#include <stdio.h>

#include <unistd.h>

int main() {

    int num = 100;

*// Create a child process*

    pid\_t pid = fork();

    if (pid < 0) {

*// Error occurred while forking*

        fprintf(stderr, "Fork failed\n");

        return 1;

    } else if (pid == 0) {

*// This is the child process*

        printf("Value in child process: %d\n", num);

    } else {

*// This is the parent process*

        printf("Value in parent process: %d\n", num);

    }

    return 0;

}

**2) Write another program using**fork()**.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, char \*argv[]) {

 int id = fork();

  if (id == 0) {

    printf("Hello from the child\n");

  }

  else if (id > 0) {

    sleep(1);

    printf("Goodbye from the parent\n");

  }

  else {

    printf("fork did not work as intended\n");

    exit(1);

  }

  return 0;

}

Assignment 2

**FCFS Scheduling**

#include <stdio.h>

int main() {

    int n; // number of processes

    int duration[100], waiting\_time[100], turnaround\_time[100];

    float total\_waiting\_time = 0, total\_turnaround\_time = 0;

    // Input number of processes and their durations

    printf("Enter the number of processes: ");

    scanf("%d", &n);

    for(int i = 0; i < n; i++) {

        printf("Enter duration for process %d: ", i+1);

        scanf("%d", &duration[i]);

    }

    // Initialize waiting time for the first process

    waiting\_time[0] = 0;

    // Calculate waiting time and turnaround time for each process

    for(int i = 1; i < n; i++) {

        waiting\_time[i] = duration[i-1] + waiting\_time[i-1];

        turnaround\_time[i] = duration[i] + waiting\_time[i];

    }

    // Calculate total waiting time and total turnaround time

    for(int i = 0; i < n; i++) {

        total\_waiting\_time += waiting\_time[i];

        total\_turnaround\_time += turnaround\_time[i];

    }

    // Display the results

    printf("Process\tDuration\tWaiting Time\tTurnaround Time\n");

    for(int i = 0; i < n; i++) {

        printf("%d\t%d\t\t%d\t\t%d\n", i+1, duration[i], waiting\_time[i], turnaround\_time[i]);

    }

    printf("Average Waiting Time: %.2f\n", total\_waiting\_time/n);

    printf("Average Turnaround Time: %.2f\n", total\_turnaround\_time/n);

    return 0;

}

**SJF Scheduling**

#include <stdio.h>

int main() {

    int n; *// number of processes*

    int execution\_time[100], waiting\_time[100], turnaround\_time[100], process\_id[100];

    float total\_waiting\_time = 0, total\_turnaround\_time = 0;

*// Gather input from the user*

    printf("Enter the number of processes: ");

    scanf("%d", &n);

    for(int i = 0; i < n; i++) {

        printf("Enter execution time for process %d: ", i+1);

        scanf("%d", &execution\_time[i]);

        process\_id[i] = i+1;

    }

*// Bubble sort to arrange processes based on their execution time*

    for(int i = 0; i < n-1; i++) {

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

            if(execution\_time[j] > execution\_time[j+1]) {

*// Swap execution times*

                int temp = execution\_time[j];

                execution\_time[j] = execution\_time[j+1];

                execution\_time[j+1] = temp;

*// Swap process IDs to keep them in sync with execution times*

                int temp\_id = process\_id[j];

                process\_id[j] = process\_id[j+1];

                process\_id[j+1] = temp\_id;

            }

        }

    }

*// Calculate waiting and turnaround times*

    waiting\_time[0] = 0;

    turnaround\_time[0] = execution\_time[0];

    for(int i = 1; i < n; i++) {

        waiting\_time[i] = execution\_time[i-1] + waiting\_time[i-1];

        turnaround\_time[i] = execution\_time[i] + waiting\_time[i];

    }

*// Calculate total waiting and turnaround times*

    for(int i = 0; i < n; i++) {

        total\_waiting\_time += waiting\_time[i];

        total\_turnaround\_time += turnaround\_time[i];

    }

*// Display the results*

    printf("Process\tExecution Time\tWaiting Time\tTurnaround Time\n");

    for(int i = 0; i < n; i++) {

        printf("%d\t%d\t\t%d\t\t%d\n", process\_id[i], execution\_time[i], waiting\_time[i], turnaround\_time[i]);

    }

    printf("\nAverage Waiting Time: %.2f\n", total\_waiting\_time/n);

    printf("Average Turnaround Time: %.2f\n", total\_turnaround\_time/n);

    return 0;

}

**Multi-Level Feedback**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

    int id;

    int duration;

    int remaining\_time;

    int waiting\_time;

    int turnaround\_time;

} Process;

typedef struct {

    Process \*processes[100];

    int front, rear;

    int time\_quantum;

} Queue;

void initializeQueue(Queue \*q, int time\_quantum) {

    q->front = q->rear = -1;

    q->time\_quantum = time\_quantum;

}

void enqueue(Queue \*q, Process \*p) {

    if(q->rear == 99) {

        printf("Queue is full!\n");

        return;

    }

    q->processes[++q->rear] = p;

    if(q->front == -1) {

        q->front = 0;

    }

    printf("Process %d enqueued in queue with time quantum %d\n", p->id, q->time\_quantum);

}

Process\* dequeue(Queue \*q) {

    if(q->front == -1) {

        return NULL;

    }

    Process \*p = q->processes[q->front];

    if(q->front == q->rear) {

        q->front = q->rear = -1;

    } else {

        q->front++;

    }

    printf("Process %d dequeued from queue with time quantum %d\n", p->id, q->time\_quantum);

    return p;

}

void mlfq\_scheduling(Queue \*high\_priority\_q, Queue \*medium\_priority\_q, Queue \*low\_priority\_q, int n) {

    int total\_time = 0;

    while(1) {

        Process \*p = dequeue(high\_priority\_q);

        if(p != NULL) {

            printf("Process %d is running in high priority queue\n", p->id);

            if(p->remaining\_time <= high\_priority\_q->time\_quantum) {

                total\_time += p->remaining\_time;

                p->remaining\_time = 0;

                p->waiting\_time = total\_time - p->duration;

                p->turnaround\_time = total\_time;

                printf("Process %d finished execution\n", p->id);

            } else {

                p->remaining\_time -= high\_priority\_q->time\_quantum;

                total\_time += high\_priority\_q->time\_quantum;

                enqueue(medium\_priority\_q, p);

            }

        } else {

            p = dequeue(medium\_priority\_q);

            if(p != NULL) {

                printf("Process %d is running in medium priority queue\n", p->id);

                if(p->remaining\_time <= medium\_priority\_q->time\_quantum) {

                    total\_time += p->remaining\_time;

                    p->remaining\_time = 0;

                    p->waiting\_time = total\_time - p->duration;

                    p->turnaround\_time = total\_time;

                    printf("Process %d finished execution\n", p->id);

                } else {

                    p->remaining\_time -= medium\_priority\_q->time\_quantum;

                    total\_time += medium\_priority\_q->time\_quantum;

                    enqueue(low\_priority\_q, p);

                }

            } else {

                p = dequeue(low\_priority\_q);

                if(p != NULL) {

                    printf("Process %d is running in low priority queue\n", p->id);

                    if(p->remaining\_time <= low\_priority\_q->time\_quantum) {

                        total\_time += p->remaining\_time;

                        p->remaining\_time = 0;

                        p->waiting\_time = total\_time - p->duration;

                        p->turnaround\_time = total\_time;

                        printf("Process %d finished execution\n", p->id);

                    } else {

                        p->remaining\_time -= low\_priority\_q->time\_quantum;

                        total\_time += low\_priority\_q->time\_quantum;

                        enqueue(low\_priority\_q, p);

                    }

                } else {

                    break;

                }

            }

        }

    }

}

int main() {

    Queue high\_priority\_q, medium\_priority\_q, low\_priority\_q;

    initializeQueue(&high\_priority\_q, 2);

    initializeQueue(&medium\_priority\_q, 4);

    initializeQueue(&low\_priority\_q, 8);

    int n;

    printf("Enter the number of processes: ");

    scanf("%d", &n);

    Process \*processes = (Process \*)malloc(n \* sizeof(Process));

    for(int i = 0; i < n; i++) {

        printf("Enter duration for process %d: ", i+1);

        scanf("%d", &processes[i].duration);

        processes[i].id = i+1;

        processes[i].remaining\_time = processes[i].duration;

        processes[i].waiting\_time = 0;

        processes[i].turnaround\_time = 0;

        enqueue(&high\_priority\_q, &processes[i]);

    }

    mlfq\_scheduling(&high\_priority\_q, &medium\_priority\_q, &low\_priority\_q, n);

    printf("Process\tDuration\tWaiting Time\tTurnaround Time\n");

    for(int i = 0; i < n; i++) {

        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].duration, processes[i].waiting\_time, processes[i].turnaround\_time);

    }

    free(processes);

    return 0;

}

Assignment 3  
  
#include <stdio.h>

#include <stdlib.h>

int main(void) {

*// integer variables*

  int N = 0;

  int sum = 0;

  int i;

*// integer pointer variables*

  int \*iptr, \*tmp;

*// take user input*

  printf("Enter value of N [1-10]: ");

  scanf("%d", &N);

*// allocate memory*

  iptr = (int \*) malloc(N \* sizeof(int));

*// check if memory allocated*

  if (iptr == NULL) {

    printf("Unable to allocate memory space. Program terminated.\n");

    return -1;

  }

*// take integers*

  printf("Enter %d integer number(s)\n", N);

  for (i = 0, tmp = iptr; i < N; i++, tmp++) {

    printf("Enter #%d: ", (i+1));

    scanf("%d", tmp);

*// compute the sum*

    sum += \*tmp;

  }

*// display result*

  printf("Sum: %d\n", sum);

*// free memory location*

  free(iptr);

  return 0;

}