Universidade do Vale do Itajaí – UNIVALI
Centro de Ciências Tecnológicas da Terra e do Mar – CTTMar
Ciência da Computação – Sistemas Operacionais– Felipe Viel

Avaliação 04 – Escalonamento

Acadêmicos: Pamela Bandeira Gerber e Vinicius Stumpf

18/05/22

Introdução

O presente trabalho apresentado é uma implementação de três algoritmos de escalonamento, sendo eles o Round-Robin (RR), Shartest-Job First (SJF) e Round-Robin com prioridade baseado em FIFO (RR_p_FIFO). Foi realizado a análise de desempenho de cada algoritmo em questão, no qual utilizamos execuções com 5, 10 e 20 tasks, utilizado como base os arquivos disponibilizados pelo professor, na qual foi necessário implementar apenas os arquivos do escalonador fazendo pequenas alterações no comportamento da lista para se adequar as nossas necessidades.

Round-Robin (RR): é um algoritmo escalonador de tarefas que consiste em dividir o tempo de uso da CPU. Cada processo recebe uma fatia de tempo, esse tempo é chamado Quantum. Os processos são todos armazenados em Fila (Buffer) circular.

Shartest-Job First (SJF): é um algoritmo no qual o processo com o menor tempo de execução é escolhido para a próxima execução. Esse método de agendamento pode ser preemptivo ou não preemptivo. Reduz significativamente o tempo médio de espera para outros processos que aguardam a execução.

First-In First-Out (FIFO): é um algoritmo escalonador comumente usado para uma fila de tarefas que enfileira os processos na ordem em que eles chegam na fila de prontos.

Desenvolvimento

Round-Robin

Foi implementado uma struct para grava a posição inicial da lista escadeada na biblioteca schedule_rr.h (Figura 1).

```
struct filaAptos{
    struct node * fila;
    int lastId;
};
```

Figura 1.

Na função responsável por adicionar as tasks na lista foi realizado a criação de um novo espaço de memória para armazenar a nova task utilizando a biblioteca malloc.h, e também, foi feito a chamada da função insert passado como atributos a lista e a posição da memória da nova tesk para assim ser realizado o incremento na fila (Figura 2).

```
void add(char *name, int priority, int burst, struct filaAptos *fila) {

    struct task *newTask = malloc( size: sizeof(struct task)); // criando u
    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;
    newTask->tid = fila->lastId +
                   1; // é pego a variável responsável por guardar o últi
    insert( head: &fila->fila,  task: newTask);// chama a função para alocar
    fila->lastId++;
}
```

Figura 2.

Nessa função (Figura 3) localizada no arquivo list.c foi realizado a modificação para que a lista deixasse de se comportar como uma pilha e começasse a agir como uma fila para se adequar melhor a necessidade do escalonador.

```
void insert(struct node **head, Task *newTask) {
    //criadno um novo nó para a fila
    struct  node * newNode = malloc( size: sizeof(struct node));
    newNode->task = newTask;
    newNode->next = NULL;

    if(*head == NULL){
        *head = newNode;
    } else {
        struct node * nav;
        nav = *head;
        while (nav->next != NULL){
            nav = nav->next;
        }
        nav->next = newNode;
    }
}
```

Figura 3.

A função (Figura 4) que é a responsável por realizar o escalonamento foi adicionado uma variável do tipo int chamado time para ser usado como o tempo limite de execução de cada tarefa e uma variável que armazenará a tarefa que será executada pela CPU. Foi implementado um laço de repetição infinito, a primeira coisa que será feita nesse laço será a verificação se a lista já não está vazia, se caso sim ele irá sair do laço assim finalizando a função.

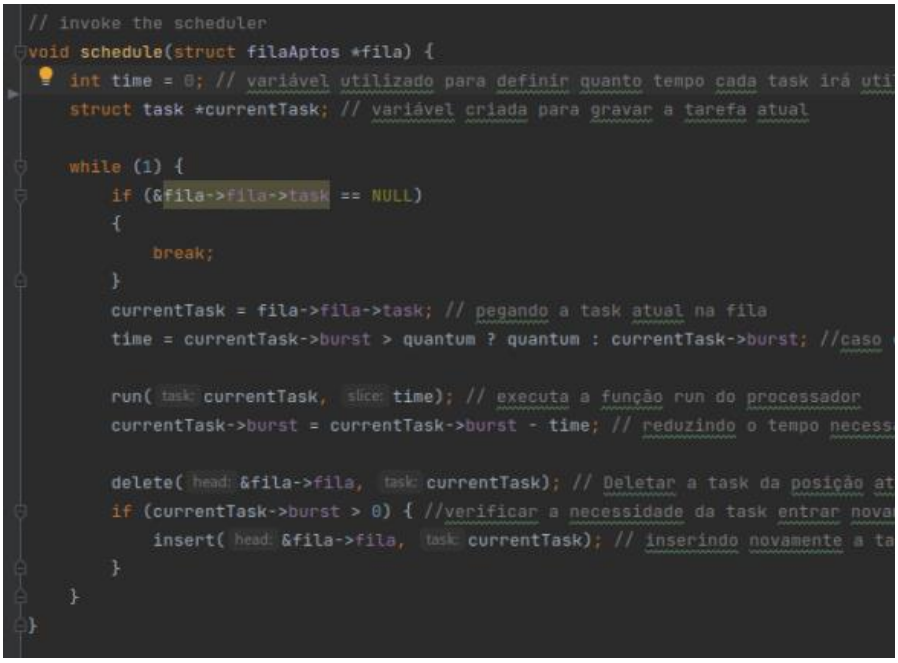


Figura 4.

Logo em seguida é pego a tarefa que se encontra na primeira posição da lista e jogado para variável responsável por armazenar a task atual, após isso é realizado uma operação com um if ternário para determinar o tempo de execução da task na CPU, caso o tempo necessário para finalizar a execução da tarefa é maior que o tempo limite de execução definido na variável quantum, o time recebera o valor de quantum em si, caso o contrário será utilizado como valor o tempo necessário para execução da própria task. É realiza a chamada da função run da biblioteca CPU.h para a execução da tarefa, e é reduzido o tempo que ela ficou em execução do tempo necessário para a execução daquela task . Por final é realizado o delete da task na fila e verificado a necessidade de inserir ela novamente caso ainda não tenha sido concluída totalmente.

Na função run da biblioteca CPU.h foi inserido o comando sleep() da biblioteca unistd.h para realizar o delay do processamento da CPU, foi passado como argumento para essa função o silice dividido por 60 para que seja pausado em milissegundos (Figura 5).

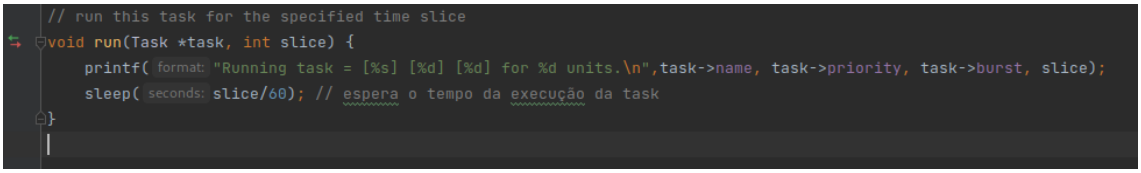


Figura 5.

Round-Robin #define 5 tasks

```
Running task = [T1] [1] [50] for 50 units.
Running task = [T2] [2] [25] for 25 units.
Running task = [T3] [3] [150] for 50 units.
Running task = [T4] [2] [50] for 50 units.
Running task = [T5] [4] [35] for 35 units.
Running task = [T6] [5] [5] for 5 units.
Running task = [T3] [3] [100] for 50 units.
Running task = [T3] [3] [50] for 50 units.

Process finished with exit code 0
```

Round-Robin #define 10 tasks

```
Running task = [T1] [1] [50] for 50 units.
Running task = [T2] [2] [25] for 25 units.
Running task = [T3] [3] [150] for 50 units.
Running task = [T4] [2] [50] for 50 units.
Running task = [T5] [4] [35] for 35 units.
Running task = [T6] [5] [5] for 5 units.
Running task = [T7] [7] [16] for 16 units.
Running task = [T8] [8] [80] for 50 units.
Running task = [T9] [9] [200] for 50 units.
Running task = [T10] [6] [40] for 40 units.
Running task = [T3] [3] [100] for 50 units.
Running task = [T8] [8] [30] for 30 units.
Running task = [T9] [9] [150] for 50 units.
Running task = [T3] [3] [50] for 50 units.
Running task = [T9] [9] [100] for 50 units.
Running task = [T9] [9] [50] for 50 units.

Process finished with exit code 0
```

Round-Robin #define 20 tasks

```
Running task = [T1] [1] [50] for 50 units.
Running task = [T2] [2] [25] for 25 units.
Running task = [T3] [3] [150] for 50 units.
Running task = [T4] [2] [50] for 50 units.
Running task = [T5] [4] [35] for 35 units.
Running task = [T6] [5] [5] for 5 units.
Running task = [T7] [7] [16] for 16 units.
Running task = [T8] [8] [80] for 50 units.
Running task = [T9] [9] [200] for 50 units.
Running task = [T10] [6] [40] for 40 units.
Running task = [T11] [1] [50] for 50 units.
Running task = [T12] [2] [25] for 25 units.
Running task = [T13] [3] [150] for 50 units.
Running task = [T14] [2] [50] for 50 units.
Running task = [T15] [4] [35] for 35 units.
Running task = [T16] [5] [5] for 5 units.
Running task = [T17] [7] [16] for 16 units.
Running task = [T18] [8] [80] for 50 units.
Running task = [T19] [9] [200] for 50 units.
Running task = [T20] [6] [40] for 40 units.
Running task = [T3] [3] [100] for 50 units.
Running task = [T8] [8] [30] for 30 units.
Running task = [T9] [9] [150] for 50 units.
Running task = [T13] [3] [100] for 50 units.
Running task = [T18] [8] [30] for 30 units.
Running task = [T19] [9] [150] for 50 units.
Running task = [T3] [3] [50] for 50 units.
Running task = [T9] [9] [100] for 50 units.
Running task = [T13] [3] [50] for 50 units.
Running task = [T19] [9] [100] for 50 units.
Running task = [T9] [9] [50] for 50 units.
Running task = [T19] [9] [50] for 50 units.
```

Lista de tasks

driver.c

```
T1, 1, 50
T2, 2, 25
T3, 3, 150
T4, 2, 50
T5, 4, 35
T6, 5, 5
T7, 7, 16
T8, 8, 80
T9, 9, 200
T10, 6, 40
```

Shortest-Job First

Tanto a função add da biblioteca schedule_sfj.h quanto a função run da biblioteca CPU.h continuam a mesma coisa de no algoritmo do escaldador Round-Robin (Figura 6).

```c
void run(Task *task, int slice) {
    printf( format: "Running task = [%s] [%d] [%d] for %d units.\n",task->name, task->priority, task->burst, slice);
    sleep( seconds: slice/60); // espera o tempo da execução da task
}

void add(char *name, int priority, int burst, struct filaAptos * fila){

    struct task * newTask = malloc( size: sizeof(struct task)); // criando
    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;
    newTask->tid = fila->lastId + 1; // é pego a variável responsável por
    insert( head: &fila->fila,  task: newTask);// chama a função para alocar
    fila->lastId++;
}
```

Figura 6.

Na função insert (Figura 7) do arquivo list.c foi feito a alteração para que se insira a task de maneira ordenada, sendo que a tarefa que necessita menos tempos de execução irá ficar no início da fila e a que tiver o maior tempo de execução irá para o final da fila de aptos.

Figura 7.

Essa função (Figura 8) da biblioteca schedule_sfj focou relativamente mais simples do que a implementada no escalonador Round-Robin já que as tasks são inseridas na lista de forma já ordenada, na função existe um laço infinito para percorrer a lista, dentro dele é verificado se a lista está vazia como condição de parada, também é pego a task da primeira posição da fila como no escalonado Round-Robin, executada a task e deletada da fila assim que é finalizado.

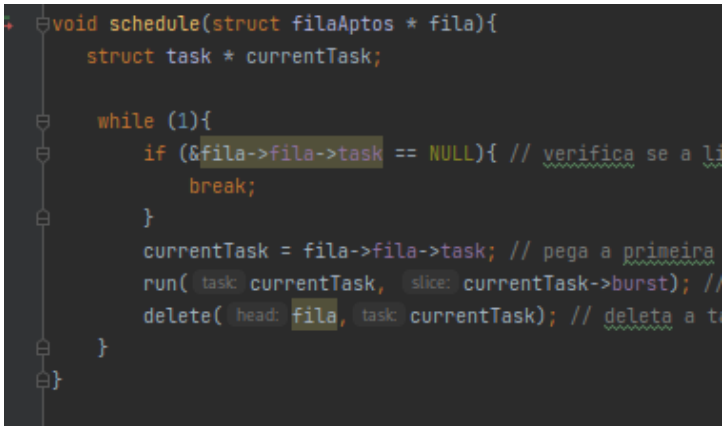


Figura 8.

Shortest-Job First #define 5 tasks

```
Running task = [T7] [40] [5] for 5 units.
Running task = [T4] [40] [28] for 28 units.
Running task = [T2] [40] [40] for 40 units.
Running task = [T1] [40] [50] for 50 units.
Running task = [T3] [40] [60] for 60 units.
Running task = [T6] [40] [78] for 78 units.
Running task = [T5] [40] [80] for 80 units.
Running task = [T9] [40] [93] for 93 units.
Running task = [T8] [40] [120] for 120 units.

Process finished with exit code 0
```

Shortest-Job First #define 10 tasks

```
Running task = [T7] [40] [5] for 5 units.
Running task = [T4] [40] [28] for 28 units.
Running task = [T2] [40] [40] for 40 units.
Running task = [T1] [40] [50] for 50 units.
Running task = [T3] [40] [60] for 60 units.
Running task = [T6] [40] [78] for 78 units.
Running task = [T5] [40] [80] for 80 units.
Running task = [T9] [40] [93] for 93 units.
Running task = [T8] [40] [120] for 120 units.

Process finished with exit code 0
```

Shortest-Job First #define 20 tasks

```
Running task = [T6] [5] [5] for 5 units.
Running task = [T16] [5] [5] for 5 units.
Running task = [T7] [7] [16] for 16 units.
Running task = [T17] [7] [16] for 16 units.
Running task = [T2] [2] [25] for 25 units.
Running task = [T12] [2] [25] for 25 units.
Running task = [T5] [4] [35] for 35 units.
Running task = [T15] [4] [35] for 35 units.
Running task = [T10] [6] [40] for 40 units.
Running task = [T20] [6] [40] for 40 units.
Running task = [T1] [1] [50] for 50 units.
Running task = [T4] [2] [50] for 50 units.
Running task = [T11] [1] [50] for 50 units.
Running task = [T14] [2] [50] for 50 units.
Running task = [T8] [8] [80] for 80 units.
Running task = [T18] [8] [80] for 80 units.
Running task = [T3] [3] [150] for 150 units.
Running task = [T13] [3] [150] for 150 units.
Running task = [T9] [9] [200] for 200 units.
Running task = [T19] [9] [200] for 200 units.
```

Lista de tasks

driver.c

```
T1, 1, 50
T2, 2, 25
T3, 3, 150
T4, 2, 50
T5, 4, 35
T6, 5, 5
T7, 7, 16
T8, 8, 80
T9, 9, 200
T10, 6, 40
```

Round-Robin_FIFO

Foi realizado a implementação de duas struct que serão utilizadas para a criação de múltiplas filas de aptos, na primeira struct é a responsável pelo nó dessa lista e a segunda é responsável por armazenar o ponteiro inicial da lista (Figura 9).

```c
#define MIN_PRIORITY 1
#define MAX_PRIORITY 10

static int quantum = 60;

struct FilaDeAptos{
    struct node * filaAdp;
    int prioridade;
    struct FilaDeAptos * nextFila;
};

struct FilaDePrioridade{

    struct FilaDeAptos * filaPri;
    int lastId;

};
```

Figura 9.

Na função responsável por adicionar as tasks na lista foi realizado a criação de um novo espaço de memória para armazenar a nova task utilizando a biblioteca malloc.h. Foi criado um navegador que vai receber a primeira posição das listas e será verificado se a primeira posição é igual a NULL, caso for será criado um novo espaço de memória para armazenar as listas de aptos em si, vai setar a prioridade que esta recebeu no parâmetro acima para aquela fila dando insert na nova task na fila criada, caso não, irá ser verificado se aquele conjunto de fila tem a prioridade que a task está querendo adicionar, se for maior vai empurrar a fila para trás jogando aquele novo conjunto de fila no lugar dela, caso o primeiro não for maior, irá fazer um while procurando até encontrar a lista na qual a prioridade da task está querendo inserir, ou caso chegar no final e ele criar um novo conjunto de lista novamente, ou encontrar alguma prioridade maior do que está tentando inserir, adiciona a lista de forma ordenada sendo criado um novo e substituirá o que existe empurrando ela para trás e colocando no lugar (Figura 10).

```
void add(char *name, int priority, int burst, struct FilaDePrioridade * fila){

    struct task * newTask = malloc( Size: sizeof(struct task));
    newTask->name = name;
    newTask->priority = priority;
    newTask->burst = burst;
    newTask->tid = fila->lastId + 1;

    struct FilaDeAptos * nav;
    nav = fila->filaPri;

    if(fila->filaPri == NULL){ // verifica se a lista de prioridade esta vazia
        struct FilaDeAptos * newFilaDeAptos = malloc( Size: sizeof(struct FilaDeAptos));
        newFilaDeAptos->prioridade = priority;
        insert( head: &newFilaDeAptos->filaAdp, task: newTask);
        fila->filaPri = newFilaDeAptos;
    }else if(fila->filaPri->prioridade > priority){
        struct FilaDeAptos * newFilaDeAptos = malloc( Size: sizeof(struct  FilaDeAptos));
        newFilaDeAptos->prioridade = priority;
        insert( head: &newFilaDeAptos->filaAdp,  task: newTask);
        newFilaDeAptos->nextFila = fila->filaPri;
        fila->filaPri = newFilaDeAptos;
    } else {
        while (1){
            if(nav->prioridade == priority){
                insert( head: &nav->filaAdp, task: newTask);
                break;
            } else if (nav->nextFila == 0){
                struct  FilaDeAptos * newFilaDeAptos = malloc( Size: sizeof(struct FilaDeAptos));
                newFilaDeAptos->prioridade = priority;
                insert( head: &newFilaDeAptos->filaAdp, task: newTask);
                nav->nextFila= newFilaDeAptos;
                break;
            } else if(nav->nextFila->prioridade > priority){
                struct  FilaDeAptos * newFilaDeAptos = malloc( Size: sizeof(struct FilaDeAptos));
                newFilaDeAptos->prioridade = priority;
                insert( head: &newFilaDeAptos->filaAdp,  task: newTask);
                newFilaDeAptos->nextFila = nav->nextFila;
                nav->nextFila = newFilaDeAptos;
                break;
            }
            nav = nav->nextFila;
        }
    }
    fila->lastId++;
}
```

Figura 10.

A função (Figura 11) que é a responsável por realizar o escalonamento foi adicionado uma variável do tipo int chamado time para ser usado como o tempo limite de execução de cada tarefa e uma variável que armazenará a tarefa que será executada pela CPU. Irá percorrer a lista de prioridade executando a lista

que tiver maior prioridade (menor valor), depois vai passar para a próxima lista de prioridade, caso precise é pego a nova task e joga novamente no processador, dá o tempo limite verifica se precisa ou não estar na lista, fazendo isso para cada lista de prioridade.

```c
void schedule(struct  FilaDePrioridade * fila){
    int time = 0;
    struct task * currentTask;

    struct FilaDeAptos * navFilas;
    navFilas = fila->filaPri;

    while (1){
        if(navFilas == NULL){
            break;
        }
        while (1){
            if(&navFilas->filaAdp->task == NULL){
                break;
            }
            currentTask = navFilas->filaAdp->task;
            time = currentTask->burst > quantum ? quantum : currentTask->burst;
            run( task: currentTask, slice: time);
            currentTask->burst = currentTask->burst - time;
            delete( head: &navFilas->filaAdp, task: currentTask);
            if (currentTask->burst > 0){
                insert( head: &navFilas->filaAdp, task: currentTask);
            }
        }
        navFilas = navFilas->nextFila;
    }
```

Figura 11.

Round-Robin_FIFO #define 20 tasks

```
Running task = [T1] [1] [50] for 50 units.
Running task = [T11] [1] [50] for 50 units.
Running task = [T2] [2] [25] for 25 units.
Running task = [T4] [2] [50] for 50 units.
Running task = [T12] [2] [25] for 25 units.
Running task = [T14] [2] [50] for 50 units.
Running task = [T3] [3] [150] for 50 units.
Running task = [T13] [3] [150] for 50 units.
Running task = [T3] [3] [100] for 50 units.
Running task = [T13] [3] [100] for 50 units.
Running task = [T3] [3] [50] for 50 units.
Running task = [T13] [3] [50] for 50 units.
Running task = [T5] [4] [35] for 35 units.
Running task = [T15] [4] [35] for 35 units.
Running task = [T6] [5] [5] for 5 units.
Running task = [T16] [5] [5] for 5 units.
Running task = [T10] [6] [40] for 40 units.
Running task = [T20] [6] [40] for 40 units.
Running task = [T7] [7] [16] for 16 units.
Running task = [T17] [7] [16] for 16 units.
Running task = [T8] [8] [80] for 50 units.
Running task = [T18] [8] [80] for 50 units.
Running task = [T8] [8] [30] for 30 units.
Running task = [T18] [8] [30] for 30 units.
Running task = [T9] [9] [200] for 50 units.
Running task = [T19] [9] [200] for 50 units.
Running task = [T9] [9] [150] for 50 units.
Running task = [T19] [9] [150] for 50 units.
Running task = [T9] [9] [100] for 50 units.
Running task = [T19] [9] [100] for 50 units.
Running task = [T9] [9] [50] for 50 units.
Running task = [T19] [9] [50] for 50 units.
```

Round-Robin_FIFO #define 5 tasks

```
Running task = [T1] [1] [50] for 50 units.
Running task = [T2] [2] [25] for 25 units.
Running task = [T4] [2] [50] for 50 units.
Running task = [T3] [3] [150] for 50 units.
Running task = [T3] [3] [100] for 50 units.
Running task = [T3] [3] [50] for 50 units.
Running task = [T5] [4] [35] for 35 units.
Running task = [T6] [5] [5] for 5 units.

Process finished with exit code 0
```

Round-Robin_FIFO #define 10 tasks

```
Running task = [T1] [1] [50] for 50 units.
Running task = [T2] [2] [25] for 25 units.
Running task = [T4] [2] [50] for 50 units.
Running task = [T3] [3] [150] for 50 units.
Running task = [T3] [3] [100] for 50 units.
Running task = [T3] [3] [50] for 50 units.
Running task = [T5] [4] [35] for 35 units.
Running task = [T6] [5] [5] for 5 units.
Running task = [T10] [6] [40] for 40 units.
Running task = [T7] [7] [16] for 16 units.
Running task = [T8] [8] [80] for 50 units.
Running task = [T8] [8] [30] for 30 units.
Running task = [T9] [9] [200] for 50 units.
Running task = [T9] [9] [150] for 50 units.
Running task = [T9] [9] [100] for 50 units.
Running task = [T9] [9] [50] for 50 units.
```

Lista de tasks

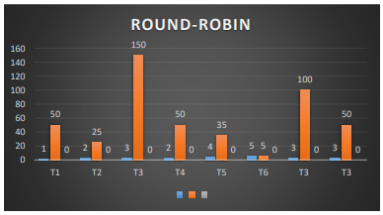
driver.c

```
T1, 1, 50
T2, 2, 25
T3, 3, 150
T4, 2, 50
T5, 4, 35
T6, 5, 5
T7, 7, 16
T8, 8, 80
T9, 9, 200
T10, 6, 40
```

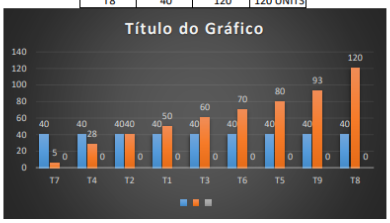
Comparações de execução

5 tasks

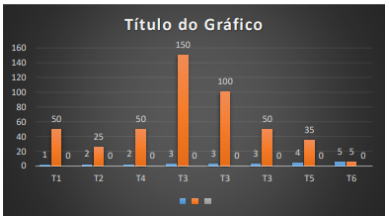
ROUND-ROBIN			
T1	1	50	50 UNITS
T2	2	25	25 UNITS
T3	3	150	50 UNITS
T4	2	50	50 UNITS
T5	4	35	35 UNITS
T6	5	5	5 UNITS
T3	3	100	50 UNITS
T3	3	50	50 UNITS



SHORTEST-JOB FIRST			
T7	40	5	5 UNITS
T4	40	28	28 UNITS
T2	40	40	40 UNITS
T1	40	50	50 UNITS
T3	40	60	60 UNITS
T6	40	70	70 UNITS
T5	40	80	80 UNITS
T9	40	93	93 UNITS
T8	40	120	120 UNITS

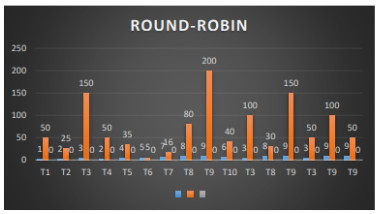


ROUND-ROBIN FIFO			
T1	1	50	50 UNITS
T2	2	25	25 UNITS
T4	2	50	50 UNITS
T3	3	150	50 UNITS
T3	3	100	50 UNITS
T3	3	50	50 UNITS
T5	4	35	35 UNITS
T6	5	5	5 UNITS

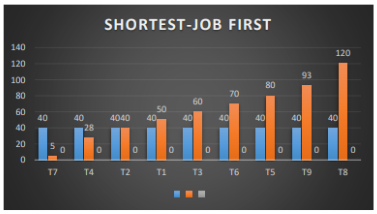


10 tasks

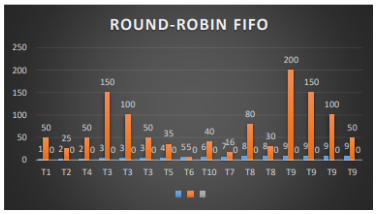
ROUND-ROBIN			
T1	1	50	50 UNITS
T2	2	25	25 UNITS
T3	3	150	50 UNITS
T4	2	50	50 UNITS
T5	4	35	35 UNITS
T6	5	5	5 UNITS
T7	7	16	16 UNITS
T8	8	80	50 UNITS
T9	9	200	50 UNITS
T10	6	40	40 UNITS
T3	3	100	50 UNITS
T8	8	30	30 UNITS
T9	9	150	50 UNITS
T3	3	50	50 UNITS
T9	9	100	50 UNITS
T9	9	50	50 UNITS



SHORTEST-JOB FIRST			
T7	40	5	5 UNITS
T4	40	28	28 UNITS
T2	40	40	40 UNITS
T1	40	50	50 UNITS
T3	40	60	60 UNITS
T6	40	70	70 UNITS
T5	40	80	80 UNITS
T9	40	93	93 UNITS
T8	40	120	120 UNITS

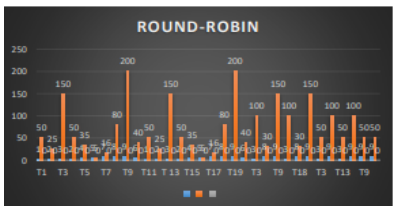


ROUND-ROBIN FIFO			
T1	1	50	50 UNITS
T2	2	25	25 UNITS
T4	2	50	50 UNITS
T3	3	150	50 UNITS
T3	3	100	50 UNITS
T3	3	50	50 UNITS
T5	4	35	35 UNITS
T6	5	5	5 UNITS
T10	6	40	40 UNITS
T7	7	16	16 UNITS
T8	8	80	50 UNITS
T8	8	30	30 UNITS
T9	9	200	50 UNITS
T9	9	150	50 UNITS
T9	9	100	50 UNITS
T9	9	50	50 UNITS

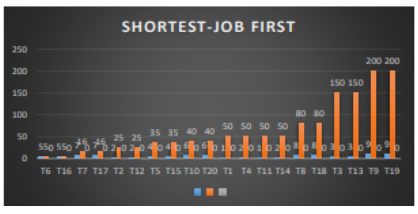


20 tasks

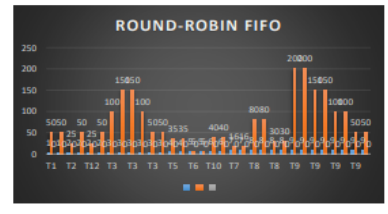
ROUND-ROBIN			
T1	1	50	50 UNITS
T2	2	25	25 UNITS
T3	3	150	50 UNITS
T4	2	50	50 UNITS
T5	4	35	35 UNITS
T6	5	5	5 UNITS
T7	7	16	16 UNITS
T8	8	80	50 UNITS
T9	9	200	50 UNITS
T10	6	40	40 UNITS
T11	1	50	50 UNITS
T 12	2	25	25 UNITS
T 13	3	150	50 UNITS
T14	2	50	50 UNITS
T15	4	35	35 UNITS
T16	5	5	5 UNITS
T17	7	16	16 UNITS
T18	8	80	50 UNITS
T19	9	200	50 UNITS
T20	6	40	40 UNITS
T3	3	100	50 UNITS
T8	8	30	30 UNITS
T9	9	150	50 UNITS
T13	3	100	50 UNITS
T18	8	30	30 UNITS
T19	9	150	50 UNITS
T3	3	50	50 UNITS
T9	9	100	50 UNITS
T13	3	50	50 UNITS
T19	9	100	50 UNITS
T9	9	50	50 UNITS
T19	9	50	50 UNITS



SHORTEST-JOB FIRST			
T6	5	5	5 UNITS
T16	5	5	5 UNITS
T7	7	16	16 UNITS
T17	7	16	16 UNITS
T2	2	25	25 UNITS
T12	2	25	25 UNITS
T5	4	35	35 UNITS
T15	4	35	35 UNITS
T10	6	40	40 UNITS
T20	6	40	40 UNITS
T1	1	50	50 UNITS
T4	2	50	50 UNITS
T11	1	50	50 UNITS
T14	2	50	50 UNITS
T8	8	80	80 UNITS
T18	8	80	80 UNITS
T3	3	150	150 UNITS
T13	3	150	150 UNITS
T9	9	200	200 UNITS
T19	9	200	200 UNITS



ROUND-ROBIN FIFO			
T1	1	50	50 UNITS
T11	1	50	50 UNITS
T2	2	25	25 UNITS
T4	2	50	50 UNITS
T12	2	25	25 UNITS
T14	2	50	50 UNITS
T3	3	100	50 UNITS
T13	3	150	50 UNITS
T3	3	150	50 UNITS
T13	3	100	50 UNITS
T3	3	50	50 UNITS
T13	3	50	50 UNITS
T5	4	35	35 UNITS
T15	4	35	35 UNITS
T6	5	5	5 UNITS
T16	5	5	5 UNITS
T10	6	40	40 UNITS
T20	6	40	40 UNITS
T7	7	16	16 UNITS
T17	7	16	16 UNITS
T8	8	80	50 UNITS
T18	8	80	50 UNITS
T8	8	30	30 UNITS
T18	8	30	30 UNITS
T9	9	200	50 UNITS
T19	9	200	50 UNITS
T9	9	150	50 UNITS
T19	9	150	50 UNITS
T9	9	100	50 UNITS
T19	9	100	50 UNITS
T9	9	50	50 UNITS
T19	9	50	50 UNITS



Conclusão

Conclui-se que na parte de implementação o Shortest-Job First ganha por ser mais simples, ocupando menos espaço de memória para executar a ordenação e escalonamento das tasks.

Já na parte de espera da execução para cada task o algoritmo Round Robin é melhor, pois limita para cada task um tempo específico na CPU jogando de volta na fila caso necessário, como exemplo pode-se destacar a task que antes teria que esperar a execução de todas as outras antes dela, vai executar antes do tempo que seria feito no schedule. Enquanto na prioridade, a de maior prioridade sempre serão executadas antes, o que é bom se existe prioridade alta tendo importância, e as prioridades baixas são executadas por último, e as vezes acaba não sendo executadas.