }
}
```

Searching

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

Binary Search

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

- We start by comparing the element to be searched with the element in the middle of the list/array.
- If we get a match, we return the index of the middle element.
- If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
- If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
- If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.
- Binary Search is useful when there are large number of elements in an array and they are sorted.
- So a necessary condition for Binary search to work is that the list/array should be sorted.

Algorithm	Best case	Expected	Worst case
Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Merge sort	O(N log N)	O(N log N)	O(N log N)
Linear search	O(1)	O(N)	O(N)
Binary search	O(1)	O(log N)	O(log N)

Algorithm	Space complexity
Selection sort	O(1)
Merge sort	O(N)
Linear search	O(1)
Binary search	O(1)

SORTING ALGORITHMS

In order to have a good comparison between different algorithms we can compare based on the resources it uses: how much time it needs to complete, how much memory it uses to solve a problem or how many operations it must do in order to solve the problem:

Time efficiency: a measure of the amount of time an algorithm takes to solve a problem.

Space efficiency: a measure of the amount of memory an algorithm needs to solve a problem.

Complexity theory: a study of algorithm performance based on cost functions of statement counts.

Bubble Sort

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary

```
void bubbleSort(int ar[])
{
   for (int i = (ar.length - 1); i >= 0; i--)
   {
      for (int j = 1; j ≤ i; j++)
      {
         if (ar[j-1] > ar[j])
         {
              int temp = ar[j-1];
              ar[j-1] = ar[j];
              ar[j] = temp;
   } } } }
```

Example: Here is one step of the algorithm. The largest element - 7 - is bubbled to the top:

7, 5, 2, 4, 3, 9
5, **7, 2**, 4, 3, 9
5, 2, **7, 4**, 3, 9
5, 2, 4, **7, 3**, 9
5, 2, 4, 3, **7, 9**
5, 2, 4, 3, 7, 9

Selection Sort

The algorithm works by selecting the smallest unsorted item and then swapping it with the item in the next position to be filled.

The selection sort works as follows: you look through the entire array for the smallest element, once you find it you swap it (the smallest element) with the first element of the array. Then you look for the smallest element in the remaining array (an array without the first element) and swap it with the second element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on. Here is an example,

```
void selectionSort(int[] ar){
   for (int i = 0; i < ar.length-1; i++)
   {
      int min = i;
      for (int j = i+1; j < ar.length; j++)
            if (ar[j] < ar[min]) min = j;
      int temp = ar[i];
      ar[i] = ar[min];
      ar[min] = temp;
} }
```

Example.

29, 64, 73, 34, **20**,
20, **64**, 73, 34, **29**,
20, 29, **73**, **34**, 64
20, 29, 34, **73**, **64**
20, 29, 34, 64, 73

Insertion Sort

To sort unordered list of elements, we remove its entries one at a time and then insert each of them into a sorted part (initially empty):

```
void insertionSort(int[] ar)
{
   for (int i=1; i < ar.length; i++)
   {
      int index = ar[i]; int j = i;
      while (j > 0 && ar[j-1] > index)
      {
```

```
            ar[j] = ar[j-1];
            j--;
        }
        ar[j] = index;
} }
```

Example. We color a sorted part in green, and an unsorted part in black. Here is an insertion sort step by step. We take an element from unsorted part and compare it with elements in sorted part, moving form right to left.

29, 20, 73, 34, 64
29, 20, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 34, 73, 64
20, 29, 34, 64, 73

Mergesort

Merge-sort is based on the divide-and-conquer paradigm. It involves the following three steps:

- Divide the array into two (or more) subarrays
- Sort each subarray (Conquer)
- Merge them into one (in a smart way!)

Example. Consider the following array of numbers

27 10 12 25 34 16 15 31

divide it into two parts

27 10 12 25   34 16 15 31

divide each part into two parts

27 10   12 25   34 16   15 31

divide each part into two parts

27   10   12   25   34   16   15   31

merge (cleverly-!) parts

10 27   12 25   16 34   15 31

merge parts

10 12 25 27   15 16 31 34

merge parts into one

10 12 15 16 25 27 31 34

Quick Sort Algorithm

Quick Sort is also based on the concept of Divide and Conquer, just like merge sort. But in quick sort all the heavy lifting(major work) is done while dividing the array into subarrays, while in case of merge sort, all the real work happens during merging the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called partition-exchange sort. This algorithm divides the list into three main parts:

1. Elements less than the Pivot element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as pivot.

How Quick Sorting Works?

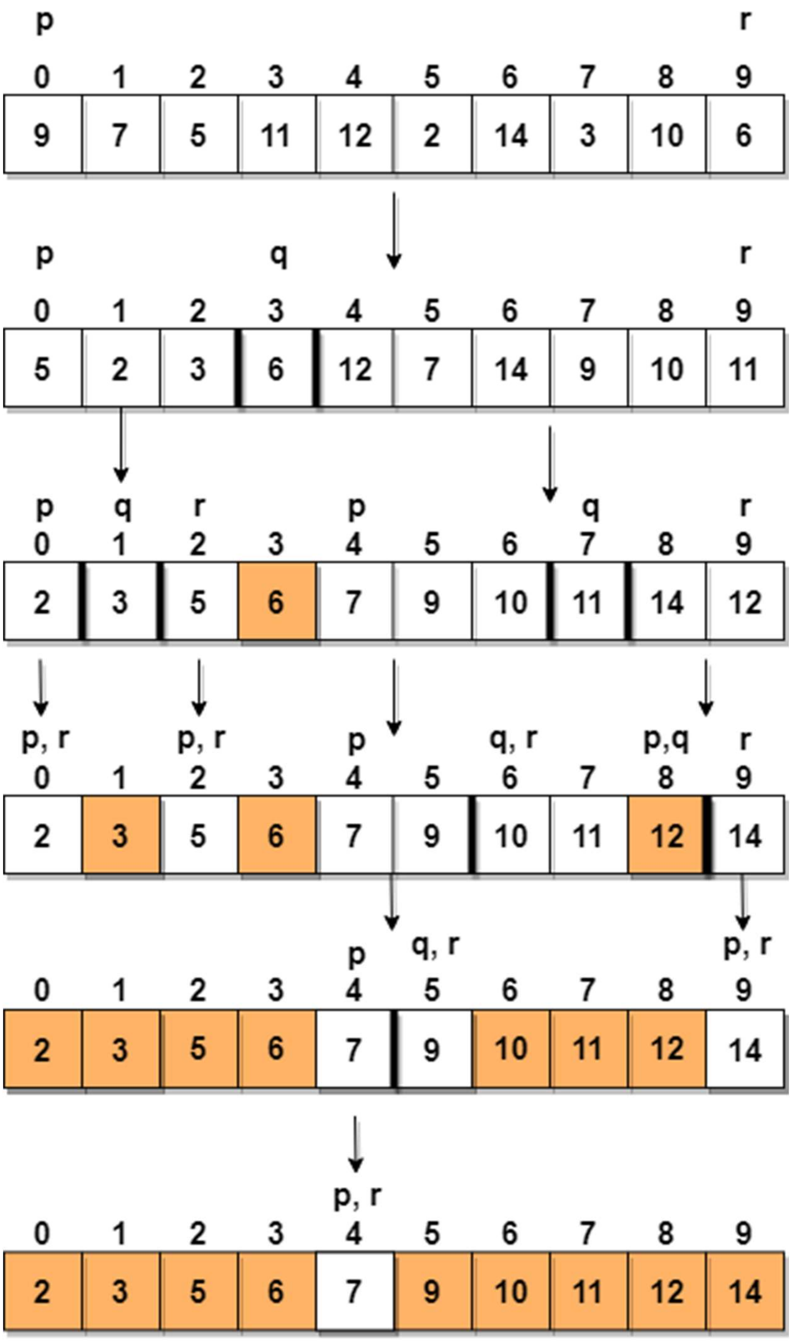
Following are the steps involved in quick sort algorithm:

1. After selecting an element as pivot, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this partitioning. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the pivot will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the pivot element will be at its final sorted position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of pivot and elements on the right of pivot, and we perform partitioning on them by choosing a pivot in the subarrays.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.

https://www.studytonight.com/data-structures/heap-sort



In step 1, we select the last element as the pivot, which is 6 in this case, and call for partitioning, hence re-arranging the array in such a way that 6 will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a pivot for them, in the above diagram, we chose 3 as pivot for the left subarray and 11 as pivot for the right subarray.

And we again call for partitioning.

Implementing Quick Sort Algorithm
Below we have a simple C program implementing the Quick sort algorithm:

```
// simple C program for Quick Sort
# include <stdio.h>

// to swap two numbers
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/*
    a[] is the array, p is starting index, that is 0,
    and r is the last index of array.
*/
void quicksort(int a[], int p, int r)
{
    if(p < r)
    {
        int q;
        q = partition(a, p, r);
        quicksort(a, p, q);
        quicksort(a, q+1, r);
    }
}

int partition (int a[], int low, int high)
{
    int pivot = arr[high];  // selecting last element as pivot
    int i = (low - 1);  // index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
```

```
}

// function to print the array
void printArray(int a[], int size)
{
    int i;
    for (i=0; i < size; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int main()
{
    int arr[] = {9, 7, 5, 11, 12, 2, 14, 3, 10, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    // call quickSort function
    quickSort(arr, 0, n-1);

    printf("Sorted array: n");
    printArray(arr, n);
    return 0;
}
```

Hash table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.

```
struct DataItem {
   int data;
   int key;
};
```

```
int hashCode(int key){
   return key % SIZE;
}
```

```
struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }
```

```
   return NULL;
}
```

```
void insert(int key,int data) {
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct
DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}
```

```
struct DataItem* delete(struct DataItem* item) {
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL) {

      if(hashArray[hashIndex]->key == key) {
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
```

```
    }

    return NULL;
}
```

Graph

Operating systems

https://www.tutorialspoint.com/operating_system/os_overview.htm

https://medium.com/cracking-the-data-science-interview/the-10-operating-system-concepts-software-developers-need-to-remember-480d0734d710

https://www.tldp.org/LDP/tlk/ds/ds.html - Linux Data structures

PROCESS MANAGEMENT

Processes

A process is basically a program in execution. When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data.

- Stack: The process Stack contains the temporary data such as method/function parameters, return address and local variables.
- Heap: This is dynamically allocated memory to a process during its run time.
- Text: This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
- Data: This section contains the global and static variables.

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below:

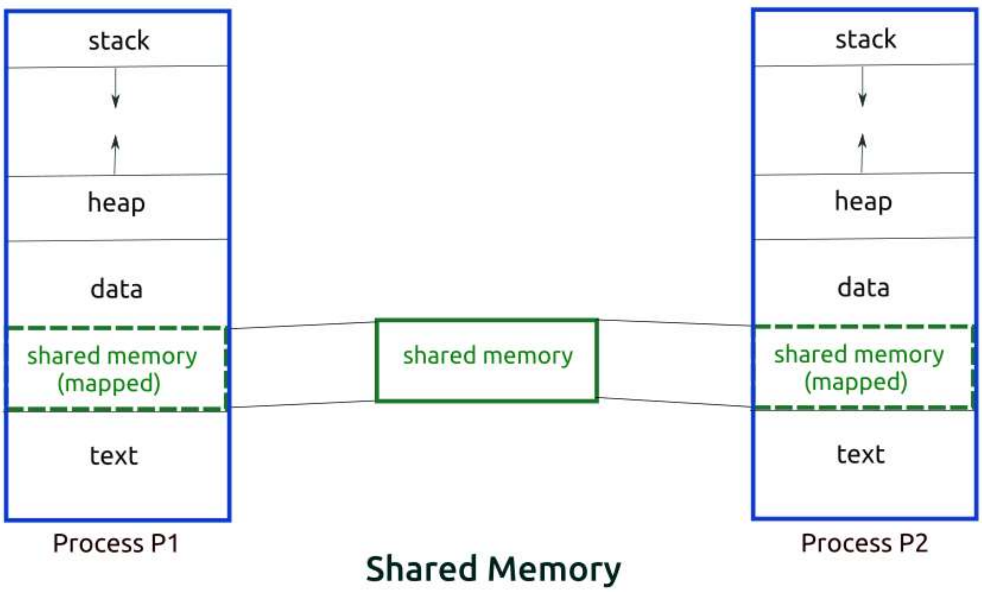
- Process State: The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- Process Privileges: This is required to allow/disallow access to system resources.
- Process ID: Unique identification for each of the process in the operating system.
- Pointer: A pointer to parent process.
- Program Counter: Program Counter is a pointer to the address of the next instruction to be executed for this process.
- CPU Registers: Various CPU registers where process need to be stored for execution for running state.
- CPU Scheduling Information: Process priority and other scheduling information which is required to schedule the process.
- Memory Management Information: This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
- Accounting Information: This includes the amount of CPU used for process execution, time limits, execution ID etc.
- IO Status Information: This includes a list of I/O devices allocated to the process.

Inter-process communication

A process can be of 2 types: Independent process and Co-operating process. An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity. Inter-process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways: Shared Memory and Message Parsing.

Shared Memory Method

- There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it.

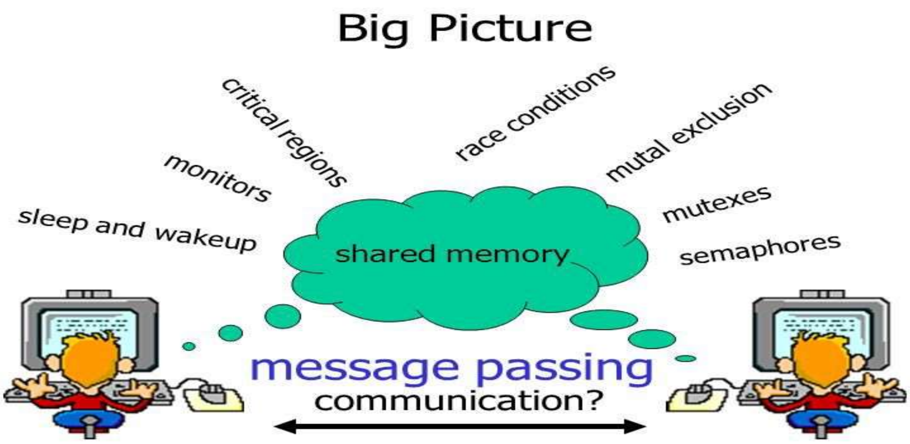


In the bounded buffer problem: First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer first check for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it.

Message Parsing Method

In this method, processes communicate with each other without using any kind of of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives. We need at least two primitives: send(message, destination) or send(message) and receive(message, host) or receive(message)



The message size can be of fixed size or of variable size. if it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: header and body.

The header part is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Each task_struct data structure describes a process or task in the system.

```
struct task_struct {
/* these are hardcoded - don't touch */
  volatile long        state;          /* -1 unrunnable, 0 runnable, >0
stopped */
  long                 counter;
  long                 priority;
  unsigned             long signal;
```

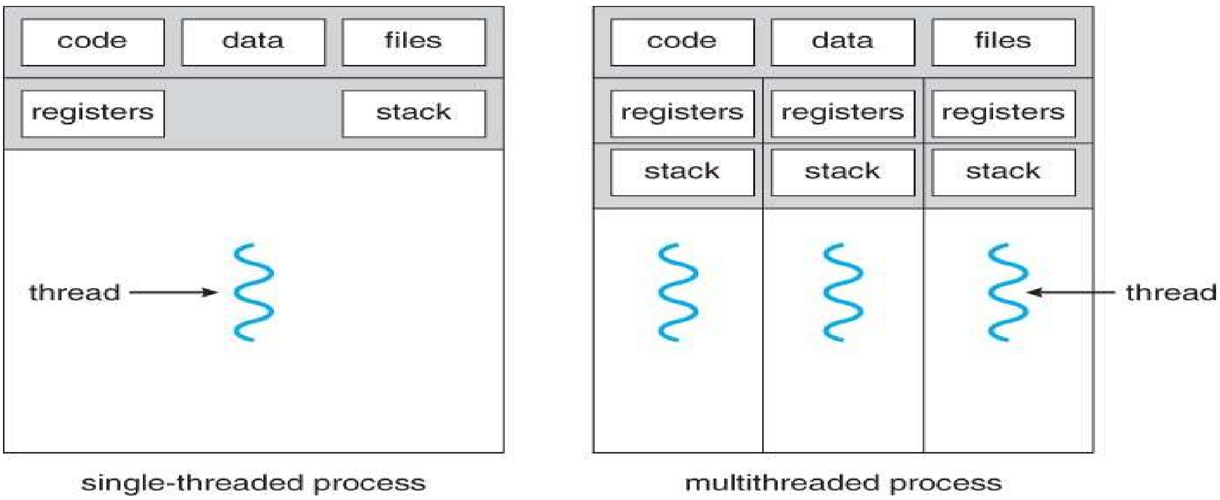
```
  unsigned              long blocked;   /* bitmap of masked signals */
  unsigned              long flags;     /* per process flags, defined below */
  int errno;
  long                  debugreg[8];    /* Hardware debugging registers */
  struct exec_domain    *exec_domain;
/* various fields */
  struct linux_binfmt   *binfmt;
  struct task_struct    *next_task, *prev_task;
  struct task_struct    *next_run,  *prev_run;
  unsigned long         saved_kernel_stack;
  unsigned long         kernel_stack_page;
  int                   exit_code, exit_signal;
  /* ??? */
  unsigned long         personality;
  int                   dumpable:1;
  int                   did_exec:1;
  int                   pid;
  int                   pgrp;
  int                   tty_old_pgrp;
  int                   session;
  /* boolean value for session group leader */
  int                   leader;
  int                   groups[NGROUPS];
  /*
   * pointers to (original) parent process, youngest child, younger sibling,
   * older sibling, respectively.  (p->father can be replaced with
   * p->p_pptr->pid)
   */
  struct task_struct    *p_opptr, *p_pptr, *p_cptr,
                        *p_ysptr, *p_osptr;
  struct wait_queue     *wait_chldexit;
  unsigned short        uid,euid,suid,fsuid;
  unsigned short        gid,egid,sgid,fsgid;
  unsigned long         timeout, policy, rt_priority;
  unsigned long         it_real_value, it_prof_value, it_virt_value;
  unsigned long         it_real_incr, it_prof_incr, it_virt_incr;
  struct timer_list     real_timer;
  long                  utime, stime, cutime, cstime, start_time;
/* mm fault and swap info: this can arguably be seen as either
   mm-specific or thread-specific */
  unsigned long         min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
  int swappable:1;
  unsigned long         swap_address;
  unsigned long         old_maj_flt;    /* old value of maj_flt */
  unsigned long         dec_flt;        /* page fault count of the last time
*/
  unsigned long         swap_cnt;       /* number of pages to swap on next
pass */
/* limits */
  struct rlimit         rlim[RLIM_NLIMITS];
  unsigned short        used_math;
  char                  comm[16];
/* file system info */
  int                   link_count;
  struct tty_struct     *tty;           /* NULL if no tty */
/* ipc stuff */
```

```
  struct sem_undo       *semundo;
  struct sem_queue      *semsleeping;
/* ldt for this task - used by Wine.  If NULL, default_ldt is used */
  struct desc_struct *ldt;
/* tss for this task */
  struct thread_struct tss;
/* filesystem information */
  struct fs_struct      *fs;
/* open file information */
  struct files_struct   *files;
/* memory management info */
  struct mm_struct      *mm;
/* signal handlers */
  struct signal_struct *sig;
#ifdef __SMP__
  int                   processor;
  int                   last_processor;
  int                   lock_depth;     /* Lock depth.
                                           We can context switch in and out
                                           of holding a syscall kernel lock...
*/
#endif
};
```

Threads

A thread is also called a lightweight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.



Advantages of Thread:

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Threads are implemented in the following 2 ways:

User Level Threads: User managed threads.

Advantages:

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages:

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads: Operating System managed threads acting on kernel, an operating system core.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.

3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Difference between User-Level & Kernel-Level Thread

S.N.	**User-Level Threads**	**Kernel-Level Thread**
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

https://users.cs.cf.ac.uk/Dave.Marshall/C/node30.html

https://stuff.mit.edu/afs/sipb/project/pthreads/include/pthread.h https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*)
message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*)
message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

```
$ cat /proc/1/status
Name: init Name of command run by this process
State: S (sleeping) State of this process
Tgid: 1 Thread group ID (traditional PID, getpid())
Pid: 1 Actually, thread ID (gettid())
PPid: 0 Parent process ID
TracerPid: 0 PID of tracing process (0 if not traced)
Uid: 0 0 0 0 Real, effective, saved set, and FS UIDs
Gid: 0 0 0 0 Real, effective, saved set, and FS GIDs
FDSize: 256 # of file descriptor slots currently allocated
```

```
Groups: Supplementary group IDs
VmPeak: 852 kB Peak virtual memory size
VmSize: 724 kB Current virtual memory size
VmLck: 0 kB Locked memory
VmHWM: 288 kB Peak resident set size
VmRSS: 288 kB Current resident set size
VmData: 148 kB Data segment size
VmStk: 88 kB Stack size
VmExe: 484 kB Text (executable code) size
VmLib: 0 kB Shared library code size
VmPTE: 12 kB Size of page table (since 2.6.10)
Threads: 1 # of threads in this thread's thread group
SigQ: 0/3067 Current/max. queued signals (since 2.6.12)
SigPnd: 0000000000000000 Signals pending for thread
ShdPnd: 0000000000000000 Signals pending for process (since 2.6)
SigBlk: 0000000000000000 Blocked signals
SigIgn: fffffffe5770d8fc Ignored signals
SigCgt: 00000000280b2603 Caught signals
CapInh: 0000000000000000 Inheritable capabilities
CapPrm: 00000000ffffffff Permitted capabilities
CapEff: 00000000fffffeff Effective capabilities
CapBnd: 00000000ffffffff Capability bounding set (since 2.6.26)
Cpus_allowed: 1 CPUs allowed, mask (since 2.6.24)
Cpus_allowed_list: 0 Same as above, list format (since 2.6.26)
Mems_allowed: 1 Memory nodes allowed, mask (since 2.6.24)
Mems_allowed_list: 0 Same as above, list format (since 2.6.26)
voluntary_ctxt_switches: 6998 Voluntary context switches (since 2.6.23)
nonvoluntary_ctxt_switches: 107 Involuntary context switches (since 2.6.23)
Stack usage: 8 kB Stack usage high-water mark (since 2.6.32)
```

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

```
#include <unistd.h>
int pipe(int filedes[2]);
```

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Process Synchronization

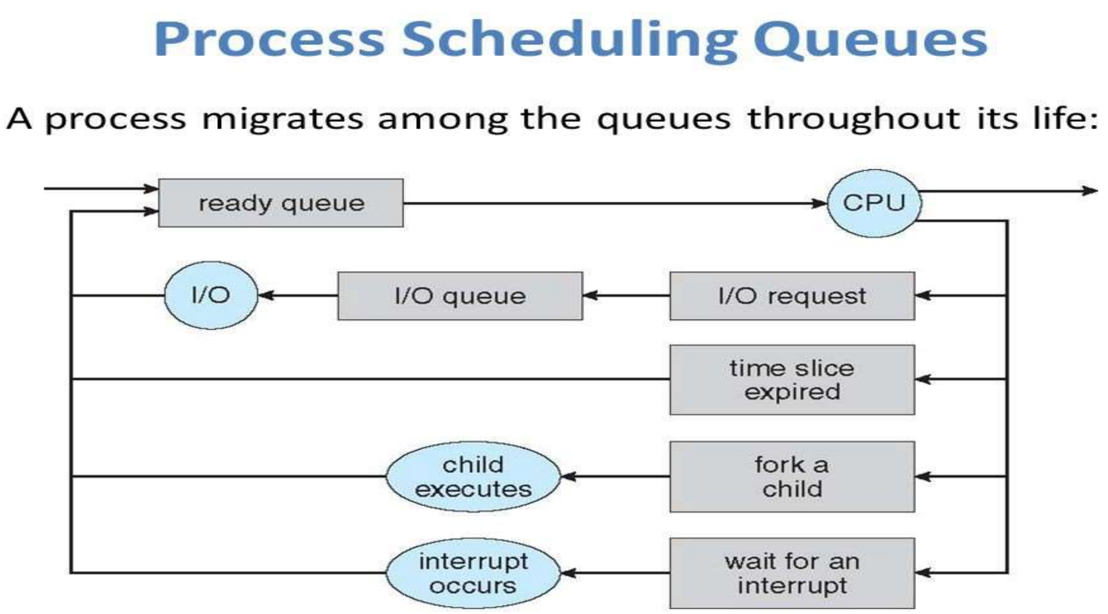
The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process based on a strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

The OS maintains all Process Control Blocks (PCBs) in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues:

- Job queue − This queue keeps all the processes in the system.
- Ready queue − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- Device queues − The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-state process model refers to running and non-running states:

- Running: When a new process is created, it enters into the system as in the running state.
- Not Running: Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use: Program Counter, Scheduling Information, Base and Limit Register Value, Currently Used Register, Changed State, I/O State Information, and Accounting Information.

CPU Scheduling

A Process Scheduler schedules different processes to be assigned to the CPU based on scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling

- Multiple-Level Queues Scheduling

MEMORY MANAGEMENT

Main Memory

Virtual Memory

STORAGE MANAGEMENT

File-System Interface

File-System Implementation

I/O Systems

Networking