Operating Systems

Homework Report

Muhammad Youshay (07103)

Assignment 02 – Simulate a scheduler

October 11, 2023

Table of Contents

1 Code Snippets –

This is the code that was provided to us. I have used these helping functions multiple times in my code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include<stdbool.h>
//process control block (PCB)
struct pcb
{
   unsigned int pid;
   char pname[20];
   unsigned int ptimeleft;
   unsigned int ptimearrival;
    bool pfirsttime;
};

typedef struct pcb pcb;

//queue node
struct dlq_node
{
   struct dlq_node *pfwd;
   struct dlq_node *pbck;
   struct pcb *data;
};

typedef struct dlq_node dlq_node;

//queue
struct dlq
{
   struct dlq_node *head;
   struct dlq_node *tail;
};

typedef struct dlq dlq;

//function to add a pcb to a new queue node
dlq_node * get_new_node (pcb *ndata)
{
   if (!ndata)
      return NULL;

   dlq_node *new = malloc(sizeof(dlq_node));
   if(!new)
```

```
    {
      fprintf(stderr, "Error: allocating memory\n");exit(1);
   }

   new->pfwd = new->pbck = NULL;
   new->data = ndata;
   return new;
}

//function to add a node to the tail of queue
void add_to_tail (dlq *q, dlq_node *new)
{
   if (!new)
      return;

   if (q->head==NULL)
    {
      if(q->tail!=NULL)
        {
         fprintf(stderr, "DLList inconsitent.\n"); exit(1);
      }
      q->head = new;
      q->tail = q->head;
   }
   else
    {
      new->pfwd = q->tail;
      new->pbck = NULL;
      new->pfwd->pbck = new;
      q->tail = new;
   }
}

//function to remove a node from the head of queue
dlq_node* remove_from_head(dlq * const q){
   if (q->head==NULL){ //empty
      if(q->tail!=NULL){fprintf(stderr, "DLList inconsitent.\n"); exit(1);}
      return NULL;
   }
   else if (q->head == q->tail) { //one element
      if (q->head->pbck!=NULL || q->tail->pfwd!=NULL) {
         fprintf(stderr, "DLList inconsitent.\n"); exit(1);
      }

      dlq_node *p = q->head;
      q->head = NULL;
      q->tail = NULL;

      p->pfwd = p->pbck = NULL;
      return p;
   }
```

```
   else { // normal
      dlq_node *p = q->head;
      q->head = q->head->pbck;
      q->head->pfwd = NULL;

      p->pfwd = p->pbck = NULL;
      return p;
   }
}

//function to print our queue
void print_q (const dlq *q)
{
   dlq_node *n = q->head;
   if (n == NULL)
      return;

   while (n)
    {
      printf("%s(%d),", n->data->pname, n->data->ptimeleft);
      n = n->pbck;
   }
}

//function to check if the queue is empty
int is_empty (const dlq *q)
{
   if (q->head == NULL && q->tail==NULL)
      return 1;
   else if (q->head != NULL && q->tail != NULL)
      return 0;
   else
    {
      fprintf(stderr, "Error: DLL queue is inconsistent."); exit(1);
   }
}

//function to sort the queue on completion time
void sort_by_timetocompletion(const dlq *q)
{
   // bubble sort
   dlq_node *start = q->tail;
   dlq_node *end = q->head;

   while (start != end)
    {
      dlq_node *node = start;
      dlq_node *next = node->pfwd;

      while (next != NULL)
        {
```

```
        if (node->data->ptimeleft < next->data->ptimeleft)
           {
           // do a swap
           pcb *temp = node->data;
           node->data = next->data;
           next->data = temp;
        }
        node = next;
        next = node->pfwd;
      }
      end = end ->pbck;
   }
}

//function to sort the queue on arrival time
void sort_by_arrival_time (const dlq *q)
{
   // bubble sort
   dlq_node *start = q->tail;
   dlq_node *end = q->head;

   while (start != end)
    {
      dlq_node *node = start;
      dlq_node *next = node->pfwd;

      while (next != NULL)
        {
         if (node->data->ptimearrival < next->data->ptimearrival)
           {
           // do a swap
           pcb *temp = node->data;
           node->data = next->data;
           next->data = temp;
        }
        node = next;
        next = node->pfwd;
      }
      end = end->pbck;
   }
}

//function to tokenize the one row of data
pcb* tokenize_pdata (char *buf)
{
   pcb* p = (pcb*) malloc(sizeof(pcb));
   if(!p)
    {
        fprintf(stderr, "Error: allocating memory.\n"); exit(1);
    }
```

```
    char *token = strtok(buf, ":\n");
    if(!token)
     {
         fprintf(stderr, "Error: Expecting token pname\n"); exit(1);
     }
    strcpy(p->pname, token);

    token = strtok(NULL, ":\n");
    if(!token)
     {
         fprintf(stderr, "Error: Expecting token pid\n"); exit(1);
     }
    p->pid = atoi(token);

    token = strtok(NULL, ":\n");
    if(!token)
     {
         fprintf(stderr, "Error: Expecting token duration\n"); exit(1);
     }

    p->ptimeleft= atoi(token);

    token = strtok(NULL, ":\n");
    if(!token)
     {
         fprintf(stderr, "Error: Expecting token arrival time\n"); exit(1);
     }
    p->ptimearrival = atoi(token);
     p->pfirsttime = true;

    token = strtok(NULL, ":\n");
    if(token)
     {
         fprintf(stderr, "Error: Oh, what've you got at the end of the
             line?\n");exit(1);
     }

    return p;
}
int main()
{
    /* Enter your code here. Read input from STDIN. Print output to STDOUT */
    int N = 0;
    char tech[20]={'\0'};
    char buffer[100]={'\0'};
    scanf("%d", &N);
    scanf("%s", tech);

    dlq queue;
    queue.head = NULL;
    queue.tail = NULL;
```

```
    for(int i=0; i<N; ++i)
    {
        scanf("%s\n", buffer);
        pcb *p = tokenize_pdata(buffer);
        add_to_tail (&queue, get_new_node(p) );
    }
    //print_q(&queue);
    unsigned int system_time = 0;
    sort_by_arrival_time (&queue);
    //print_q (&queue);

    // run scheduler
    if(!strncmp(tech,"FIFO",4))
        sched_FIFO(&queue, &system_time);
    else if(!strncmp(tech,"SJF",3))
        sched_SJF(&queue, &system_time);
    else if(!strncmp(tech,"STCF",4))
        sched_STCF(&queue, &system_time);
    else if(!strncmp(tech,"RR",2))
        sched_RR(&queue, &system_time);
    else
        fprintf(stderr, "Error: unknown POLICY\n");
    return 0;
}
```

This is the code that I wrote to implement the First In First Out scheduler. Comments have been added for explanation. This code is different from that implemented on Hackerank as it includes the additional code to calculate metrics.

```
void sched_FIFO(dlq *const p_fq, int *p_time)
{
    int NoOfProcesses = 0; //To keep a check for the no of processes for our
        metrics
    double Throughput = 0; //Throughput initialization
    double AverageTurnaroundTime = 0; //Average Turnaround Time initialization
    double TotalTurnaroundTime = 0; //Total Turnaround Time initialization
    double AverageResponseTime = 0; //Average Response Time initialization
    double TotalResponseTime = 0; //Total Response Time initialization
    struct pcb *Current = NULL; //Current process
    dlq ReadyQueue;             //Queue where the prcess arrives and waits for
        its turn
    ReadyQueue.head = NULL;
    ReadyQueue.tail = NULL;
    dlq_node *head = p_fq->head; //Head of the queue to keep track of moving
        from one process to another
    int EndTime = 0;            // Used this for calculating time for the whole
        schdeule
    int size = 0;

    dlq_node *n = p_fq->head; //Used this to calculate end time
    EndTime = n->data->ptimearrival;
```

```
if (n == NULL)
    return;
while (n)
{
    size = size + 1;
    EndTime = EndTime + n->data->ptimeleft;
    n = n->pbck;
}                           //Now we have the end time for our schedule
for(int i = 1; i<EndTime + 1; i++){ //Looping through the whole schedule
    using the endtime
    if(Current!=NULL){              //If the current process is not null
        *(p_time) = *(p_time) + 1;   //Incrementing the time
        if(Current->ptimeleft == 0){ //If the time left for the current
            process is 0
            if(ReadyQueue.head!=NULL){ //If the ready queue is not empty
                Current=ReadyQueue.head->data; //Current process becomes
                    the head of the ready queue
                NoOfProcesses = NoOfProcesses+1; //Incrementing the no of
                    processes
            if(Current->pfirsttime == true){ //This is a check to calculate
                response time metric
                    Current->pfirsttime = false;
                    TotalResponseTime = (double) TotalResponseTime +
                        (*(p_time) - Current->ptimearrival) - 1;
                    // printf("%s%f\n","Response Time = ",
                        TotalResponseTime);
                    // TotalResponseTime = 0;
            }
                remove_from_head(&ReadyQueue);  //Removing the head of the
                    ready queue since it has now become current
            }
        }

        if(head->data->ptimearrival==i-1)      //if the head, which is at
            the head of p_fq, becomes equal to the current sys time
        {
            add_to_tail(&ReadyQueue, get_new_node(head->data)); //Adding
                the head of p_fq to the tail of the ready queue
            remove_from_head(p_fq);                          //Removing the
                head of p_fq
            if(is_empty(p_fq)!=1){                          //If p_fq is not
                empty
                head = p_fq->head;                          //Head moves to the
                    next element of p_fq as we removed head
            }
            printf("%d:",*p_time);
            printf("%s(%d):",Current->pname,Current->ptimeleft);
            print_q(&ReadyQueue);
            printf("%s",":");
            printf("\n");
        }
```

```
    else{
        printf("%d:",*p_time);
        printf("%s(%d):",Current->pname,Current->ptimeleft);
        if(is_empty(&ReadyQueue)==1){
            printf("%s\n","empty:");
        }
        else{
            print_q(&ReadyQueue);
            printf("%s",":");
            printf("\n");
        }
    }
    if(Current->ptimeleft == 1){ //This is the check for turnaround
        time, this is where the execution time becomes zero so we need
        it for our calculation
    TotalTurnaroundTime = (double) TotalTurnaroundTime + (*(p_time) -
        Current->ptimearrival);
    }
    Current->ptimeleft = Current->ptimeleft - 1;
}
else{ //If the current process is null
    *(p_time) = *(p_time) + 1; //Incrementing the time
    if(head->data->ptimearrival==i){ //If the process at the head of
        p_fq's time arrival becomes equal to the current time
        Current = head->data; //Current process becomes the head of p_fq
        NoOfProcesses = NoOfProcesses+1; //Incrementing the no of
            processes
        remove_from_head(p_fq);
        head = p_fq->head;
        printf("%d:",*p_time);
        printf("idle:");
        printf("%s\n","empty:");
        if(Current->pfirsttime == true){ //This is a check to calculate
            response time metric
                Current->pfirsttime = false;
                TotalResponseTime = 0;
                // printf("%s%f\n","Response Time = ",
                    TotalResponseTime);
                // TotalResponseTime = 0;
        }
    }
    else if(head->data->ptimearrival==0){ //Condtion to check when the
        process is arriving at time 0
        Current = head->data;
    if(Current->pfirsttime == true){ //This is a check to calculate
        response time metric
                Current->pfirsttime = false;
                TotalResponseTime = 0;
                // printf("%s%f\n","Response Time = ",
                    TotalResponseTime);
                // TotalResponseTime = 0;
```

```
                }
                remove_from_head(p_fq);
                NoOfProcesses = NoOfProcesses+1;
                head = p_fq->head;
                printf("%d:",*p_time);
                printf("%s:",Current->pname);
                printf("%s\n","empty:");
                Current->ptimeleft = Current->ptimeleft - 1;
            }
            else{
                printf("%d:",*p_time);
                printf("idle:");
                printf("%s\n","empty:");
            }
        }
    }
    //Metrics Calculation
    Throughput = (double)NoOfProcesses/EndTime;
    AverageTurnaroundTime = (double) TotalTurnaroundTime/NoOfProcesses;
    AverageResponseTime = (double) TotalResponseTime/NoOfProcesses;
    printf("%s%d\n","No Of Processess = ", NoOfProcesses);
    printf("%s%f\n","Throughput = ", Throughput);
    printf("%s%f\n","Turnaround Time = ", AverageTurnaroundTime);
    printf("%s%f","Response Time = ", AverageResponseTime);
}
```

This is the code that I wrote to implement the Shortest Job First scheduler. Comments have been added for explanation. This code is different from that implemented on Hackerank as it includes the additional code to calculate metrics.

```
void sched_SJF(dlq *const p_fq, int *p_time)
{
    int NoOfProcesses = 0;
    double Throughput = 0;
    double AverageTurnaroundTime = 0;
    double TotalTurnaroundTime = 0;
    double AverageResponseTime = 0;
    double TotalResponseTime = 0;
    struct pcb *Current = NULL;
    dlq ReadyQueue;
    ReadyQueue.head = NULL;
    ReadyQueue.tail = NULL;
    dlq_node *head = p_fq->head;
    int EndTime = 0;
    int size = 0;
    dlq_node *n = p_fq->head;
    EndTime = n->data->ptimearrival;
    if (n == NULL)
        return;
    while (n)
    {
```

```
    size = size + 1;
    EndTime = EndTime + n->data->ptimeleft;
    n = n->pbck;
}
for(int i = 1; i<EndTime + 1; i++){
    if(Current!=NULL){
        *(p_time) = *(p_time) + 1;
        if(Current->ptimeleft == 0){
            if(ReadyQueue.head!=NULL){
                Current=ReadyQueue.head->data;
                NoOfProcesses = NoOfProcesses+1;
                if(Current->pfirsttime == true){
                    Current->pfirsttime = false;
                    TotalResponseTime = (double) TotalResponseTime +
                        (*(p_time) - Current->ptimearrival) - 1;
                    // printf("%s%f\n","Response Time = ",
                        TotalResponseTime);
                    // TotalResponseTime =0;
                }
                remove_from_head(&ReadyQueue);
            }
        }
        if(head->data->ptimearrival==i-1)
        {
            add_to_tail(&ReadyQueue, get_new_node(head->data));
            sort_by_timetocompletion(&ReadyQueue); //Everything is same as
                FIFO except that where we have added to the tail in Ready
                Queue, we sort it by time to completion
            remove_from_head(p_fq);
            if(is_empty(p_fq)!=1){
                head = p_fq->head;
            }
            printf("%d:",*p_time);
            printf("%s:",Current->pname);
            print_q(&ReadyQueue);
            printf("%s",":");
            printf("\n");
        }
        else{
            printf("%d:",*p_time);
            printf("%s:",Current->pname);
            if(is_empty(&ReadyQueue)==1){
                printf("%s\n","empty:");
            }
            else{
                print_q(&ReadyQueue);
                printf("%s",":");
                printf("\n");
            }
        }
        if(Current->ptimeleft == 1){
```

```
                TotalTurnaroundTime = (double) TotalTurnaroundTime + (*(p_time)
                    - Current->ptimearrival);
            }
            Current->ptimeleft = Current->ptimeleft - 1;
        }
        else{
            *(p_time) = *(p_time) + 1;
            if(head->data->ptimearrival==i){
                Current = head->data;
                NoOfProcesses = NoOfProcesses+1;
                if(Current->pfirsttime == true){
                    Current->pfirsttime = false;
                    TotalResponseTime = 0;
                    // printf("%s%f\n","Response Time = ", TotalResponseTime);
                    // TotalResponseTime = 0;
                }
                remove_from_head(p_fq);
                head = p_fq->head;
                printf("%d:",*p_time);
                printf("idle:");
                printf("%s\n","empty:");
            }
            else if(head->data->ptimearrival==0){
                Current = head->data;
                NoOfProcesses = NoOfProcesses+1;
                if(Current->pfirsttime == true){
                    Current->pfirsttime = false;
                    TotalResponseTime = 0;
                    printf("%s%f\n","Response Time = ", TotalResponseTime);
                }
                remove_from_head(p_fq);
                head = p_fq->head;
                printf("%d:",*p_time);
                printf("%s:",Current->pname);
                printf("%s\n","empty:");
                Current->ptimeleft = Current->ptimeleft - 1;
            }
            else{
                printf("%d:",*p_time);
                printf("idle:");
                printf("%s\n","empty:");
            }
        }
    }
    Throughput = (double)NoOfProcesses/EndTime;
    AverageTurnaroundTime = (double) TotalTurnaroundTime/NoOfProcesses;
    AverageResponseTime = (double) TotalResponseTime/NoOfProcesses;
    printf("%s%f\n","Throughput = ", Throughput);
    printf("%s%f\n","Turnaround Time = ", AverageTurnaroundTime);
    printf("%s%f","Response Time = ", AverageResponseTime);
}
```

This is the code that I wrote to implement the Shortest Time To Completion First scheduler. Comments have been added for explanation. This code is different from that implemented on Hackerank as it includes the additional code to calculate metrics.

```
void sched_STCF(dlq *const p_fq, int *p_time){
    int NoOfProcesses = 0;
    double Throughput = 0;
    double AverageTurnaroundTime = 0;
    double TotalTurnaroundTime = 0;
    double AverageResponseTime = 0;
    double TotalResponseTime = 0;
    struct pcb *Current = NULL;
    dlq ReadyQueue;
    ReadyQueue.head = NULL;
    ReadyQueue.tail = NULL;
    dlq_node *head = p_fq->head;
    int EndTime = 0;
    int size = 0;
    dlq_node *n = p_fq->head;
    EndTime = n->data->ptimearrival;
    if (n == NULL)
        return;
    while (n)
    {
        size = size + 1;
        EndTime = EndTime + n->data->ptimeleft;
        n = n->pbck;
    }
    for(int i = 1; i<EndTime + 1; i++){
        if(Current!=NULL){
            *(p_time) = *(p_time) + 1;
            if(Current->ptimeleft == 0){
                if(ReadyQueue.head!=NULL){
                    Current=ReadyQueue.head->data;
                    NoOfProcesses = NoOfProcesses+1;
                    if(Current->pfirsttime == true){
                        Current->pfirsttime = false;
                        TotalResponseTime = (double) TotalResponseTime +
                            (*(p_time) - Current->ptimearrival) - 1;
                        // printf("%s%f\n","Response Time = ",
                            TotalResponseTime);
                        // TotalResponseTime = 0;
                    }
                    remove_from_head(&ReadyQueue);
                }
            }
            if(head->data->ptimearrival==i-1)
            {
                 //The only difference between SJF and STCF code is that we
                    have added a check for comparing, if less, Current is
                    replaced, if not head gets added to the tail of ready queue
```

```
            if(head->data->ptimeleft<Current->ptimeleft){
                add_to_tail(&ReadyQueue, get_new_node(Current));
                sort_by_timetocompletion(&ReadyQueue);
                Current = head->data;
                if(Current->pfirsttime == true){
                    Current->pfirsttime = false;
                    TotalResponseTime = (double) TotalResponseTime +
                        (*(p_time) - Current->ptimearrival) - 1;
                    // printf("%s%f\n","Response Time = ",
                        TotalResponseTime);
                    //TotalResponseTime = 0;
                }
                remove_from_head(p_fq);
                if(is_empty(p_fq)!=1){
                    head = p_fq->head;
                }
            }
            else{
                add_to_tail(&ReadyQueue, get_new_node(head->data));
                sort_by_timetocompletion(&ReadyQueue);
                remove_from_head(p_fq);
                if(is_empty(p_fq)!=1){
                    head = p_fq->head;
                }
            }

            printf("%d:",*p_time);
            printf("%s:",Current->pname);
            print_q(&ReadyQueue);
            printf("%s",":");
            printf("\n");
        }
        else{
            printf("%d:",*p_time);
            printf("%s:",Current->pname);
            if(is_empty(&ReadyQueue)==1){
                printf("%s\n","empty:");
            }
            else{
                print_q(&ReadyQueue);
                printf("%s",":");
                printf("\n");
            }
        }
        if(Current->ptimeleft == 1){
            TotalTurnaroundTime = (double) TotalTurnaroundTime + (*(p_time)
                - Current->ptimearrival);
        }
        Current->ptimeleft = Current->ptimeleft - 1;
    }
    else{
```

```
            *(p_time) = *(p_time) + 1;
            if(head->data->ptimearrival==i){
                Current = head->data;
                remove_from_head(p_fq);
                head = p_fq->head;
                printf("%d:",*p_time);
                printf("idle:");
                printf("%s\n","empty:");
                NoOfProcesses = NoOfProcesses+1;
                if(Current->pfirsttime == true){
                    Current->pfirsttime = false;
                    TotalResponseTime = 0;
                    // printf("%s%f\n","Response Time = ", TotalResponseTime);
                    //TotalResponseTime = 0;
                }
            }
            else if(head->data->ptimearrival==0){
                Current = head->data;
                NoOfProcesses = NoOfProcesses+1;
                if(Current->pfirsttime == true){
                    Current->pfirsttime = false;
                    TotalResponseTime = 0;
                    // printf("%s%f\n","Response Time = ", TotalResponseTime);
                    //TotalResponseTime = 0;
                }
                remove_from_head(p_fq);
                head = p_fq->head;
                printf("%d:",*p_time);
                printf("%s:",Current->pname);
                printf("%s\n","empty:");
                Current->ptimeleft = Current->ptimeleft - 1;
            }
            else{
                printf("%d:",*p_time);
                printf("idle:");
                printf("%s\n","empty:");
            }
        }
    }
    Throughput = (double)NoOfProcesses/EndTime;
    AverageTurnaroundTime = (double) TotalTurnaroundTime/NoOfProcesses;
    AverageResponseTime = (double) TotalResponseTime/NoOfProcesses;
    printf("%s%d\n","No Of Processess = ", NoOfProcesses);
    printf("%s%f\n","Throughput = ", Throughput);
    printf("%s%f\n","Turnaround Time = ", AverageTurnaroundTime);
    printf("%s%f","Response Time = ", AverageResponseTime);
}
```

This is the code that I wrote to implement the Round Robin. Comments have been added for explanation. This code is different from that implemented on Hackerank as it

includes the additional code to calculate metrics.

```
void sched_RR(dlq *const p_fq, int *p_time)
{
    int NoOfProcesses = 0;
    double Throughput = 0;
    double AverageTurnaroundTime = 0;
    double TotalTurnaroundTime = 0;
    double AverageResponseTime = 0;
    double TotalResponseTime = 0;
    struct pcb *Current = NULL;
    dlq ReadyQueue;
    ReadyQueue.head = NULL;
    ReadyQueue.tail = NULL;
    dlq_node *head = p_fq->head;
    int EndTime = 0;
    int size = 0;
    dlq_node *n = p_fq->head;
    EndTime = n->data->ptimearrival;
    if (n == NULL)
        return;
    while (n)
    {
        size = size + 1;
        EndTime = EndTime + n->data->ptimeleft;
        n = n->pbck;
    }
    for(int i = 1; i<EndTime + 1; i++){
        if(Current!=NULL){
            *(p_time) = *(p_time) + 1;
            if(Current->ptimeleft == 0){
                if(ReadyQueue.head!=NULL){
                    Current=ReadyQueue.head->data;
                NoOfProcesses = NoOfProcesses+1;
                if(Current->pfirsttime == true){
                        Current->pfirsttime = false;
                        TotalResponseTime = (double) TotalResponseTime +
                            (*(p_time) - Current->ptimearrival) - 1;
                        // printf("%s%f\n","Response Time = ",
                            TotalResponseTime);
                        //TotalResponseTime = 0;
                }
                    remove_from_head(&ReadyQueue);
                }
            }
            else{ //This is the difference in code b/w FIFO and RR where we
                have added a check to shift between the current process and the
                process at the head of ReadyQueue
                if(is_empty(&ReadyQueue)!=1){
                    add_to_tail(&ReadyQueue, get_new_node(Current));
                    Current = ReadyQueue.head->data;
```

```
        if(Current->pfirsttime == true){
                Current->pfirsttime = false;
                TotalResponseTime = (double) TotalResponseTime +
                    (*(p_time) - Current->ptimearrival) - 1;
                // printf("%s%f\n","Response Time = ",
                    TotalResponseTime);
                //TotalResponseTime = 0;
        }
            remove_from_head(&ReadyQueue);
        }
    }

    if(head->data->ptimearrival==i-1)
    {
        add_to_tail(&ReadyQueue, get_new_node(head->data));
        remove_from_head(p_fq);
        if(is_empty(p_fq)!=1){
            head = p_fq->head;
        }
        printf("%d:",*p_time);
        printf("%s:",Current->pname);
        print_q(&ReadyQueue);
        printf("%s",":");
        printf("\n");
    }
    else{
        printf("%d:",*p_time);
        printf("%s:",Current->pname);
        if(is_empty(&ReadyQueue)==1){
            printf("%s\n","empty:");
        }
        else{
            print_q(&ReadyQueue);
            printf("%s",":");
            printf("\n");
        }
    }
    if(Current->ptimeleft == 1){
    TotalTurnaroundTime = (double) TotalTurnaroundTime + (*(p_time) -
        Current->ptimearrival);
    }
    Current->ptimeleft = Current->ptimeleft - 1;
}
else{
    *(p_time) = *(p_time) + 1;
    if(head->data->ptimearrival==i){
        Current = head->data;
        NoOfProcesses = NoOfProcesses+1;
        if(Current->pfirsttime == true){
                Current->pfirsttime = false;
```

```
                        TotalResponseTime = 0;
                        // printf("%s%f\n","Response Time = ",
                            TotalResponseTime);
                        //TotalResponseTime = 0;
                    }
                    remove_from_head(p_fq);
                    head = p_fq->head;
                    printf("%d:",*p_time);
                    printf("idle:");
                    printf("%s\n","empty:");

                }
                else if(head->data->ptimearrival==0){
                    Current = head->data;
                    NoOfProcesses = NoOfProcesses+1;
                    if(Current->pfirsttime == true){
                        Current->pfirsttime = false;
                        TotalResponseTime = 0;
                        // printf("%s%f\n","Response Time = ",
                            TotalResponseTime);
                        //TotalResponseTime = 0;
                    }
                    remove_from_head(p_fq);
                    head = p_fq->head;
                    printf("%d:",*p_time);
                    printf("%s:",Current->pname);
                    printf("%s\n","empty:");
                    Current->ptimeleft = Current->ptimeleft - 1;
                }
                else{
                    printf("%d:",*p_time);
                    printf("idle:");
                    printf("%s\n","empty:");
                }
            }
        }
        Throughput = (double)NoOfProcesses/EndTime;
        AverageTurnaroundTime = (double) TotalTurnaroundTime/NoOfProcesses;
        AverageResponseTime = (double) TotalResponseTime/NoOfProcesses;
        printf("%s%d\n","NOP = ", NoOfProcesses);
        printf("%s%f\n","Throughput = ", Throughput);
        printf("%s%f\n","Turnaround Time = ", AverageTurnaroundTime);
        printf("%s%f","Response Time = ", AverageResponseTime);
}
```

2 MakeFile

Make file for running the Schedulers has been added to the zip file. Use the commands 'make build' and then 'make run' to run the generated output file. Once you are inside the .out file, you'll have to insert the input.

```
build:
	gcc main.c -o out
run:
	./out
clean:
	rm out
rebuild: clean build
```

3 Performance Metrics Comparison

	FIFO			SJF		
	TURNAROUND TIME	THROUGHPUT	RESPONSE TIME	TURNAROUND TIME	THROUGHPUT	RESPONSE TIME
TEST CASE 0	10.333	0.167	5	9	0.167	3.67
TEST CASE 2	10.333	0.167	5	9	0.167	3.67
TEST CASE 5	19.167	0.154	12.67	16.83	0.154	10.3
TEST CASE 10	27.67	0.107	18.33	23	0.107	13.67
TEST CASE 13	36.5	0.113	27.63	27.38	0.113	18.5
AVERAGES	20.8006	0.1416	13.726	17.042	0.1416	9.962
	STCF			ROUNDROBIN		
	TURNAROUND TIME	THROUGHPUT	RESPONSE TIME	TURNAROUND TIME	THROUGHPUT	RESPONSE TIME
TEST CASE 0	9	0.167	3.67	12.33	0.167	1
TEST CASE 2	9	0.167	3.67	12.33	0.167	1
TEST CASE 5	16.83	0.154	10.33	26	0.154	2.5
TEST CASE 10	22.67	0.107	12.5	36.33	0.107	2.5
TEST CASE 13	27.25	0.113	17.88	49.63	0.113	3.5
AVERAGES	16.95	0.1416	9.61	27.324	0.1416	2.1

–Best Avg ResponseTime and TurnaroundTime is highlighted in Green. Avg Throughput remains the same for all - highlighted in Blue.

In the figure above, I have shown the result of running all 4 different scheduling policies on all five test cases and then we've calculated the average Turnaround Time, average Throughput and average Response Time for each of the scheduling policies. From our results, we see that the Average Throughput remains the same across all four policies, this is because the throughput is not changing for individual test cases across different policies. Now why is the throughput not changing? This is because the total time taken by the schedule for a particular test case remains the same along with the number of processes and Throughput is simply the No of Processes in a Schedule divided by the total time taken by that schedule. From our results, we see that RoundRobin gives us the best Response Time, while STCF gives us the best Turnaround Time. Now, lets discuss I calculated the Turnaround time -

```
if(Current->ptimeleft == 1){ //This is the check for turnaround time, this is
    where the execution time becomes zero so we need it for our calculation
TotalTurnaroundTime = (double) TotalTurnaroundTime + (*(p_time) -
    Current->ptimearrival);
          }
```

In all the four policies I used the same code to calculate the Turn around time (and then averaged it at the end). The formulae for calculating the turnaround time is the Time of completion of a process subtracted by the time of arrival of that process. We already know the time of arrival of every process as it is given to us, however to find the time of completion, I have added the condition as stated above, this is the the place in my code where the my current process ends, so this is exactly the time I want and am using it as the time of completion to calculate turn around time.

Now, moving on towards the calculation of response time. By definition, the formulae for calculating response time is simply the time of FIRST run subtracted by the time of arrival. We already have the time of arrival, what we need is the time of first run. To get this, we need to add a check so that we know if our process is running for the first time or not. For this purpose, I have modified the PCB to add a check and have initialized it to being false during Tokenization. Modified PCB -

```
struct pcb
{
   unsigned int pid;
   char pname[20];
   unsigned int ptimeleft;
   unsigned int ptimearrival;
    bool pfirsttime;
};
```

The condition I have added in the code to calculate response time-

```
if(Current->pfirsttime == true){
       Current->pfirsttime = false;
                      TotalResponseTime = (double) TotalResponseTime +
                          (*(p_time) - Current->ptimearrival) - 1;
               }
```

This condition has been added in our code wherever current is not NULL initially and we are updating the current. However, where the current is NULL initially, the condition changes a bit to become -

```
if(Current->pfirsttime == true){
                      Current->pfirsttime = false;
                      TotalResponseTime = 0;
               }
```

This change is because the response time should be zero when there's nothing on current. To further understand this lets dry run test case zero on FIFO to calculate the response time of each individual process. This is the output that we can when we run test case zero and print the process time of each individual process -

```
PS C:\Users\Youshay\Desktop\Os_Assignment_2> make build
gcc main.c -o out
PS C:\Users\Youshay\Desktop\Os_Assignment_2> make run
./out
3
FIFO
P1:12:7:3
P2:15:3:5
P3:1:6:2
END
1:idle:empty:
2:idle:empty:
P3 Response Time = 0.000000
3:P3(6):empty:
4:P3(5):P1(7),:
5:P3(4):P1(7),:
6:P3(3):P1(7),P2(3),:
7:P3(2):P1(7),P2(3),:
8:P3(1):P1(7),P2(3),:
P1 Response Time = 5.000000
9:P1(7):P2(3),:
10:P1(6):P2(3),:
11:P1(5):P2(3),:
12:P1(4):P2(3),:
13:P1(3):P2(3),:
14:P1(2):P2(3),:
15:P1(1):P2(3),:
P2 Response Time = 10.000000
16:P2(3):empty:
17:P2(2):empty:
18:P2(1):empty:
```

In the output you can see that the response time of p3 is zero, this is because the arrival time of p3 is 2, and at two we can see that current is idle so automatically according to our condition the response time at p3 becomes zero. The response time of p1 is 5. According to our input, the arrival time of p1 is 3, and from 9 we can see it running on our output which actually means that it has started running at 8 and is being shown at 9 (This is because according to Hackerank test cases, even when an input is coming at 2 we print it at 3 like it can be seen in the case of p3) so our response time for p1 becomes 8 minus 3 which is 5 as can be seen. Similarly for p2, the arrival is at 5, and it first runs at 15 so the response time becomes 15 minus 5 which is 10 as can be seen. No we calculate the average by adding them all which is 0 plus 10 plus 5 which is 15 and then diving by the number of processes i.e 3 gives us 15 divided by 5 which is 3 (as can be seen in the metrics comparison chart above). The same method is applied for all scheduling policies.