1.System Call – Display the running process ID

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

int main() {
   pid_t process_id = getpid();
   printf("Running Process ID: %d\n", process_id);
   return 0;
}
```

2. System Call – Copy the file content from one to another

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

int main(int argc, char *argv[]) {
   if (argc != 3) {
      fprintf(stderr, "Usage: %s <source_file> <destination_file>\n", argv[0]);
      return 1;
   }

   int source_fd = open(argv[1], O_RDONLY);
   if (source_fd < 0) {
      perror("Error opening source file");
      return 1;
   }

   int dest_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
   if (dest_fd < 0) {
      perror("Error opening destination file");
      close(source_fd);
      return 1;
   }

   char buffer[1024];
   ssize_t bytes_read, bytes_written;

   while ((bytes_read = read(source_fd, buffer, sizeof(buffer))) > 0) {
      bytes_written = write(dest_fd, buffer, bytes_read);
      if (bytes_written < 0) {
         perror("Error writing to destination file");
         close(source_fd);
         close(dest_fd);
         return 1;
      }
   }

   if (bytes_read < 0) {
      perror("Error reading from source file");
```

```
    }

    close(source_fd);
    close(dest_fd);
    return 0;
}
```

3.

CPU Scheduling Algorithm – First Come First Serve

```
#include <stdio.h>

struct Process {
    int id;
    int burst_time;
    int waiting_time;
    int turnaround_time;
};

void findWaitingTime(struct Process proc[], int n) {
    proc[0].waiting_time = 0;
    for (int i = 1; i < n; i++) {
        proc[i].waiting_time = proc[i - 1].waiting_time + proc[i - 1].burst_time;
    }
}

void findTurnaroundTime(struct Process proc[], int n) {
    for (int i = 0; i < n; i++) {
        proc[i].turnaround_time = proc[i].burst_time + proc[i].waiting_time;
    }
}

void findavgTime(struct Process proc[], int n) {
    findWaitingTime(proc, n);
    findTurnaroundTime(proc, n);

    float total_waiting_time = 0, total_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        total_waiting_time += proc[i].waiting_time;
        total_turnaround_time += proc[i].turnaround_time;
    }

    printf("Average waiting time: %.2f\n", total_waiting_time / n);
    printf("Average turnaround time: %.2f\n", total_turnaround_time / n);
}

int main() {
    int n;
```

```
    printf("Enter number of processes: ");
    scanf("%d", &n);

    struct Process proc[n];
    for (int i = 0; i < n; i++) {
        proc[i].id = i + 1;
        printf("Enter burst time for process %d: ", proc[i].id);
        scanf("%d", &proc[i].burst_time);
    }

    findavgTime(proc, n);
    return 0;
}
```

4.CPU Scheduling Algorithm – Shortest Seek Time First

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

void sort(int arr[], int n) {
    int temp;
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    int n, head, seek_time = 0;
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);
    int requests[n];

    printf("Enter the disk requests: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }

    printf("Enter the initial head position: ");
    scanf("%d", &head);

    sort(requests, n);

    for (int i = 0; i < n; i++) {
        seek_time += abs(requests[i] - head);
        head = requests[i];
```

```
    }

    printf("Total Seek Time: %d\n", seek_time);
    return 0;
}
```

5. CPU Scheduling Algorithm – Longest Seek Time First

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

void LSTF(int requests[], int n, int head) {
    int completed[n];
    int seek_sequence[n];
    int seek_count = 0;
    int distance, max_distance, index;

    for (int i = 0; i < n; i++) {
        completed[i] = 0;
    }

    for (int i = 0; i < n; i++) {
        max_distance = -1;
        index = -1;

        for (int j = 0; j < n; j++) {
            if (!completed[j]) {
                distance = abs(requests[j] - head);
                if (distance > max_distance) {
                    max_distance = distance;
                    index = j;
                }
            }
        }

        completed[index] = 1;
        seek_sequence[seek_count++] = requests[index];
        head = requests[index];
    }

    printf("Seek Sequence: ");
    for (int i = 0; i < seek_count; i++) {
        printf("%d ", seek_sequence[i]);
    }
    printf("\n");
}

int main() {
    int requests[] = {100, 180, 50, 30, 200};
    int head = 100;
    int n = sizeof(requests) / sizeof(requests[0]);
```

```c
    LSTF(requests, n, head);
    return 0;
}
```

6. Pre-emptive priority scheduling algorithm

```c
#include <stdio.h>

struct Process {
    int id;
    int burst_time;
    int priority;
};

void sortProcesses(struct Process proc[], int n) {
    struct Process temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (proc[j].priority > proc[j + 1].priority) {
                temp = proc[j];
                proc[j] = proc[j + 1];
                proc[j + 1] = temp;
            }
        }
    }
}

void preemptivePriorityScheduling(struct Process proc[], int n) {
    sortProcesses(proc, n);
    int waiting_time[n], turnaround_time[n];
    waiting_time[0] = 0;

    for (int i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + proc[i - 1].burst_time;
    }

    for (int i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + proc[i].burst_time;
    }

    printf("Process ID\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", proc[i].id, proc[i].burst_time,
proc[i].priority, waiting_time[i], turnaround_time[i]);
    }
}

int main() {
    int n;
    printf("Enter the number of processes: ");
```

```c
    scanf("%d", &n);
    struct Process proc[n];

    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time and Priority for Process %d: ", i + 1);
        scanf("%d %d", &proc[i].burst_time, &proc[i].priority);
        proc[i].id = i + 1;
    }

    preemptivePriorityScheduling(proc, n);
    return 0;
}
```

7. Non-pre-emptive algorithm – Shortest Job First

```c
#include <stdio.h>

struct Process {
    int id;
    int burst_time;
};

void findWaitingTime(struct Process proc[], int n, int waiting_time[]) {
    waiting_time[0] = 0;

    for (int i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + proc[i - 1].burst_time;
    }
}

void findTurnAroundTime(struct Process proc[], int n, int waiting_time[], int
turn_around_time[]) {
    for (int i = 0; i < n; i++) {
        turn_around_time[i] = proc[i].burst_time + waiting_time[i];
    }
}

void findavgTime(struct Process proc[], int n) {
    int waiting_time[n], turn_around_time[n];

    findWaitingTime(proc, n, waiting_time);
    findTurnAroundTime(proc, n, waiting_time, turn_around_time);

    float total_waiting_time = 0, total_turn_around_time = 0;

    for (int i = 0; i < n; i++) {
        total_waiting_time += waiting_time[i];
        total_turn_around_time += turn_around_time[i];
    }

    printf("Average waiting time: %.2f\n", total_waiting_time / n);
```

```
    printf("Average turn around time: %.2f\n", total_turn_around_time / n);
}

void sortProcesses(struct Process proc[], int n) {
    struct Process temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (proc[j].burst_time > proc[j + 1].burst_time) {
                temp = proc[j];
                proc[j] = proc[j + 1];
                proc[j + 1] = temp;
            }
        }
    }
}

int main() {
    struct Process proc[] = { {1, 6}, {2, 8}, {3, 7}, {4, 3} };
    int n = sizeof(proc) / sizeof(proc[0]);

    sortProcesses(proc, n);
    findavgTime(proc, n);

    return 0;
}
```

8. Round Robin scheduling algorithm.

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {
    int rem_bt[n];
    for (int i = 0; i < n; i++)
        rem_bt[i] = bt[i];

    int t = 0;
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (rem_bt[i] > 0) {
                done = 0;
                if (rem_bt[i] > quantum) {
                    t += quantum;
                    rem_bt[i] -= quantum;
                } else {
                    t = t + rem_bt[i];
                    wt[i] = t - bt[i];
                    rem_bt[i] = 0;
                }
            }
        }
```

```
        if (done == 1)
            break;
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++)
        tat[i] = bt[i] + wt[i];
}

void findavgTime(int processes[], int n, int bt[], int quantum) {
    int wt[n], tat[n];
    findWaitingTime(processes, n, bt, wt, quantum);
    findTurnAroundTime(processes, n, bt, wt, tat);

    float total_wt = 0, total_tat = 0;
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
    }
    printf("Average waiting time: %.2f\n", total_wt / n);
    printf("Average turnaround time: %.2f\n", total_tat / n);
}

int main() {
    int processes[] = { 0, 1, 2, 3 };
    int n = sizeof(processes) / sizeof(processes[0]);
    int burst_time[] = { 10, 5, 8, 12 };
    int quantum = 4;

    findavgTime(processes, n, burst_time, quantum);
    return 0;
}
```

9. **Inter-process communication using shared memory**

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

#define SHM_SIZE 1024

int main() {
    int shmid;
    key_t key = 1234;
    char *str;

    shmid = shmget(key, SHM_SIZE, 0666|IPC_CREAT);
```

```
    str = (char*) shmat(shmid, (void*)0, 0);

    if (fork() == 0) {
        // Child process
        printf("Enter a string: ");
        fgets(str, SHM_SIZE, stdin);
        shmdt(str);
    } else {
        // Parent process
        wait(NULL);
        printf("You wrote: %s\n", str);
        shmdt(str);
        shmctl(shmid, IPC_RMID, NULL);
    }

    return 0;
}
```

10.Inter-process communication using message queue.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define MAX_TEXT 512

struct message {
    long msg_type;
    char text[MAX_TEXT];
};

int main() {
    key_t key;
    int msgid;
    struct message msg;

    key = ftok("progfile", 65);
    msgid = msgget(key, 0666 | IPC_CREAT);

    msg.msg_type = 1;
    strcpy(msg.text, "Hello, this is a message!");

    msgsnd(msgid, &msg, sizeof(msg), 0);
    printf("Message sent: %s\n", msg.text);

    msgrcv(msgid, &msg, sizeof(msg), 1, 0);
    printf("Message received: %s\n", msg.text);
```

```c
    msgctl(msgid, IPC_RMID, NULL);
    return 0;
}
```

21.Memory management - worst fit algorithm.

```c
#include <stdio.h>

#define MAX 100

int blockSize[MAX], processSize[MAX], allocation[MAX];

void worstFit(int m, int n) {
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

    for (int i = 0; i < n; i++) {
        int worstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {
                    worstIdx = j;
                }
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }
}

void printAllocation(int n) {
    printf("Process No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%d\n", allocation[i] + 1);
        } else {
            printf("Not Allocated\n");
        }
    }
}

int main() {
    int m, n;

    printf("Enter number of blocks: ");
```

```
    scanf("%d", &m);
    printf("Enter number of processes: ");
    scanf("%d", &n);

    printf("Enter size of blocks:\n");
    for (int i = 0; i < m; i++) {
        scanf("%d", &blockSize[i]);
    }

    printf("Enter size of processes:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processSize[i]);
    }

    worstFit(m, n);
    printAllocation(n);

    return 0;
}
```

22. **Memory management - Best fit algorithm**

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

#define MAX 100

struct Block {
    int size;
    int isFree;
};

struct Block memory[MAX];

void initializeMemory(int totalSize) {
    memory[0].size = totalSize;
    memory[0].isFree = 1;
    for (int i = 1; i < MAX; i++) {
        memory[i].size = 0;
        memory[i].isFree = 0;
    }
}

int bestFit(int size) {
    int bestIndex = -1;
    for (int i = 0; i < MAX; i++) {
        if (memory[i].isFree && memory[i].size >= size) {
            if (bestIndex == -1 || memory[i].size < memory[bestIndex].size) {
                bestIndex = i;
            }
        }
```

```
    }
    return bestIndex;
}

void allocateMemory(int size) {
    int index = bestFit(size);
    if (index != -1) {
        memory[index].isFree = 0;
        if (memory[index].size > size) {
            memory[index + 1].size = memory[index].size - size;
            memory[index + 1].isFree = 1;
            memory[index].size = size;
        }
        printf("Allocated %d bytes at block %d\n", size, index);
    } else {
        printf("No suitable block found for allocation of %d bytes\n", size);
    }
}

void freeMemory(int index) {
    if (index >= 0 && index < MAX && !memory[index].isFree) {
        memory[index].isFree = 1;
        printf("Freed memory at block %d\n", index);
    } else {
        printf("Invalid block index or already free\n");
    }
}

int main() {
    initializeMemory(500);
    allocateMemory(200);
    allocateMemory(100);
    freeMemory(0);
    allocateMemory(50);
    return 0;
}
```

Memory management FF

```
#include <stdio.h>

#define MAX 100

void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
```

```
            allocation[i] = j;
            blockSize[j] -= processSize[i];
            break;
          }
       }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
       printf("%d\t\t%d\t\t", i + 1, processSize[i]);
       if (allocation[i] != -1) {
          printf("%d\n", allocation[i] + 1);
       } else {
          printf("Not Allocated\n");
       }
    }
}

int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);
    return 0;
}
```

37.Disk Scheduling Algorithm - First Come First Served.

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
    wt[0] = 0;

    for (int i = 1; i < n; i++)
        wt[i] = bt[i - 1] + wt[i - 1];
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++)
        tat[i] = bt[i] + wt[i];
}

void findavgTime(int processes[], int n, int bt[]) {
    int wt[n], tat[n];

    findWaitingTime(processes, n, bt, wt);
    findTurnAroundTime(processes, n, bt, wt, tat);

    float total_wt = 0, total_tat = 0;
```

```
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("Average waiting time: %.2f\n", total_wt / n);
    printf("Average turnaround time: %.2f\n", total_tat / n);
}

int main() {
    int processes[] = { 1, 2, 3, 4 };
    int n = sizeof(processes) / sizeof(processes[0]);
    int burst_time[] = { 10, 5, 8, 12 };

    findavgTime(processes, n, burst_time);
    return 0;
}
```

Disk SCAN

```
#include <stdio.h>

#define MAX 200
#define SIZE 10

void SCAN(int arr[], int head, int direction, int size) {
    int seek_sequence[MAX], distance, seek_count = 0, cur_track;
    int i, j;

    // Sort the request array
    for (i = 0; i < size; i++) {
        for (j = i + 1; j < size; j++) {
            if (arr[i] > arr[j]) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }

    // Find the index of the head
    int index = 0;
    for (i = 0; i < size; i++) {
        if (arr[i] >= head) {
            index = i;
            break;
        }
    }

    // Service the requests going towards the left
```

```c
    if (direction == 0) {
        for (i = index - 1; i >= 0; i--) {
            cur_track = arr[i];
            seek_sequence[seek_count++] = cur_track;
            distance = cur_track - head;
            seek_count += distance < 0 ? -distance : distance;
            head = cur_track;
        }

        // Now service the requests going towards the right
        for (i = index; i < size; i++) {
            cur_track = arr[i];
            seek_sequence[seek_count++] = cur_track;
            distance = cur_track - head;
            seek_count += distance < 0 ? -distance : distance;
            head = cur_track;
        }
    }
    // Service the requests going towards the right
    else {
        for (i = index; i < size; i++) {
            cur_track = arr[i];
            seek_sequence[seek_count++] = cur_track;
            distance = cur_track - head;
            seek_count += distance < 0 ? -distance : distance;
            head = cur_track;
        }

        // Now service the requests going towards the left
        for (i = index - 1; i >= 0; i--) {
            cur_track = arr[i];
            seek_sequence[seek_count++] = cur_track;
            distance = cur_track - head;
            seek_count += distance < 0 ? -distance : distance;
            head = cur_track;
        }
    }

    printf("Seek Sequence is: ");
    for (i = 0; i < seek_count; i++) {
        printf("%d ", seek_sequence[i]);
    }
    printf("\nTotal Seek Count: %d\n", seek_count);
}

int main() {
    int arr[SIZE] = { 100, 180, 30, 90, 40, 150, 60, 200, 70, 110 };
    int head = 100;
    int direction = 0; // 0 for left, 1 for right
```

```
    SCAN(arr, head, direction, SIZE);
    return 0;
}
```

Dining-Philosophers Algorithm.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_PHILOSOPHERS 5

sem_t forks[NUM_PHILOSOPHERS];

void* philosopher(void* num) {
    int id = *(int*)num;
    int left_fork = id;
    int right_fork = (id + 1) % NUM_PHILOSOPHERS;

    while (1) {
        printf("Philosopher %d is thinking.\n", id);
        sleep(1);

        sem_wait(&forks[left_fork]);
        sem_wait(&forks[right_fork]);

        printf("Philosopher %d is eating.\n", id);
        sleep(1);

        sem_post(&forks[left_fork]);
        sem_post(&forks[right_fork]);
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&forks[i], 0, 1);
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_ids[i]);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
```

```
        sem_destroy(&forks[i]);
    }

    return 0;
}
```

Thread concepts - (i)create (ii) join (iii) equal (iv) exit

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

void* threadFunction(void* arg) {
    printf("Thread %d is running.\n", *(int*)arg);
    sleep(1);
    return NULL;
}

int main() {
    pthread_t threads[5];
    int threadIds[5];

    // Create threads
    for (int i = 0; i < 5; i++) {
        threadIds[i] = i;
        if (pthread_create(&threads[i], NULL, threadFunction, &threadIds[i]) != 0) {
            perror("Failed to create thread");
            exit(EXIT_FAILURE);
        }
    }

    // Join threads
    for (int i = 0; i < 5; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("Failed to join thread");
            exit(EXIT_FAILURE);
        }
    }

    // Check if threads are equal
    if (pthread_equal(threads[0], threads[1])) {
        printf("Thread 0 and Thread 1 are equal.\n");
    } else {
        printf("Thread 0 and Thread 1 are not equal.\n");
    }

    // Exit
    pthread_exit(NULL);
    return 0;
```

}

PAGE TECHNiQue FIFO

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

#define FRAME_SIZE 4
#define PAGE_SIZE 4
#define TOTAL_PAGES 10

int frames[FRAME_SIZE];
int pageFaults = 0;
int front = 0;

void initializeFrames() {
    for (int i = 0; i < FRAME_SIZE; i++) {
        frames[i] = -1;
    }
}

int isPageInFrames(int page) {
    for (int i = 0; i < FRAME_SIZE; i++) {
        if (frames[i] == page) {
            return 1;
        }
    }
    return 0;
}

void addPageToFrames(int page) {
    frames[front] = page;
    front = (front + 1) % FRAME_SIZE;
}

void fifoPaging(int pages[], int n) {
    initializeFrames();
    for (int i = 0; i < n; i++) {
        if (!isPageInFrames(pages[i])) {
            addPageToFrames(pages[i]);
            pageFaults++;
        }
    }
}

int main() {
    int pages[TOTAL_PAGES] = {0, 1, 2, 3, 0, 4, 0, 5, 1, 2};
    fifoPaging(pages, TOTAL_PAGES);
    printf("Total Page Faults: %d\n", pageFaults);
    return 0;
}
```

MULTI THREADING

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void* print_message(void* arg) {
   char* message = (char*)arg;
   printf("%s\n", message);
   pthread_exit(NULL);
}

int main() {
   pthread_t thread1, thread2;  // Thread identifiers
   int result1, result2;
v// Messages for threads
   char* message1 = "Hello from Thread 1!";
   char* message2 = "Hello from Thread 2!";

   // Create threads
   result1 = pthread_create(&thread1, NULL, print_message, (void*)message1);
   result2 = pthread_create(&thread2, NULL, print_message, (void*)message2);
   // Wait for threads to complete
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   printf("Threads completed successfully.\n");
   return 0;
}
```