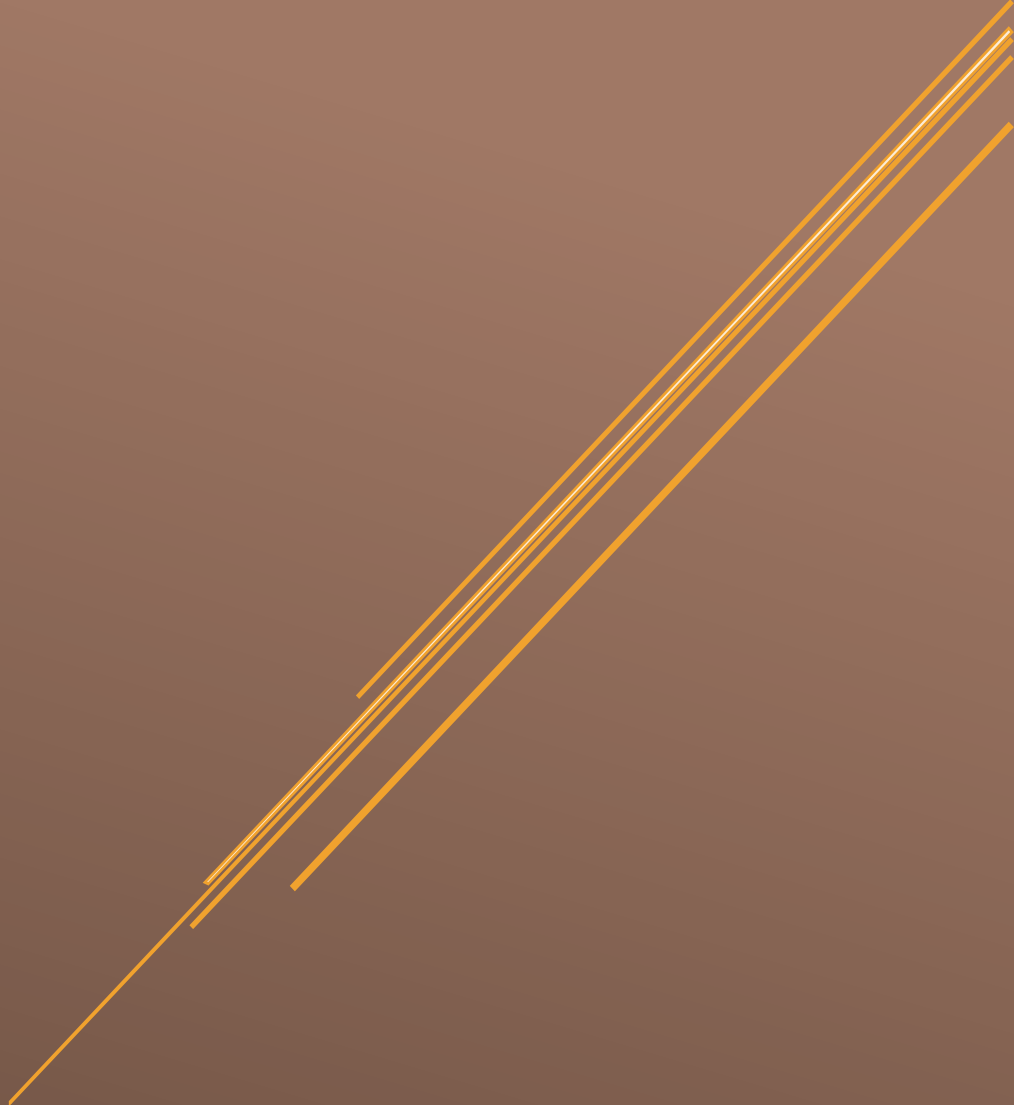




VIT<sup>®</sup>

Vellore Institute of Technology  
(Deemed to be University under section 3 of UGC Act, 1956)



# EMBEDDED PROGRAMMING ECE4025 (L41+L42)

Allen Ben Philipose – 18BIS0043

---

# TASK – 3

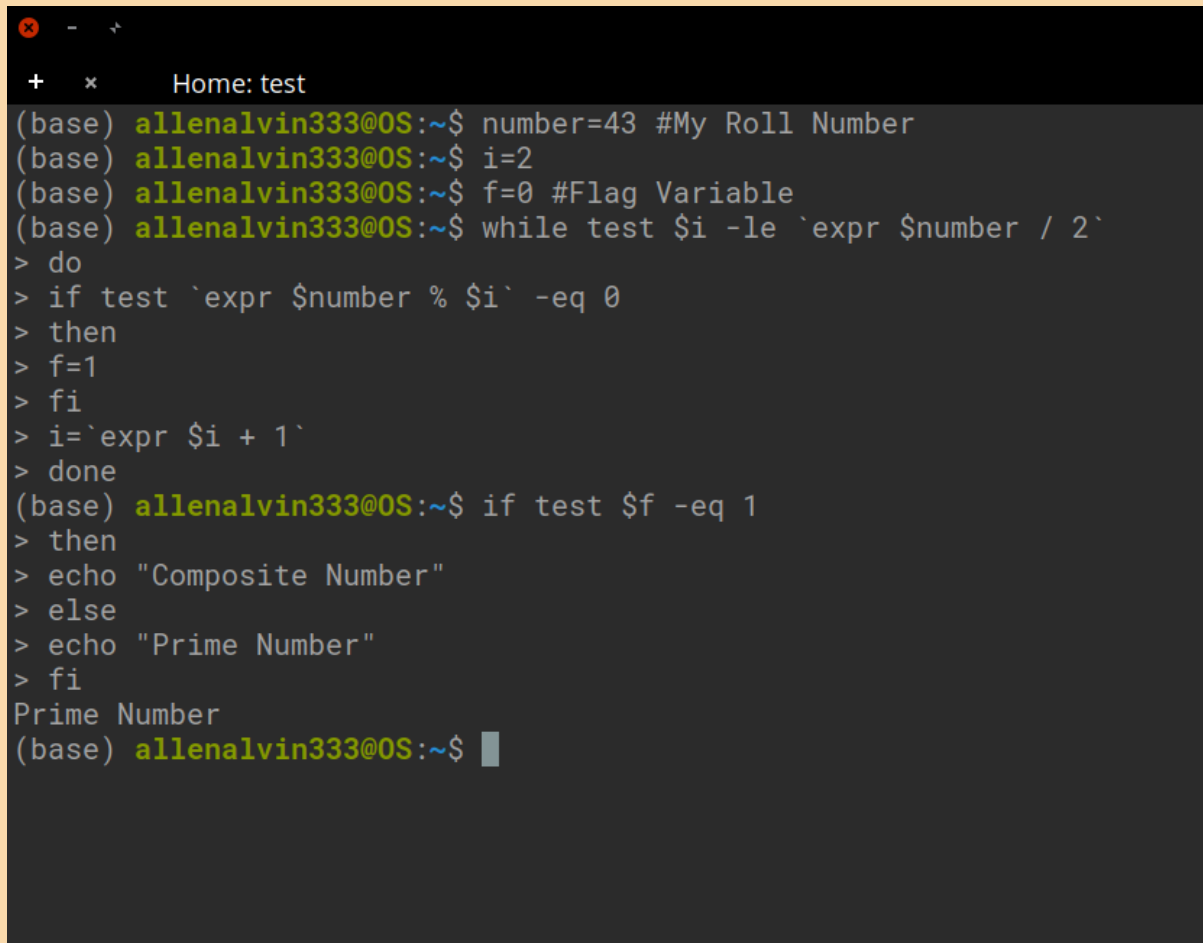
---

**a) Write a shell script program to find whether the number is prime number or not.**

```
number=43 #My Roll Number
i=2
f=0 #Flag Variable

while test $i -le `expr $number / 2`
do
if test `expr $number % $i` -eq 0
then
f=1
fi
i=`expr $i + 1`
done
if test $f -eq 1
then
echo "Composite Number"
else
echo "Prime Number"
fi
```

## Output



```
+ × Home: test
(base) allenalvin333@OS:~$ number=43 #My Roll Number
(base) allenalvin333@OS:~$ i=2
(base) allenalvin333@OS:~$ f=0 #Flag Variable
(base) allenalvin333@OS:~$ while test $i -le `expr $number / 2`
> do
> if test `expr $number % $i` -eq 0
> then
> f=1
> fi
> i=`expr $i + 1`
> done
(base) allenalvin333@OS:~$ if test $f -eq 1
> then
> echo "Composite Number"
> else
> echo "Prime Number"
> fi
Prime Number
(base) allenalvin333@OS:~$
```

## Explanation

- The variables – **number**, **i**, **f** – are declared and initialized.
- The while loop goes from the numerical value **2** to the value of **number/2** and the variable **i** will be considered as the loop variable.
- The if condition inside the loop checks whether the remainder of **number/i** is equal to 0. If yes, the flag variable **f** is made 1.
- Variable **i** is incremented after each iteration.
- Outside the loop, the if condition checks if the flag variable **f** is 1, if yes, “Composite Number” is printed, else, “Prime Number” is printed.

**b) Write a C code to reversing an already existing list of 10 nodes implemented using singly linked list. Clearly explain the declaration and initialization.**

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
static void reverse(struct Node** head_ref)
{
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}
```

```
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct
Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}
```

```
void printList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

```
int main()
{
    int k;
    struct Node* head = NULL;
    for(int a=1;a<=10;a++) {
```

```
        printf(" Enter number %d: ",a);  
        scanf("%d",&k);  
        push(&head, k);  
    }  
  
    printf("\n Reversed linked list\n");  
    printf(" ");  
    printList(head);  
    reverse(&head);  
    printf("\n Given Linked list \n");  
    printf(" ");  
    printList(head);  
}
```

## Output

```
Enter number 1: 01  
Enter number 2: 12  
Enter number 3: 23  
Enter number 4: 34  
Enter number 5: 45  
Enter number 6: 56  
Enter number 7: 67  
Enter number 8: 78  
Enter number 9: 89  
Enter number 10: 90  
  
Reversed linked list  
90  89  78  67  56  45  34  23  12  1  
Given Linked list  
1  12  23  34  45  56  67  78  89  90
```

## Explanation

A singly linked list is a type of linked list (sequence of elements in which every element has a link to its next element in the sequence) that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail).

Each element in a linked list is called a node. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.

The first node is called the head; it points to the first node of the list and helps us access every other element in the list. The last node, also sometimes called the tail, points to NULL which helps us in determining when the list ends.

1. Include all the header files which are used in the program.
2. Declare all the user defined functions.
3. Define a Node structure with two members data and next.
4. Define a Node pointer 'head' and set it to NULL.
5. Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Common Singly Linked List Operations:

### Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows:

- Inserting at Beginning of the list
- Inserting at End of the list
- Inserting at Specific location in the list

## Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows:

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

## Display

To display the elements of a single linked list, following steps are to be used:

1. Check whether list is Empty (head == NULL)
2. If it is Empty then, display 'List is Empty!!!' and terminate the function.
3. If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
4. Keep displaying temp → data with an arrow (--->) until temp reaches to the last node!
5. Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

These commands in the program are responsible for the reversing of the linked list. The pointers are changed, and the values are interchanged between the nodes. This process is done inside a while

```
next = current->next;  
current->next = prev;  
prev = current;  
current = next;
```

loop and is continued till the current node becomes NULL i.e., execution reached the end of the linked list and hence stopped.



**c) Write a C code to perform “Dequeue a node and Enqueue the same node” to an already existing QUEUE of 10 nodes implemented using singly linked list. Clearly explain the declaration and initialization.**

```
#include <stdio.h>
#include <stdlib.h>
struct QNode {
    int key;
    struct QNode* next;
};
struct Queue {
    struct QNode *front, *rear;
};
struct QNode* newNode(int k)
{
    struct QNode* temp = (struct
QNode*)malloc(sizeof(struct QNode));
    temp->key = k;
    temp->next = NULL;
    return temp;
}
struct Queue* createQueue()
```

```
{
    struct Queue* q = (struct
Queue*)malloc(sizeof(struct Queue));
    q->front = q->rear = NULL;
    return q;}
void enqueue(struct Queue* q, int k)
{
    struct QNode* temp = newNode(k);
    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }
    q->rear->next = temp;
    q->rear = temp;
}
int dequeue(struct Queue* q)
{
    if (q->front == NULL)
        return -1;
    struct QNode* temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL)
        q->rear = NULL;
```

```
        return temp->key;
        free(temp);
    }
int main()
{
    int k;
    struct Queue* q = createQueue();
    for(int a=1;a<=10;a++) {
        printf(" Enter number %d: ",a);
        scanf("%d",&k);
        enqueue(q, k);
    }
    int q1=deQueue(q);
    enqueue(q,q1);
    printf(" Dequeued node: %d \n", q1);
    printf(" Enqueued node: %d", q1);
    return 0;
}
```

## Output

```
Enter number 1: 12
Enter number 2: 21
Enter number 3: 23
Enter number 4: 32
Enter number 5: 34
Enter number 6: 43
Enter number 7: 45
Enter number 8: 54
Enter number 9: 56
Enter number 10: 64
Dequeued node: 12
Enqueued node: 12
```

## Explanation

Queue implemented using linked list can organize as many data values as we want. In a linked queue, each node of the queue consists of two parts i.e., data part and the link part. Each element of the queue points to its immediate next element in the memory.

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end, respectively. If front and rear both are NULL, it indicates that the queue is empty.

## **Enqueue (Insert)**

The insert operation appends the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

There are two scenarios of inserting a new node into the linked queue. In the first scenario, we insert element into an empty queue. In this case, the condition `front = NULL` becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL. In the second case, the queue contains more than one element. The condition

`front = NULL` becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node. We also need to make the next pointer of rear point to NULL.

## **Dequeue (Deletion)**

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition `front == NULL` becomes true if the list is empty, in this case, we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front.

**d) Write a C program to implement STACK operations using Linked list.**

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

void push();
void pop();
void display();

struct node
{
    int val;
    struct node *next;
};

struct node *head;

int main ()
{
    int choice=0;
    printf(" Stack operations using linked list");

    while(choice!=4)
    {
        printf("\n\n Select...\n");
        printf("\n 1.Push\n 2.Pop\n 3.Show\n 4.Exit");
        printf("\n Enter: ");
```

```
scanf("%d",&choice);
switch(choice)
{
    case 1: {
        push();
        break;
    }
    case 2: {
        pop();
        break;
    }
    case 3: {
        display();
        break;
    }
    case 4: {
        printf(" Exiting....");
        break;
    }
    default: {
        printf(" Invalid Choice\n");
    }
};
```

```
    }  
}  
void push ()  
{  
    int val;  
    struct    node    *ptr    =    (struct  
node*)malloc(sizeof(struct node));  
    if(ptr == NULL)  
    {  
        printf(" Unable to push");  
    }  
    else  
    {  
        printf(" Enter the value: ");  
        scanf("%d",&val);  
        if(head==NULL)  
        {  
            ptr->val = val;  
            ptr -> next = NULL;  
            head=ptr;  
        }  
        else  
        {  

```



```
        ptr->val = val;
        ptr->next = head;
        head=ptr;

    }

    printf(" Item pushed");

}

void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf(" Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
    }
}
```

```
        printf(" Item popped");
    }
}
void display()
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
        printf(" Stack is empty\n");
    }
    else
    {
        printf(" Printing\n");
        while(ptr!=NULL)
        {
            printf(" %d\n",ptr->val);
            ptr = ptr->next;
        }
    }
}
```

## Output

```
Stack operations using linked list
```

```
Select...
```

```
1.Push
```

```
2.Pop
```

```
3.Show
```

```
4.Exit
```

```
Enter: 2
```

```
Underflow
```

```
Select...
```

```
1.Push
```

```
2.Pop
```

```
3.Show
```

```
4.Exit
```

```
Enter: 1
```

```
Enter the value: 20
```

```
Item pushed
```

```
Select...
```

```
1.Push
```

```
2.Pop
```

```
3.Show
```

```
4.Exit
```

```
Enter: 1
```

```
Enter the value: 25
```

```
Item pushed
```

```
Select...
```

```
1.Push
```

```
2.Pop
```

```
3.Show
```

```
4.Exit
```

```
Enter: 3
```

```
Printing
```

```
25
```

```
20
```

```
Select...
```

```
1.Push
```

```
2.Pop
```

```
3.Show
```

```
4.Exit
```

```
Enter: 2
```

```
Item popped
```

```
Select...
```

```
1.Push
```

```
2.Pop
```

```
3.Show
```

```
4.Exit
```

```
Enter: 1
```

```
Enter the value: 45
```

```
Item pushed
```

```
Select...
```

```
1.Push
```

```
2.Pop
```

```
3.Show
```

```
4.Exit
```

```
Enter: 3
```

```
Printing
```

```
45
```

```
20
```

```
Select...
```

```
1.Push
```

```
2.Pop
```

```
3.Show
```

```
4.Exit
```

```
Enter: 4
```

```
Exiting....
```

## Explanation

Stacks can be easily implemented using a linked list. Stack is a data structure to which a data can be added using the push() method and data can be removed from it using the pop() method. With Linked list, the push operation can be replaced by the addAtFront() method of linked list and pop operation can be replaced by a function which deletes the front node of the linked list.

First, a class node is created. This is the Linked list node class which will have data in it and a node pointer to store the address of the next node element. Then the stack class is to be defined.

## Inserting Data in Stack

To insert an element into the stack, a node is to be created and placed in front of the list. Whenever the push() function is called, a new node will get added to the list in the front, which is exactly how a stack behaves.

To push an element, the following steps are involved:

- Create a node first and allocate memory to it.
- If the list is empty, then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
- If there are some nodes in the list already, then we must add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

*Time Complexity :  $O(1)$*

## Removing Element from Stack

To do this, simply delete the first node, and make the second node as the head of the list. Deleting a node from the top of stack is referred to as pop operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation.

- Check for the underflow condition: The underflow condition occurs when a pop operation is attempted on an already empty stack. The stack will be empty if the head pointer of the list points to null.
- Adjust the head pointer accordingly: In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

*Time Complexity:  $O(n)$*

## Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, following steps must be done:

- Copy the head pointer into a temporary pointer.
- Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

*Time Complexity:  $O(n)$*

### **e) Explain CPU Scheduling with an example.**

CPU scheduling is a process that allows one process to use the CPU while the execution of another process is on hold (in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast, and fair.

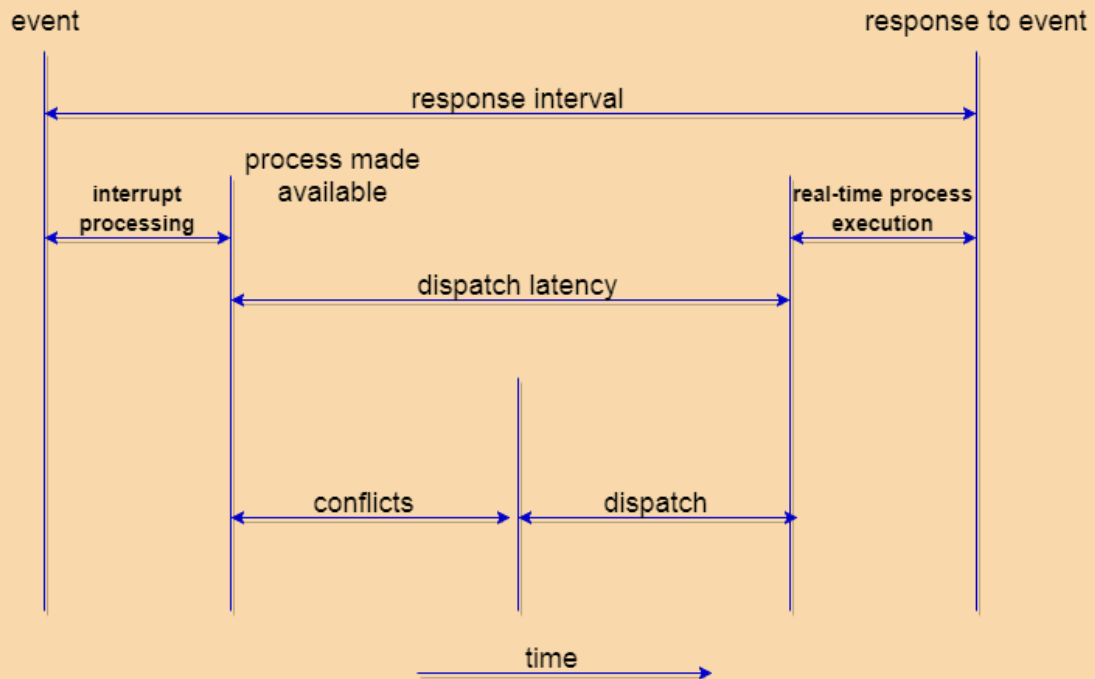
Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute and allocates the CPU to one of them.

### **Dispatcher**

Another component involved in the CPU scheduling function is the Dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program from where it left last time.

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another process is known as the Dispatch Latency.



## Types of CPU Scheduling

- When a process switches from the running state to the waiting state (for I/O request or invocation of wait for the termination of one of the child processes).
- When a process switches from the running state to the ready state (for example, when an interrupt occurs).
- When a process switches from the waiting state to the ready state (for example, completion of I/O).
- When a process terminates.

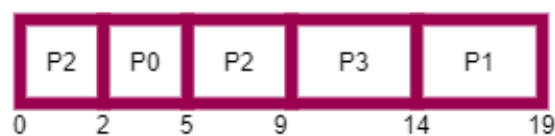
In circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however in circumstances 2 and 3. When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is non-preemptive; otherwise, the scheduling scheme is preemptive.



## Preemptive Scheduling

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. Thus, this type of scheduling is used mainly when a process switches either from running state to ready state or from waiting state to ready state.

Process	Arrival time	CPU Burst Time (in millisecond)
P0	2	3
P1	3	5
P2	0	6
P3	1	5



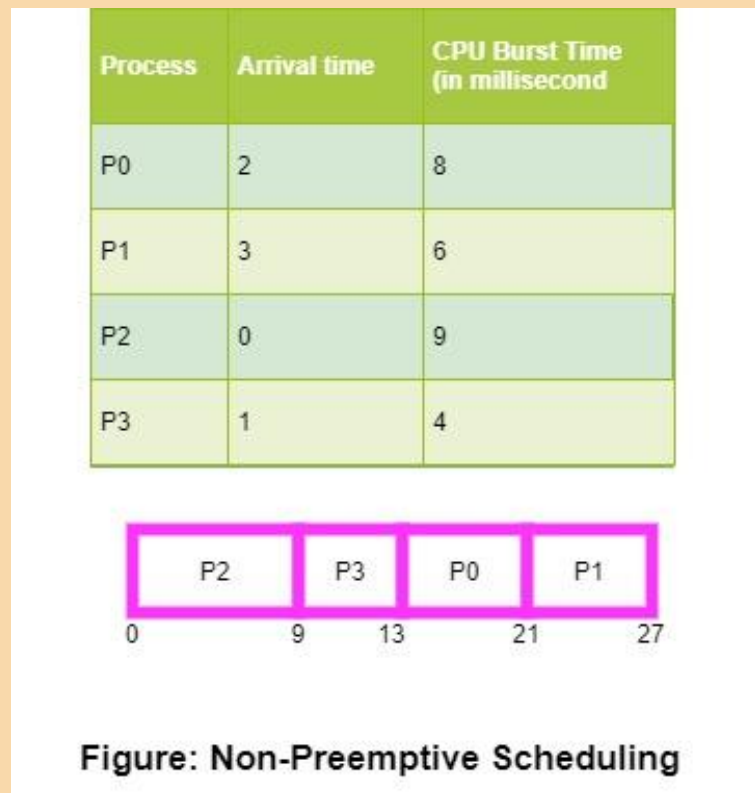
**Figure: Preemptive Scheduling**

The resources (that is CPU cycles) are mainly allocated to the process for a limited amount of time and then are taken away, and after that, the process is again placed back in the ready queue in the case if that process still has a CPU burst time remaining. That process stays in the ready queue until it gets the next chance to execute.

Some Algorithms that are based on preemptive scheduling are Round Robin Scheduling (RR), Shortest Remaining Time First (SRTF), Priority (preemptive version) Scheduling, etc.

## Non-Preemptive Scheduling

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems. It is the only method that



can be used on certain hardware platforms because It does not require the special hardware (for example a timer) needed for preemptive scheduling.

In non-preemptive scheduling, it does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then after that it can allocate the CPU to any other process.

Some Algorithms based on non-preemptive scheduling are: Shortest Job First (SJF basically non-preemptive) Scheduling and Priority (non- preemptive version) Scheduling, etc.

## Example - First Come First Serve Scheduling

In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.

- First Come First Serve, is just like FIFO (First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.
- This is used in Batch Systems.
- It is easy to understand and implement programmatically, using a Queue data structure, where a new process enters through the tail of the queue, and the scheduler selects process from the head of the queue.
- A perfect real-life example of FCFS scheduling is buying tickets at ticket counter.
- It is Non-Preemptive algorithm, which means the process priority does not matter. If a process with very least priority is being executed, more like daily routine backup process, which takes more time, and suddenly, some other high priority process arrives, like interrupt to avoid system crash, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.
- Not optimal Average Waiting Time. AWT or Average waiting time is the average of the waiting times of the processes in the queue, waiting for the scheduler to pick them for execution.
- Resource's utilization in parallel is not possible, which leads to Convoy Effect, and hence poor resource (CPU, I/O etc) utilization.