Multilevel queue

```
#include <stdio.h>

// Function to find waiting time for FCFS
void findWaitingTime(int processes[], int n, int bt[], int at[], int wt[]) {
    wt[0] = 0;
    for (int i = 1; i < n; i++) {
        wt[i] = bt[i-1] + wt[i-1] - at[i-1];
        if (wt[i] < 0)
            wt[i] = 0;
    }
}

// Function to find turnaround time
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];
    }
}

// Function to implement Round Robin scheduling
void roundRobin(int processes[], int n, int bt[], int at[], int quantum) {
    int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;
    int remaining_bt[n];
    int completed = 0;
    int time = 0;

    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
    }
```

```
    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0 && at[i] <= time) {
                if (remaining_bt[i] <= quantum) {
                    time += remaining_bt[i];
                    remaining_bt[i] = 0;
                    ct[i] = time;
                    completed++;
                } else {
                    time += quantum;
                    remaining_bt[i] -= quantum;
                }
            }
        }
    }

    findWaitingTime(processes, n, bt, at, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);

    printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], at[i], wt[i], tat[i], ct[i]);
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("Average Waiting Time (Round Robin) = %f\n", (float)total_wt / n);
    printf("Average Turnaround Time (Round Robin) = %f\n", (float)total_tat / n);
}

// Function to implement FCFS scheduling
```

```
void fcfs(int processes[], int n, int bt[], int at[]) {
    int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;

    findWaitingTime(processes, n, bt, at, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);

    printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
    for (int i = 0; i < n; i++) {
        ct[i] = at[i] + bt[i];
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], at[i], wt[i], tat[i], ct[i]);
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("Average Waiting Time (FCFS) = %f\n", (float)total_wt / n);
    printf("Average Turnaround Time (FCFS) = %f\n", (float)total_tat / n);
}

int main() {
    int processes[] = {1, 2, 3, 4, 5};
    int n = sizeof(processes) / sizeof(processes[0]);
    int bt[] = {10, 5, 8, 12, 15};
    int at[] = {0, 1, 2, 3, 4};
    int quantum = 2;

    roundRobin(processes, n, bt, at, quantum);
    fcfs(processes, n, bt, at);

    return 0;
}
```

```
Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time
P1              10              0              0               10              39
P2              5               1              10              15              23
P3              8               2              14              22              33
P4              12              3              20              32              45
P5              15              4              29              44              50
Average Waiting Time (Round Robin) = 14.600000
Average Turnaround Time (Round Robin) = 24.600000
Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time
P1              10              0              0               10              10
P2              5               1              10              15              6
P3              8               2              14              22              10
P4              12              3              20              32              15
P5              15              4              29              44              19
Average Waiting Time (FCFS) = 14.600000
Average Turnaround Time (FCFS) = 24.600000

Process returned 0 (0x0)   execution time : 0.059 s
Press any key to continue.
```

Rate Monotonic Scheduling

```
#include <stdio.h>


// Structure to represent a process
struct Process {
    int execution_time;
    int time_period;
};


// Function to calculate the least common multiple (LCM)
int lcm(int a, int b) {
    int max = (a > b) ? a : b;
    while (1) {
        if (max % a == 0 && max % b == 0)
            return max;
        max++;
    }
}


// Function to check if the set of processes is schedulable
int is_schedulable(struct Process processes[], int n) {
```

```
    float utilization = 0.0;
    for (int i = 0; i < n; i++) {
        utilization += (float)processes[i].execution_time / processes[i].time_period;
    }
    return utilization <= 1.0;
}

int main() {
    struct Process processes[] = {
        {3, 20}, // P1
        {2, 5},  // P2
        {2, 10}  // P3
    };
    int n = sizeof(processes) / sizeof(processes[0]);

    // Check if the processes are schedulable
    if (!is_schedulable(processes, n)) {
        printf("The given set of processes is not schedulable.\n");
        return 0;
    }

    // Calculate the scheduling time (LCM of time periods)
    int scheduling_time = lcm(processes[0].time_period, processes[1].time_period);
    scheduling_time = lcm(scheduling_time, processes[2].time_period);

    // Display the execution order
    printf("Execution order:\n");
    for (int t = 0; t < scheduling_time; t++) {
        if (t % processes[1].time_period == 0)
            printf("P2 ");
        if (t % processes[2].time_period == 0)
```

```
            printf("P3 ");
        if (t % processes[0].time_period == 0)
            printf("P1 ");
    }
    printf("\n");

    return 0;
}
```

```
Execution order:
P2 P3 P1 P2 P2 P3 P2

Process returned 0 (0x0)   execution time : 0.058 s
Press any key to continue.
```

```
#include <stdio.h>

#define MAX_PROCESS 100

struct process {
  int pid;
  int period; // task period
  int deadline;
  int execution_time; // time required for one execution
};

// Function to swap two processes
void swap(struct process* a, struct process* b) {
  struct process temp = *a;
  *a = *b;
  *b = temp;
}
```

```
// Function to sort processes based on period (for Rate Monotonic)
void sort_by_period(struct process proc[], int n) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
      if (proc[j].period > proc[j + 1].period) {
        swap(&proc[j], &proc[j + 1]);
      }
    }
  }
}

// Function to sort processes based on deadline (for Earliest Deadline First)
void sort_by_deadline(struct process proc[], int n) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
      if (proc[j].deadline > proc[j + 1].deadline) {
        swap(&proc[j], &proc[j + 1]);
      }
    }
  }
}

// Function to simulate scheduling (replace with specific algorithm logic)
void schedule(struct process proc[], int n) {
  printf("Scheduling logic specific to the chosen algorithm needs to be implemented here.\n");
}

void print_table_header() {
  printf("Process | Period | Deadline | Execution Time\n");
  printf("------- | -------- | -------- | --------\n");
}
```

```
void print_process_info(struct process proc) {
  printf(" %d \t | %d \t\t | %d \t\t | %d \n", proc.pid, proc.period, proc.deadline,
proc.execution_time);
}

int main() {
  int n, i;
  struct process proc[MAX_PROCESS];

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

  printf("Enter details of processes:\n");
  for (i = 0; i < n; i++) {
    printf("Process ID: ");
    scanf("%d", &proc[i].pid);
    printf("Period: ");
    scanf("%d", &proc[i].period);
    printf("Deadline: ");
    scanf("%d", &proc[i].deadline);
    printf("Execution Time: ");
    scanf("%d", &proc[i].execution_time);
  }

  printf("\nScheduling Results:\n");

  // Rate Monotonic Scheduling
  printf("\n** Rate Monotonic Scheduling**\n");
  print_table_header();
  for (i = 0; i < n; i++) {
```

```
    print_process_info(proc[i]);
  }
  sort_by_period(proc, n);
  schedule(proc, n);

  // Earliest Deadline First Scheduling
  printf("\n\n** Earliest Deadline First Scheduling**\n");
  print_table_header();
  for (i = 0; i < n; i++) {
    print_process_info(proc[i]);
  }
  sort_by_deadline(proc, n);
  schedule(proc, n);

  // Proportional Scheduling (Implementation needed)
  printf("\n\n** Proportional Scheduling**\n");
  print_table_header();
  for (i = 0; i < n; i++) {
    print_process_info(proc[i]);
  }
  printf("  (Implementation required for Proportional Scheduling)\n");

  return 0;
}
```

```
Enter the number of processes: 3
Enter details of processes:
Process ID: 1
Period: 20
Deadline: 7
Execution Time: 3
Process ID: 2
Period: 5
Deadline: 4
Execution Time: 2
Process ID: 3
Period: 10
Deadline: 8
Execution Time: 2

Scheduling Results:

** Rate Monotonic Scheduling**
Process | Period | Deadline | Execution Time
------- | -------- | -------- | --------
 1       | 20            | 7            | 3
 2       | 5             | 4            | 2
 3       | 10            | 8            | 2
Scheduling logic specific to the chosen algorithm needs to be implemented here.


** Earliest Deadline First Scheduling**
Process | Period | Deadline | Execution Time
------- | -------- | -------- | --------
 2       | 5             | 4            | 2
 3       | 10            | 8            | 2
 1       | 20            | 7            | 3
Scheduling logic specific to the chosen algorithm needs to be implemented here.


** Proportional Scheduling**
Process | Period | Deadline | Execution Time
------- | -------- | -------- | --------
 2       | 5             | 4            | 2
 1       | 20            | 7            | 3
 3       | 10            | 8            | 2
  (Implementation required for Proportional Scheduling)

Process returned 0 (0x0)   execution time : 20.828 s
Press any key to continue.
```