

### **PROGRAM OBJECTIVE :**

1. Demonstration of FORK() System Call

### **PROGRAM CODE :**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    // Create a child process
    pid = fork();
    if (pid < 0) {
        // fork failed
        perror("Fork failed");
        return 1;
    }
    else if (pid == 0) {
        printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());
    }
    else {
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
    }
    return 0;
}
```

### **OUTPUT :**

Parent process: PID = 282, Child PID = 283

Child process: PID = 283, Parent PID = 282

### **PROGRAM OBJECTIVE :**

2. Parent Process Computes the SUM OF EVEN and Child Process Computes the sum of ODD NUMBERS using fork

### **PROGRAM CODE :**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h> // For wait() function
int sumOfEven(int n) {
    int sum = 0;
    for (int i = 2; i <= n; i += 2) {
        sum += i;
    }
    return sum;
}
int sumOfOdd(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i += 2) {
        sum += i;
    }
    return sum;
}
int main() {
    int n;
    printf("Enter the upper limit (n): ");
    scanf("%d", &n);
    pid_t pid = fork();
    if (pid < 0) {
        printf("Fork failed!\n");
        return 1;
    }
    // Child process (computes sum of odd numbers)
    else if (pid == 0) {
        int oddSum = sumOfOdd(n);
        printf("Child Process (PID: %d) - Sum of odd numbers: %d\n", getpid(), oddSum);
    }
    // Parent process (computes sum of even numbers)
    else {
        // Wait for the child process to
        wait(NULL);
        int evenSum = sumOfEven(n);
        printf("Parent Process (PID: %d) - Sum of even numbers: %d\n", getpid(), evenSum);
    }
    return 0;
}
```

### **OUTPUT :**

Enter the upper limit (n): 10

Child Process (PID: 359) - Sum of odd numbers: 25

Parent Process (PID: 357) - Sum of even numbers: 30

## **PROGRAM OBJECTIVE :**

### 3. Demonstration of WAIT() System Call

## **PROGRAM CODE :**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h> // Required for wait() system call
int main() {
    // Creating a new process using fork()
    pid_t pid = fork();
    // Check if fork() failed
    if (pid < 0) {
        printf("Fork failed!\n");
        return 1;
    }
    // Child process
    else if (pid == 0) {
        printf("Child Process (PID: %d) is running...\n", getpid());
        sleep(2); // Simulating child process work by sleeping for 2 seconds
        printf("Child Process (PID: %d) has finished execution.\n", getpid());
    }
    // Parent process
    else {
        printf("Parent Process (PID: %d) is waiting for the child process to finish ...\n",
getpid());
        int a=wait(NULL); // Parent process waits for the child process to finish
        printf("Child process after wait returning process id %d\n",a);
        printf("Parent Process (PID: %d) resumes after the child process has finished.\n",
getpid());
    }
    return 0;
}
```

## **OUTPUT :**

Parent Process (PID: 410) is waiting for the child process to finish...

Child Process (PID: 411) is running...

Child Process (PID: 411) has finished execution.

Child process after wait returning process id 411

Parent Process (PID: 410) resumes after the child process has finished.

## **PROGRAM OBJECTIVE :**

### **4. Implementation of ORPHAN PROCESS & ZOMBIE PROCESS**

## **PROGRAM CODE :**

### **Orphan process program**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    else if (pid == 0) {
        printf("Child: PID = %d, parent PID = %d\n", getpid(), getppid());
        sleep(3); // Child sleeps to stay alive after parent exits
        printf("Child after sleep: PID = %d, parent PID = %d\n", getpid(), getppid());
    }
    else {
        printf("Parent: PID = %d, exiting immediately\n", getpid());
        exit(0); // Parent exits immediately, orphaning child
    }
    return 0;
}
```

### **Zombie process program**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        return 1;
    }
    if (pid == 0) {
        printf("Child: PID = %d\n", getpid());
        exit(0);
    }
    else {
        printf("Parent: PID = %d, sleeping to keep zombie child...\n", getpid());
        sleep(30); // Sleep so you can observe the zombie
        printf("Parent exiting.\n");
    }
    return 0;
}
```

## **OUTPUT :**

*For orphan process :*

Parent: PID = 589, exiting immediately

Child: PID = 590, parent PID = 589

Child after sleep: PID = 590, parent PID = 1

*For zombie process :*

Parent: PID = 649, sleeping to keep zombie child...

Child: PID = 650

Parent exiting.

## **PROGRAM OBJECTIVE :**

### **5. Implementation of PIPE.**

## **PROGRAM CODE :**

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
int main() {
    int pipefd[2];
    pid_t pid;
    char *messages[] = {
        "Message 1 from parent",
        "Message 2 from parent",
        "Message 3 from parent"
    };
    int num_messages = sizeof(messages) / sizeof(messages[0]);
    if (pipe(pipefd) == -1) {
        perror("pipe");
        return 1;
    }
    pid = fork();
    if (pid < 0) {
        perror("fork");
        return 1;
    }
    if (pid > 0) {
        // Parent Process
        close(pipefd[0]); // Close unused read end
        for (int i = 0; i < num_messages; i++) {
            write(pipefd[1], messages[i], strlen(messages[i]) + 1); // +1 to include null
            terminator
            sleep(1); // Simulate delay between messages
        }
        close(pipefd[1]); // Done writing
        wait(NULL); // Wait for child to finish
    }
    else {
        // Child Process
        close(pipefd[1]); // Close unused write end
        char buffer[100];
        while (read(pipefd[0], buffer, sizeof(buffer)) > 0) {
            printf("Child received: %s\n", buffer);
        }
        close(pipefd[0]);
    }
    return 0;
}
```

## **OUTPUT :**

Child received: Message 1 from parent

Child received: Message 2 from parent

Child received: Message 3 from parent

## **PROGRAM OBJECTIVE :**

### 6. Implementation of FIFO

## **PROGRAM CODE :**

### **Reader.c**

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
int main() {
    const char *fifo = "/home/ubuntu/Desktop/fifo/my";
    // Create FIFO if it doesn't exist (ignore errors)
    mkfifo(fifo, 0666);
    char buf[100];
    int fd = open(fifo, O_RDONLY);
    if (fd == -1) {
        perror("open reader");
        return 1;
    }
    int n = read(fd, buf, sizeof(buf) - 1);
    printf("%d",n);
    if (n > 0) {
        buf[n] = '\0';
        printf("Reader got: %s\n", buf);
    }
    close(fd);
    return 0;
}
```

### **Writer.c**

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
int main() {
    const char *fifo = "/home/ubuntu/Desktop/fifo/my";
    // Create FIFO if it doesn't exist
    mkfifo(fifo, 0666);
    // Message to send
    const char *message = "Hello from writer process!";
    // Open FIFO for writing
    int fd = open(fifo, O_WRONLY);
    if (fd == -1) {
        perror("open writer");
        return 1;
    }
    // Write message to FIFO
    write(fd, message, strlen(message));
    // Close FIFO
    close(fd);
    return 0;
}
```

## **OUTPUT :**

### **Writer terminal:**

Hello from writer process!

### **Reader terminal:**

25Reader got: Hello from writer process!

## **PROGRAM OBJECTIVE :**

### **7. Implementation of MESSAGE QUEUE**

## **PROGRAM CODE :**

### **Receiver.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_TEXT 512
#define QUEUE_KEY 1234
// Message structure
struct msg_buffer {
    long msg_type;
    char msg_text[MAX_TEXT];
};
int main() {
    int msgid;
    // Create or get the message queue
    msgid = msgget(QUEUE_KEY, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    struct msg_buffer message;
    printf("Waiting for a message...\n");
    // Receive message of type 1
    if (msgrcv(msgid, &message, sizeof(message.msg_text), 1, 0) == -1) {
        perror("msgrcv");
        exit(EXIT_FAILURE);
    }
    printf("Received message: %s\n", message.msg_text);
    // Delete the queue after receiving the message
    if (msgctl(msgid, IPC_RMID, NULL) == -1) {
        perror("msgctl (IPC_RMID)");
        exit(EXIT_FAILURE);
    }
    printf("Message queue deleted.\n");
    return 0;
}
```

### **Sender.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_TEXT 512
#define QUEUE_KEY 1234
// Define the message buffer structure (same as receiver)
struct msg_buffer {
    long msg_type;
    char msg_text[MAX_TEXT];
};
int main() {
    int msgid;
    // Get the message queue (must exist)
    msgid = msgget(QUEUE_KEY, 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    struct msg_buffer message;
    message.msg_type = 1; // Required by System V
    printf("Enter message to send: ");
    if (fgets(message.msg_text, MAX_TEXT, stdin) == NULL) {
```

```
perror("fgets");
    exit(EXIT_FAILURE);
}
// Send the message (excluding msg_type size)
if (msgsnd(msgid, &message, strlen(message.msg_text) + 1, 0) == -1) {
    perror("msgsnd");
    exit(EXIT_FAILURE);
}
printf("Message sent: %s\n", message.msg_text);
return 0;
}
```

### **OUTPUT :**

Receiver terminal:

Waiting for a message...

Received message: Hello from sender!

Message queue deleted.

Sender terminal:

Enter message to send: Hello from sender!

Message sent: Hello from sender!

## **PROGRAM OBJECTIVE :**

### **8. Implementation of SHARED MEMORY**

## **PROGRAM CODE :**

### **Shared memory without semaphore:**

#### **Writer.c**

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <string.h>
typedef struct {
    char data[100];
} shared_mem;
int main() {
    key_t key = ftok("/home/ubuntu/Desktop/semaphore", 65);
    printf("The value of key is %d\n",key);
    int shmid = shmget(key, sizeof(shared_mem), 0666 | IPC_CREAT);
    printf("The value of shmid is %d\n",shmid);
    shared_mem *shm_ptr = (shared_mem*) shmatt(shmid, NULL, 0);
    printf("The value of shared memory pointer is %p\n", (void*)shm_ptr);
    // Keep writing messages continuously
    for (int i = 1; i <= 200; i++) {
        sprintf(shm_ptr->data, "Message-%d", i);
        printf("[Writer] Wrote: %s\n", shm_ptr->data);
        usleep(100000); // small delay (0.1 sec) so reader can interfere
    }
    shmdt(shm_ptr);
    return 0;
}
```

#### **Reader.c**

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
typedef struct {
    char data[100];
} shared_mem;
int main() {
    key_t key = ftok("/home/ubuntu/Desktop/semaphore", 65);
    printf("The value of key is %d\n",key);
    int shmid = shmget(key, sizeof(shared_mem), 0666);
    printf("The value of shmid is %d\n",shmid);
    shared_mem *shm_ptr = (shared_mem*) shmatt(shmid, NULL, 0);
    printf("The value of shared memory pointer is %p\n", (void*)shm_ptr);
    // Keep reading messages continuously
    for (int i = 1; i <= 200; i++) {
        printf("[Reader] Read : %s\n", shm_ptr->data);
        usleep(150000); // 0.15delay so we catch writer mid-update sometimes
    }
    shmdt(shm_ptr);
    shmctl(shmid, IPC_RMID, NULL); // remove shared memory
    return 0;
}
```

### **Shared memory with semaphore:**

#### **writer.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <string.h>
#include <unistd.h>
#define SHM_KEY 0x1234
```

```

#define SEM_KEY 0x5678
typedef struct {
    int number;
} shared_mem;
void sem_wait(int semid, int sem_num) {
    struct sembuf sb = {sem_num, -1, 0};
    semop(semid, &sb, 1);
}
void sem_signal(int semid, int sem_num) {
    struct sembuf sb = {sem_num, 1, 0};
    semop(semid, &sb, 1);
}
int main() {
    int shmid = shmget(SHM_KEY, sizeof(shared_mem), 0666 | IPC_CREAT);
    shared_mem *shm_ptr = (shared_mem *)shmat(shmid, NULL, 0);
    int semid = semget(SEM_KEY, 2, 0666 | IPC_CREAT);
    // Initialize semaphores (only writer should do this once)
    semctl(semid, 0, SETVAL, 1); // writer lock
    semctl(semid, 1, SETVAL, 0); // reader lock
    for (int i = 1; i <= 10; i++) {
        sem_wait(semid, 0); // wait for writer lock
        shm_ptr->number = i;
        printf("Writer wrote: %d\n", i);
        fflush(stdout);
        sem_signal(semid, 1); // signal reader
        sleep(1); // slow down for demo
    }
    shmdt(shm_ptr);
    return 0;
}

```

#### reader.c

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#define SHM_KEY 0x1234
#define SEM_KEY 0x5678
typedef struct {
    int value;
} shared_mem;
void sem_wait(int semid, int sem_num) {
    struct sembuf sb = {sem_num, -1, 0};
    if (semop(semid, &sb, 1) == -1) {
        perror("semop wait failed");
        exit(1);
    }
}
void sem_signal(int semid, int sem_num) {
    struct sembuf sb = {sem_num, 1, 0};
    if (semop(semid, &sb, 1) == -1) {
        perror("semop signal failed");
        exit(1);
    }
}
int main() {
    int shmid = shmget(SHM_KEY, sizeof(shared_mem), 0666 | IPC_CREAT);
    if (shmid == -1) { perror("shmget"); exit(1); }
    shared_mem *shm_ptr = (shared_mem *)shmat(shmid, NULL, 0);
    if (shm_ptr == (void *)-1) { perror("shmat"); exit(1); }
    int semid = semget(SEM_KEY, 2, 0666 | IPC_CREAT);
    if (semid == -1) { perror("semget"); exit(1); }
    for (int i = 1; i <= 10; i++) {
        sem_wait(semid, 1); // wait for reader turn
        printf("Reader received: %d\n", shm_ptr->value);
        sem_signal(semid, 0); // give turn back to writer
    }
}

```

```
}  
shmdt(shm_ptr);  
return 0;  
}
```

### **OUTPUT :**

Without semaphore:

Writer:

The value of key is 1097752025

The value of shmid is 65536

The value of shared memory pointer is 0x7f8b1e5a7000

[Writer] Wrote: Message-1

[Writer] Wrote: Message-2

[Writer] Wrote: Message-3

...

[Writer] Wrote: Message-200

Reader:

The value of key is 1097752025

The value of shmid is 65536

The value of shared memory pointer is 0x7f8b1e5a7000

[Reader] Read : Message-1

[Reader] Read : Message-2

[Reader] Read : Message-3

...

[Reader] Read : Message-200

With semaphore

Writer:

Writer wrote: 1

Writer wrote: 2

Writer wrote: 3

...

Writer wrote: 10

Reader:

Reader received: 1

Reader received: 2

Reader received: 3

...

Reader received: 10

## **PROGRAM OBJECTIVE :**

### 9. Implementation of FIRST COME FIRST SERVED SCHEDULING ALGO

## **PROGRAM CODE :**

```
#include <stdio.h>
int main() {
    int n, i, j, temp;
    int pid[20], at[20], bt[20], ct[20], tat[20], wt[20];
    float avgTAT = 0, avgWT = 0;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("P%d Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("P%d Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
    }
    // Sort by Arrival Time
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (at[i] > at[j]) {
                // swap arrival time
                temp = at[i]; at[i] = at[j]; at[j] = temp;
                // swap burst time
                temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;
                // swap process id
                temp = pid[i]; pid[i] = pid[j]; pid[j] = temp;
            }
        }
    }
    // Calculate Completion Time (CT)
    ct[0] = at[0] + bt[0];
    for (i = 1; i < n; i++) {
        if (at[i] > ct[i - 1]) {
            ct[i] = at[i] + bt[i]; // CPU idle
        } else {
            ct[i] = ct[i - 1] + bt[i];
        }
    }
    // Calculate TAT and WT
    for (i = 0; i < n; i++) {
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
        avgTAT += tat[i];
        avgWT += wt[i];
    }
    avgTAT /= n;
    avgWT /= n;
    //Display Table
    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], at[i], bt[i], ct[i], tat[i], wt[i]);
    }
    printf("\nAverage Turnaround Time = %.2f", avgTAT);
    printf("\nAverage Waiting Time = %.2f\n", avgWT);
    return 0;
}
```

## **OUTPUT :**

```
Enter number of processes: 3
P1 Arrival Time: 0
P1 Burst Time: 5
P2 Arrival Time: 0
```

P2 Burst Time: 3  
P3 Arrival Time: 0  
P3 Burst Time: 8

Process	AT	BT	CT	TAT	WT
P1	0	5	5	5	0
P2	0	3	8	8	5
P3	0	8	16	16	8

Average Turnaround Time = 9.67  
Average Waiting Time = 4.33

Enter number of processes: 3

P1 Arrival Time: 2  
P1 Burst Time: 5  
P2 Arrival Time: 0  
P2 Burst Time: 3  
P3 Arrival Time: 4  
P3 Burst Time: 4

Process	AT	BT	CT	TAT	WT
P2	0	3	3	3	0
P1	2	5	8	6	1
P3	4	4	12	8	4

Average Turnaround Time = 5.67  
Average Waiting Time = 1.67

## **PROGRAM OBJECTIVE :**

### **10. Implementation of SHORTEST JOB FIRST SCHEDULING ALGO (Non Preemptive SRJF)**

## **PROGRAM CODE :**

```
#include <stdio.h>
#define MAX 20
//SJF (Non Preemptive SRJF)
void sjfNonPreemptive() {
    int n, i, j;
    int pid[MAX], at[MAX], bt[MAX], ct[MAX], tat[MAX], wt[MAX], completed[MAX];
    int time = 0, count = 0, min_bt, idx;
    float avgTAT = 0, avgWT = 0;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("P%d Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("P%d Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
        completed[i] = 0;
    }
    while (count < n) {
        min_bt = 9999;
        idx = -1;
        // find process with smallest burst time among arrived
        for (i = 0; i < n; i++) {
            if (at[i] <= time && completed[i] == 0) {
                if (bt[i] < min_bt) {
                    min_bt = bt[i];
                    idx = i;
                } else if (bt[i] == min_bt) {
                    if (at[i] < at[idx]) idx = i;
                }
            }
        }
        if (idx != -1) {
            time += bt[idx];
            ct[idx] = time;
            tat[idx] = ct[idx] - at[idx];
            wt[idx] = tat[idx] - bt[idx];
            avgTAT += tat[idx];
            avgWT += wt[idx];
            completed[idx] = 1;
            count++;
        } else {
            time++; // CPU idle
        }
    }
    printf("\n--- Non-Preemptive SJF (SJFNP) ---\n");
    printf("Process\tAT\tBT\tCT\tTAT\tWT\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], at[i], bt[i], ct[i], tat[i], wt[i]);
    }
    printf("\nAverage Turnaround Time = %.2f", avgTAT / n);
    printf("\nAverage Waiting Time = %.2f\n", avgWT / n);
}

int main() {
    int choice;
    sjfNonPreemptive();
}
```

## **OUTPUT :**

Enter number of processes: 4

P1 Arrival Time: 1

P1 Burst Time: 3

P2 Arrival Time: 2

P2 Burst Time: 4

P3 Arrival Time: 1

P3 Burst Time: 2

P4 Arrival Time: 4

P4 Burst Time: 4

--- Non-Preemptive SJF (SJFNP) ---

Process	AT	BT	CT	TAT	WT
P1	1	3	6	5	2
P2	2	4	10	8	4
P3	1	2	3	2	0
P4	4	4	14	10	6

Average Turnaround Time = 6.25

Average Waiting Time = 3.00

## **PROGRAM OBJECTIVE :**

### **11. Implementation of PRIORITY SCHEDULING ALGO (Non Preemptive)**

## **PROGRAM CODE :**

**//NON-PREEMPTIVE PRIORITY SCHEDULING**

```
void priorityNonPreemptive() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int bt[n], at[n], priority[n], ct[n], tat[n], wt[n], pid[n];
    int completed[n];
    int total_tat = 0, total_wt = 0;
    int total = 0, currentTime = 0, minPriority, index = -1;
    // Input process details
    for (int i = 0; i < n; i++) {
        pid[i] = i + 1;
        completed[i] = 0;
        printf("Enter Burst Time for P%d: ", pid[i]);
        scanf("%d", &bt[i]);
        printf("Enter Arrival Time for P%d: ", pid[i]);
        scanf("%d", &at[i]);
        printf("Enter Priority for P%d (Lower number = higher priority): ", pid[i]);
        scanf("%d", &priority[i]);
    }
    while (total < n) {
        minPriority = 9999;
        index = -1;
        // Find process with highest priority among arrived & not completed
        for (int i = 0; i < n; i++) {
            if (at[i] <= currentTime && completed[i] == 0) {
                if (priority[i] < minPriority ||
                    (priority[i] == minPriority && at[i] < at[index]) ||
                    (priority[i] == minPriority && at[i] == at[index] && pid[i] < pid[index])) {
                    minPriority = priority[i];
                    index = i;
                }
            }
        }
        // If no process has arrived yet
        if (index == -1) {
            currentTime++;
            continue;
        }
        // Compute CT first
        ct[index] = currentTime + bt[index];
        tat[index] = ct[index] - at[index];
        total_tat += tat[index];
        wt[index] = tat[index] - bt[index];
        total_wt += wt[index];
        currentTime = ct[index];
        completed[index] = 1;
        total++;
    }
    // Display Process Table
    printf("\n--- Non-Preemptive Priority Scheduling ---\n");
    printf("\nPID\tBT\tAT\tPriority\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
            pid[i], bt[i], at[i], priority[i], ct[i], tat[i], wt[i]);
    }
    printf("\nAverage Turnaround Time = %.2f", (float)total_tat / n);
    printf("\nAverage Waiting Time = %.2f\n", (float)total_wt / n);
}

int main() {
```

```
    priorityNonPreemptive();  
    return 0;  
}
```

### **OUTPUT :**

Enter number of processes: 3

Enter Burst Time for P1: 4

Enter Arrival Time for P1: 0

Enter Priority for P1 (Lower number = higher priority): 2

Enter Burst Time for P2: 2

Enter Arrival Time for P2: 1

Enter Priority for P2 (Lower number = higher priority): 1

Enter Burst Time for P3: 6

Enter Arrival Time for P3: 2

Enter Priority for P3 (Lower number = higher priority): 3

--- Non-Preemptive Priority Scheduling ---

PID	BT	AT	Priority	CT	TAT	WT
P1	4	0	2	4	4	0
P2	2	1	1	6	5	3
P3	6	2	3	12	10	4

Average Turnaround Time = 6.33

Average Waiting Time = 2.33

### **PROGRAM OBJECTIVE :**

12. Implementation of First comes first serve page replacement policy

### **PROGRAM CODE :**

```
#include <stdio.h>
int main() {
    int i, j, n, f, count = 0, index = 0, found;
    printf("Enter number of pages: ");
    scanf("%d", &n);
    int pages[n];
    printf("Enter the page reference string:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &pages[i]);
    printf("Enter number of frames: ");
    scanf("%d", &f);
    int frames[f];
    for(i = 0; i < f; i++)
        frames[i] = -1; // initialize as empty
    printf("\nPage Reference\tFrames\n");
    for(i = 0; i < n; i++) {
        found = 0;
        // Check if page already in frame
        for(j = 0; j < f; j++) {
            if(frames[j] == pages[i]) {
                found = 1;
                break;
            }
        }
        // Replace if not found (FCFS order)
        if(!found) {
            frames[index] = pages[i];
            index = (index + 1) % f;
            count++;
        }
        // Display frame contents
        printf("%d\t\t", pages[i]);
        for(j = 0; j < f; j++) {
            if(frames[j] != -1)
                printf("%d ", frames[j]);
            else
                printf("- ");
        }
        printf("\n");
    }
    printf("\nTotal Page Faults = %d\n", count);
    printf("Page Fault Rate = %.2f%%\n", (float)count / n * 100);
    return 0;
}
```

### **OUTPUT :**

Enter number of pages: 7

Enter the page reference string:

0 2 1 6 4 0 1

Enter number of frames: 4

Page Reference Frames

0      0 ---

2      0 2 --

1      0 2 1 -

6      0 2 1 6

4      4 2 1 6

0      4 0 1 6

1      4 0 1 6

Total Page Faults = 6

Page Fault Rate = 85.71%

## **PROGRAM OBJECTIVE :**

### 13. Implementation of Least recent used page replacement policy

## **PROGRAM CODE :**

```
#include <stdio.h>
// Function to find the Least Recently Used frame
int findLRU(int time[], int n) {
    int i, min = time[0], pos = 0;
    for(i = 1; i < n; i++) {
        if(time[i] < min) {
            min = time[i];
            pos = i;
        }
    }
    return pos;
}

int main() {
    int no_of_frames, no_of_pages;
    int frames[10], pages[30];
    int time[10]; // Last-used timestamps
    int counter = 0, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);
    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);
    printf("Enter the page reference string:\n");
    for(int i = 0; i < no_of_pages; i++)
        scanf("%d", &pages[i]);
    // Initialize frames to empty
    for(int i = 0; i < no_of_frames; i++)
        frames[i] = -1;
    printf("\nStep\tRef\tFrames\n");
    for(int i = 0; i < no_of_pages; i++) {
        counter++;
        int page = pages[i];
        int found = 0;
        // Check if page already in frame (hit)
        for(int j = 0; j < no_of_frames; j++) {
            if(frames[j] == page) {
                time[j] = counter; // Update last-used time
                found = 1;
                break;
            }
        }
        // Page fault handling
        if(!found) {
            int placed = 0;
            // Check for empty frame first
            for(int j = 0; j < no_of_frames; j++) {
                if(frames[j] == -1) {
                    frames[j] = page;
                    time[j] = counter;
                    faults++;
                    placed = 1;
                    break;
                }
            }
            // No empty frame → replace LRU page
            if(!placed) {
                int lru_index = findLRU(time, no_of_frames);
                frames[lru_index] = page;
                time[lru_index] = counter;
                faults++;
            }
        }
    }
}
```

```

    }
    // Display current step and frame contents
    printf("%d\t%d\t", i+1, page);
    for(int j = 0; j < no_of_frames; j++) {
        if(frames[j] != -1)
            printf("%d ", frames[j]);
        else
            printf("- ");
    }
    printf("\n");
}
printf("\nTotal Page Faults = %d\n", faults);
printf("Page Fault Rate = %.2f%%\n", (float)faults / no_of_pages * 100);
return 0;
}

```

### **OUTPUT :**

Enter number of frames: 4

Enter number of pages: 13

Enter the page reference string:

7 0 1 2 0 3 0 4 2 3 0 3 2

Step	Ref	Frames
1	7	7 - - -
2	0	7 0 - -
3	1	7 0 1 -
4	2	7 0 1 2
5	0	7 0 1 2
6	3	3 0 1 2
7	0	3 0 1 2
8	4	3 0 4 2
9	2	3 0 4 2
10	3	3 0 4 2
11	0	3 0 4 2
12	3	3 0 4 2
13	2	3 0 4 2

Total Page Faults = 6

Page Fault Rate = 46.15%

### **PROGRAM OBJECTIVE :**

14. Demonstration of `execl()` where child process executes "ls" COMMAND and Parent process executes "date" COMMAND

### **PROGRAM CODE :**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    pid_t pid;
    pid = fork();    // Create a child process

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    }
    else if (pid == 0) {
        // Child process
        printf("\n[Child Process] Executing 'ls' command:\n");
        execl("/bin/ls", "ls", NULL);
        perror("execl failed");
    }
    else {
        // Parent process
        wait(NULL); // Wait for child to complete
        printf("\n[Parent Process] Executing 'date' command:\n");
        execl("/bin/date", "date", NULL);
        perror("execl failed");
    }
    return 0;
}
```

### **OUTPUT :**

[Child Process] Executing 'ls' command:

Os\_8.c 3 6 a.out main-debug os\_2.c os\_5.c os.c prac3\_1.c prac.c program.c

1 4 7 main Makefile os\_3.c os\_6.c prac1.c prac3.c pro1.c replit.nix

2 5 8 main.c myfifo os\_4.c os\_7.c prac2.c prac4.c pro2.c

[Parent Process] Executing 'date' command:

Wed Nov 12 05:51:44 PM UTC 2025

### **PROGRAM OBJECTIVE :**

15. Implementation of COMMAND `ls|wc` USING PIPES.

### **PROGRAM CODE :**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    int fd[2]; // fd[0] -> read end, fd[1] -> write end
    // Create pipe
    if (pipe(fd) == -1) {
        perror("pipe failed");
        exit(1);
    }
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        // ----- Child process -----
        // Redirect stdout to pipe write end
        dup2(fd[1], STDOUT_FILENO);
        // Close unused ends
        close(fd[0]);
        close(fd[1]);
        // Execute "ls"
        execlp("ls", "ls", NULL);
        perror("execlp failed");
        exit(1);
    }
    else {
        // ----- Parent process -----
        // Redirect stdin to pipe read end
        dup2(fd[0], STDIN_FILENO);
        // Close unused ends
        close(fd[0]);
        close(fd[1]);
        // Execute "wc"
        execlp("wc", "wc", NULL);
        perror("execlp failed");
        exit(1);
    }

    return 0;
}
```

### **OUTPUT :**

32 32 199

## **PROGRAM OBJECTIVE :**

### 16. Wait and Status

## **PROGRAM CODE :**

### **wait system call using status**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    int status;
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        return 1;
    }
    if (pid == 0) {
        // Child process
        printf("Child: PID = %d, Parent PID = %d\n", getpid(), getppid());
        exit(25); // Exit with status 5
    } else {
        // Parent process
        printf("Parent: PID = %d, Child PID = %d\n", getpid(), pid);
        wait(&status);
        printf("I am after wait hi");
        if (WIFEXITED(status)) {
            int exit_code = WEXITSTATUS(status);
            printf("\nChild exited with code: %d\n", exit_code);
            return exit_code; // This will be captured by `echo $?` in shell
        } else {
            printf("Child did not exit normally.\n");
            return 1;
        }
    }
}
```

## **OUTPUT :**

### **Wait system call using status:**

Parent: PID = 528, Child PID = 529

Child: PID = 529, Parent PID = 528

I am after wait hi

Child exited with code: 25

## **PROGRAM OBJECTIVE :**

### **17. SRJF (Preemptive SJF)**

## **PROGRAM CODE :**

```
//Preemptive SJF (SRJF)
void sjfPreemptive() {
    int n, i, time = 0, min, idx;
    int at[MAX], bt[MAX], rt[MAX], ct[MAX], tat[MAX], wt[MAX], completed[MAX] = {0};
    float avg_tat = 0, avg_wt = 0;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter Arrival time of P%d: ", i + 1);
        scanf("%d", &at[i]);
        printf("Enter Burst time of P%d: ", i + 1);
        scanf("%d", &bt[i]);
        rt[i] = bt[i]; // initialize remaining time
    }
    int completed_count = 0;
    while (completed_count < n) {
        min = 9999;
        idx = -1;
        // find process with minimum remaining time
        for (i = 0; i < n; i++) {
            if (at[i] <= time && completed[i] == 0) {
                if (rt[i] < min) {
                    min = rt[i];
                    idx = i;
                } else if (rt[i] == min) {
                    if (at[i] < at[idx]) idx = i;
                }
            }
        }
        if (idx != -1) {
            rt[idx]--;
            time++;
            if (rt[idx] == 0) {
                completed[idx] = 1;
                completed_count++;
                ct[idx] = time;
                tat[idx] = ct[idx] - at[idx];
                wt[idx] = tat[idx] - bt[idx];
            }
        } else {
            time++;
        }
    }
    printf("\n--- Preemptive SJF (SRJF) ---\n");
    printf("Process\tAT\tBT\tCT\tTAT\tWT\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);
        avg_tat += tat[i];
        avg_wt += wt[i];
    }
    printf("\nAverage Turnaround Time = %.2f", avg_tat / n);
    printf("\nAverage Waiting Time = %.2f\n", avg_wt / n);
}

int main() {
    int choice;
    sjfPreemptive();
}
```

## **OUTPUT :**

Enter number of processes: 4

Enter Arrival time of P1: 0

Enter Burst time of P1: 5

Enter Arrival time of P2: 1

Enter Burst time of P2: 3

Enter Arrival time of P3: 2

Enter Burst time of P3: 4

Enter Arrival time of P4: 4

Enter Burst time of P4: 1

--- Preemptive SJF (SRJF) ---

Process	AT	BT	CT	TAT	WT
P1	0	5	9	9	4
P2	1	3	4	3	0
P3	2	4	13	11	7
P4	4	1	5	1	0

Average Turnaround Time = 6.00

Average Waiting Time = 2.75

**PROGRAM OBJECTIVE :**

18. SRJF Non Preemptive

**PROGRAM CODE :**

Same as SJF (program 10)

## **PROGRAM OBJECTIVE :**

19. Shared Memory 2 Variation Data Consistency and fixed version

## **PROGRAM CODE :**

### **Same as program 8**

#### **Shared memory without semaphore: (Inconsistent data)**

##### **Writer.c**

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <string.h>
typedef struct {
    char data[100];
} shared_mem;
int main() {
    key_t key = ftok("/home/ubuntu/Desktop/semaphore", 65);
    printf("The value of key is %d\n",key);
    int shmid = shmget(key, sizeof(shared_mem), 0666 | IPC_CREAT);
    printf("The value of shmid is %d\n",shmid);
    shared_mem *shm_ptr = (shared_mem*) shmat(shmid, NULL, 0);
    printf("The value of shared memory pointer is %p\n", (void*)shm_ptr);
    // Keep writing messages continuously
    for (int i = 1; i <= 200; i++) {
        sprintf(shm_ptr->data, "Message-%d", i);
        printf("[Writer] Wrote: %s\n", shm_ptr->data);
        usleep(100000); // small delay (0.1 sec) so reader can interfere
    }
    shmdt(shm_ptr);
    return 0;
}
```

##### **Reader.c**

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
typedef struct {
    char data[100];
} shared_mem;
int main() {
    key_t key = ftok("/home/ubuntu/Desktop/semaphore", 65);
    printf("The value of key is %d\n",key);
    int shmid = shmget(key, sizeof(shared_mem), 0666);
    printf("The value of shmid is %d\n",shmid);
    shared_mem *shm_ptr = (shared_mem*) shmat(shmid, NULL, 0);
    printf("The value of shared memory pointer is %p\n", (void*)shm_ptr);
    // Keep reading messages continuously
    for (int i = 1; i <= 200; i++) {
        printf("[Reader] Read : %s\n", shm_ptr->data);
        usleep(150000); // 0.15delay so we catch writer mid-update sometimes
    }
    shmdt(shm_ptr);
    shmctl(shmid, IPC_RMID, NULL); // remove shared memory
    return 0;
}
```

#### **Shared memory with semaphore: (Fixed data consistency)**

##### **writer.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <string.h>
```

```

#include <unistd.h>
#define SHM_KEY 0x1234
#define SEM_KEY 0x5678
typedef struct {
    int number;
} shared_mem;
void sem_wait(int semid, int sem_num) {
    struct sembuf sb = {sem_num, -1, 0};
    semop(semid, &sb, 1);
}
void sem_signal(int semid, int sem_num) {
    struct sembuf sb = {sem_num, 1, 0};
    semop(semid, &sb, 1);
}
int main() {
    int shmid = shmget(SHM_KEY, sizeof(shared_mem), 0666 | IPC_CREAT);
    shared_mem *shm_ptr = (shared_mem *)shmat(shmid, NULL, 0);
    int semid = semget(SEM_KEY, 2, 0666 | IPC_CREAT);
    // Initialize semaphores (only writer should do this once)
    semctl(semid, 0, SETVAL, 1); // writer lock
    semctl(semid, 1, SETVAL, 0); // reader lock
    for (int i = 1; i <= 10; i++) {
        sem_wait(semid, 0); // wait for writer lock
        shm_ptr->number = i;
        printf("Writer wrote: %d\n", i);
        fflush(stdout);
        sem_signal(semid, 1); // signal reader
        sleep(1); // slow down for demo
    }
    shmdt(shm_ptr);
    return 0;
}

```

#### reader.c

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#define SHM_KEY 0x1234
#define SEM_KEY 0x5678
typedef struct {
    int value;
} shared_mem;
void sem_wait(int semid, int sem_num) {
    struct sembuf sb = {sem_num, -1, 0};
    if (semop(semid, &sb, 1) == -1) {
        perror("semop wait failed");
        exit(1);
    }
}
void sem_signal(int semid, int sem_num) {
    struct sembuf sb = {sem_num, 1, 0};
    if (semop(semid, &sb, 1) == -1) {
        perror("semop signal failed");
        exit(1);
    }
}
int main() {
    int shmid = shmget(SHM_KEY, sizeof(shared_mem), 0666 | IPC_CREAT);
    if (shmid == -1) { perror("shmget"); exit(1); }
    shared_mem *shm_ptr = (shared_mem *)shmat(shmid, NULL, 0);
    if (shm_ptr == (void *)-1) { perror("shmat"); exit(1); }
    int semid = semget(SEM_KEY, 2, 0666 | IPC_CREAT);
    if (semid == -1) { perror("semget"); exit(1); }
    for (int i = 1; i <= 10; i++) {
        sem_wait(semid, 1); // wait for reader turn
    }
}

```

```

        printf("Reader received: %d\n", shm_ptr->value);
        sem_signal(semid, 0); // give turn back to writer
    }
    shmdt(shm_ptr);
    return 0;
}

```

### **OUTPUT :**

Without semaphore:

Writer:

The value of key is 1097752025

The value of shmid is 65536

The value of shared memory pointer is 0x7f8b1e5a7000

[Writer] Wrote: Message-1

[Writer] Wrote: Message-2

[Writer] Wrote: Message-3

...

[Writer] Wrote: Message-200

Reader:

The value of key is 1097752025

The value of shmid is 65536

The value of shared memory pointer is 0x7f8b1e5a7000

[Reader] Read : Message-1

[Reader] Read : Message-2

[Reader] Read : Message-3

...

[Reader] Read : Message-200

With semaphore

Writer:

Writer wrote: 1

Writer wrote: 2

Writer wrote: 3

...

Writer wrote: 10

Reader:

Reader received: 1

Reader received: 2

Reader received: 3

...

Reader received: 10

## PROGRAM OBJECTIVE :

### 20. Priority scheduling (Preemptive)

## PROGRAM CODE :

```
#include <stdio.h>
#define MAX 20
//PREEMPTIVE PRIORITY SCHEDULING
int findHighestPriority(int n, int at[], int pr[], int rem_bt[], int completed[], int time) {
    int idx = -1;
    int min_pr = 9999;
    for (int i = 0; i < n; i++) {
        if (at[i] <= time && completed[i] == 0 && rem_bt[i] > 0) {
            if (pr[i] < min_pr) {
                min_pr = pr[i];
                idx = i;
            }
        }
    }
    return idx; // returns -1 if no process is ready
}

void priorityPreemptive(int n, int at[], int bt[], int pr[]) {
    int ct[MAX], tat[MAX], wt[MAX], rem_bt[MAX], completed[MAX];
    int total_tat = 0, total_wt = 0;
    int completed_count = 0, time = 0;
    // Initialize
    for (int i = 0; i < n; i++) {
        rem_bt[i] = bt[i];
        completed[i] = 0;
    }
    // Scheduling loop
    while (completed_count != n) {
        int idx = findHighestPriority(n, at, pr, rem_bt, completed, time);
        if (idx != -1) {
            rem_bt[idx]--;
            time++;
            if (rem_bt[idx] == 0) {
                completed[idx] = 1;
                completed_count++;
                ct[idx] = time;
            }
        } else {
            time++; // if no process has arrived yet
        }
    }
    // Calculate and display results
    printf("\n--- Preemptive Priority Scheduling ---\n");
    printf("\nProcess\tAT\tBT\tPriority\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
        total_tat += tat[i];
        total_wt += wt[i];
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
            i + 1, at[i], bt[i], pr[i], ct[i], tat[i], wt[i]);
    }
    printf("\nAverage Turnaround Time = %.2f", (float)total_tat / n);
    printf("\nAverage Waiting Time = %.2f\n", (float)total_wt / n);
}

int main() {
    int n, at[MAX], bt[MAX], pr[MAX];
    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
```

```

        printf("\nEnter Arrival Time, Burst Time, and Priority for Process %d (Lower number
= higher priority): ", i + 1);
        scanf("%d%d%d", &at[i], &bt[i], &pr[i]);
    }
    priorityPreemptive(n, at, bt, pr);
    return 0;
}

```

### **OUTPUT :**

Enter number of processes: 5

Enter Arrival Time, Burst Time, and Priority for Process 1 (Lower number = higher priority): 0 3 3

Enter Arrival Time, Burst Time, and Priority for Process 2 (Lower number = higher priority): 1 4 2

Enter Arrival Time, Burst Time, and Priority for Process 3 (Lower number = higher priority): 2 6 4

Enter Arrival Time, Burst Time, and Priority for Process 4 (Lower number = higher priority): 3 4 6

Enter Arrival Time, Burst Time, and Priority for Process 5 (Lower number = higher priority): 5 2 10

--- Preemptive Priority Scheduling ---

Process	AT	BT	Priority	CT	TAT	WT
P1	0	3	3	7	7	4
P2	1	4	2	5	4	0
P3	2	6	4	13	11	5
P4	3	4	6	17	14	10
P5	5	2	10	19	14	12

Average Turnaround Time = 10.00

Average Waiting Time = 6.20