

# Trabalho M2 - Escalonamento

Nicolas S. Renaux<sup>1</sup>, Pedro H. C. Ruthes<sup>1</sup>

<sup>1</sup>Escola politécnica – Universidade do Vale do Itajaí (UNIVALI)

{nicolas.renaux, pedro.ruthes}@edu.univali.br

**Abstract.** *This report presents the implementation of two task scheduling algorithms: Round-Robin with Priority (RR\_p) and Earliest Deadline First (EDF), as part of the Operating Systems course at Universidade do Vale do Itajaí. The project aims to enhance understanding of scheduling mechanisms in operating systems. The RR\_p algorithm uses priority levels with multiple ready queues, while EDF schedules tasks based on their deadlines. Modifications were made to provided libraries to improve functionality. The results highlight the performance of both algorithms.*

**Resumo.** *Este relatório apresenta a implementação de dois algoritmos de escalonamento de tarefas: Round-Robin com Prioridade (RR\_p) e Earliest Deadline First (EDF), como parte da disciplina de Sistemas Operacionais na Universidade do Vale do Itajaí. O projeto visa aprofundar o entendimento dos mecanismos de escalonamento em sistemas operacionais. O algoritmo RR\_p utiliza níveis de prioridade com múltiplas filas de aptos, enquanto o EDF escalona tarefas com base em seus prazos. Foram feitas modificações nas bibliotecas fornecidas para melhorar a funcionalidade. Os resultados destacam o desempenho de ambos os algoritmos.*

## 1. Introdução

O escalonamento de processos é um aspecto fundamental nos sistemas operacionais, essencial para a gestão eficiente dos recursos de CPU. Este trabalho explora a implementação e análise de dois algoritmos de escalonamento: Round-Robin com Prioridade (RR\_p) e Earliest Deadline First (EDF).

Ambos os algoritmos possuem características distintas que os tornam adequados para diferentes tipos de aplicações, desde sistemas interativos a sistemas de tempo real. Neste relatório, detalhamos a aplicação de cada algoritmo, seus resultados, e as vantagens e desvantagens de suas abordagens.

## 2. Enunciado do Projeto

Para consolidar o aprendizado sobre os escalonadores, você deverá implementar dois algoritmos de escalonadores de tarefas (tasks). Os escalonadores são o Round-Robin com prioridade (RR\_p) e EDF (Earliest Deadline First). Para essa implementação, são disponibilizados os seguintes arquivos:

- driver (.c) – implementa a função main(), a qual lê os arquivos com as informações das tasks de um arquivo de teste (fornecido), adiciona as tasks na lista (fila de aptos) e chama o escalonador. Esse arquivo já está pronto, mas pode ser completado.
- CPU (.c e .h) – esses arquivos implementam o monitor de execução, tendo como única funcionalidade exibir (via print) qual task está em execução no momento. Esse arquivo já está pronto, mas pode ser completado.
- list (.c e .h) - esses arquivos são responsáveis por implementar a estrutura de uma lista encadeada e as funções para inserção, deletar e percorrer a lista criada. Esse arquivo já está pronto, mas pode ser completado
- task (.h) – esse arquivo é responsável por descrever a estrutura da task a ser manipulada pelo escalonador (onde as informações são armazenadas ao serem lidas do arquivo). Esse arquivo já está pronto, mas pode ser completado.
- scheduler (.h) – esse arquivo é responsável por implementar as funções de adicionar as task na lista (função add()) e realizar o escalonamento (schedule()). Esse arquivo deve ser o implementado por vocês. Você irá gerar as duas versões do algoritmo de escalonamento, RR\_p e EDF em projetos diferentes.

Você poderá modificar os arquivos que já estão prontos, como o de manipulação de listas encadeada, para poder se adequar melhor, mas não pode perder a essência da implementação disponibilizada. Algumas informações sobre a implementação:

- Sobre o RR\_p, a prioridade só será levada em conta na escolha de qual task deve ser executada caso haja duas (ou mais) tasks para serem executadas no momento. Em caso de prioridades iguais, pode implementar o seu critério, como quem é a primeira da lista (por exemplo). Nesse trabalho, considere a maior prioridade como sendo 1. Além disso, é obrigatório o uso de múltiplas filas para a gerência de prioridade.

- Você deve considerar mais filas de aptos para diferentes prioridades. Acrescente duas taks para cada prioridade criada.
- A contagem de tempo (slice) pode ser implementada como desejar, como com bibliotecas ou por uma variável global compartilhada.
- Lembre-se que a lista de task (fila de aptos) deve ser mantida “viva” durante toda a execução. Sendo assim, é recomendado implementar ela em uma biblioteca (podendo ser dentro da próprio schedulers.h) e compartilhar como uma variável global.
- Novamente, você pode modificar os arquivos, principalmente o “list”, mas sem deixar a essência original deles comprometida. Porém, esse arquivo auxilia na criação de prioridade, já que funciona no modelo pilha.
- Para usar o Makefile, gere um arquivo schedule\_rr.c, schedule\_rrp.c e schedule\_fcfs.c que incluem a biblioteca schedulers.h (pode modificar o nome da biblioteca também). Caso não queira usar o Makefile, pode trabalhar com a IDE de preferência ou compilar via terminal
- Utilize um slice de no máximo 10 unidades de tempo.
- Para os algoritmos, você deverá, via uma primeira thread extra, a simulação da ocorrência do timer em hardware. Essa thread irá fazer a simulação do tempo e gerará a flag de estouro do tempo (para o slice). Além disso, para o algoritmo EDF será necessário avaliar o deadline das tasks e verificar qual das tasks está com o menor deadline.

### 3. Explicação e contexto da aplicação

O escalonamento de processos visa garantir a utilização eficiente da CPU, permitindo que múltiplas tarefas sejam executadas de maneira justa e dentro de prazos estipulados. O algoritmo Round-Robin com prioridade (RR\_p) é projetado para sistemas onde é necessário equilibrar a equidade de tempo de CPU entre processos com a capacidade de priorizar tarefas mais críticas.

Cada tarefa recebe uma fatia de tempo (quantum) para execução. Se a tarefa não for concluída dentro desse tempo, ela é colocada no final da fila de sua respectiva prioridade. Esta abordagem é ideal para sistemas interativos, onde a resposta rápida e justa é crucial.

O Earliest Deadline First (EDF) é especialmente utilizado em sistemas de tempo real, onde as tarefas devem ser concluídas antes de seus prazos (deadlines). No EDF, a tarefa com o prazo mais próximo é escolhida para execução, assegurando que as tarefas mais urgentes sejam tratadas primeiro.

Este método é eficiente para garantir que as tarefas respeitem suas restrições temporais, sendo crucial em aplicações onde o cumprimento de deadlines é crítico, como em sistemas embarcados e aplicações industriais.

Ambos os algoritmos são implementados para avaliar seu desempenho e adequação em diferentes cenários. O RR\_p, com sua abordagem justa e baseada em prioridades, é comparado ao EDF, que se concentra em atender prazos específicos. A análise considera a capacidade de cada algoritmo em gerenciar efetivamente os recursos de CPU e a adequação de cada abordagem para diferentes tipos de aplicações.

## 4. Resultados obtidos com as simulações

Os resultados da simulação dos algoritmos de escalonamento Round-Robin com Prioridade (RR\_p) e Earliest Deadline First (EDF) foram obtidos a partir de arquivos de texto fornecidos pelo professor. Esses arquivos continham as tarefas e suas respectivas informações, como nome, prioridade, tempo de burst e deadlines.

### 4.1. Algoritmo Round-Robin com prioridade

```
Tempo: 0
Task em execução = [T1] [1] [40] por 10 u.t.
Tempo: 10
Task em execução = [T7] [1] [40] por 10 u.t.
Tempo: 20
Task em execução = [T11] [1] [40] por 10 u.t.
Tempo: 30
Task em execução = [T1] [1] [30] por 10 u.t.
Tempo: 40
Task em execução = [T7] [1] [30] por 10 u.t.
Tempo: 50
Task em execução = [T11] [1] [30] por 10 u.t.
Tempo: 60
Task em execução = [T1] [1] [20] por 10 u.t.
Tempo: 70
Task em execução = [T7] [1] [20] por 10 u.t.
Tempo: 80
Task em execução = [T11] [1] [20] por 10 u.t.
Tempo: 90
Task em execução = [T1] [1] [10] por 10 u.t.
Tempo: 100
Task em execução = [T7] [1] [10] por 10 u.t.
Tempo: 110
Task em execução = [T11] [1] [10] por 10 u.t.
Tempo: 120
Task em execução = [T1] [1] [0] por 10 u.t.
Task T1 finalizada.
Tempo: 130
Task em execução = [T7] [1] [0] por 10 u.t.
Task T7 finalizada.
Tempo: 140
Task em execução = [T11] [1] [0] por 10 u.t.
Task T11 finalizada.
```

**Figura 1. Demonstração do resultado final obtido pelo algoritmo do Round-Robin com prioridade**

```
Tempo: 440
Task em execução = [T10] [4] [0] por 10 u.t.
Task T10 finalizada.
Tempo: 450
Task em execução = [T9] [5] [40] por 10 u.t.
Tempo: 460
Task em execução = [T9] [5] [30] por 10 u.t.
Tempo: 470
Task em execução = [T9] [5] [20] por 10 u.t.
Tempo: 480
Task em execução = [T9] [5] [10] por 10 u.t.
Tempo: 490
Task em execução = [T9] [5] [0] por 10 u.t.
Task T9 finalizada.
```

**Figura 2. Demonstração do resultado final obtido pelo algoritmo do Round-Robin com prioridade**

## 4.2. Algoritmo EDF (Earliest Deadline First)

```
Task em execução = [T1] [1] [10] por 10 u.t.  
Tempo: 0  
Task T1 - Burst restante: 0  
Task T1 finalizada.  
Task em execução = [T7] [1] [20] por 10 u.t.  
Tempo: 10  
Task T7 - Burst restante: 10  
Task em execução = [T11] [1] [20] por 10 u.t.  
Tempo: 20  
Task T11 - Burst restante: 10  
Task em execução = [T7] [1] [10] por 10 u.t.  
Tempo: 30  
Task T7 - Burst restante: 0  
Task T7 finalizada.  
Task em execução = [T11] [1] [10] por 10 u.t.  
Tempo: 40  
Task T11 - Burst restante: 0  
Task T11 finalizada.  
Task em execução = [T2] [2] [20] por 10 u.t.  
Tempo: 50  
Task T2 - Burst restante: 10  
Task T2 não conseguiu completar antes da deadline.  
Task em execução = [T4] [2] [10] por 10 u.t.  
Tempo: 60  
Task T4 - Burst restante: 0  
Task T4 finalizada.  
Task em execução = [T8] [3] [20] por 10 u.t.  
Tempo: 70  
Task T8 - Burst restante: 10  
Task T8 não conseguiu completar antes da deadline.  
Task em execução = [T3] [3] [20] por 10 u.t.  
Tempo: 80  
Task T3 - Burst restante: 10  
Task em execução = [T3] [3] [10] por 10 u.t.  
Tempo: 90  
Task T3 - Burst restante: 0  
Task T3 finalizada.  
Task em execução = [T5] [4] [10] por 10 u.t.  
Tempo: 100  
Task T5 - Burst restante: 0  
Task T5 finalizada.  
Task T10 não conseguiu completar antes da deadline.  
Task em execução = [T9] [5] [10] por 10 u.t.  
Tempo: 110  
Task T9 - Burst restante: 0  
Task T9 finalizada.
```

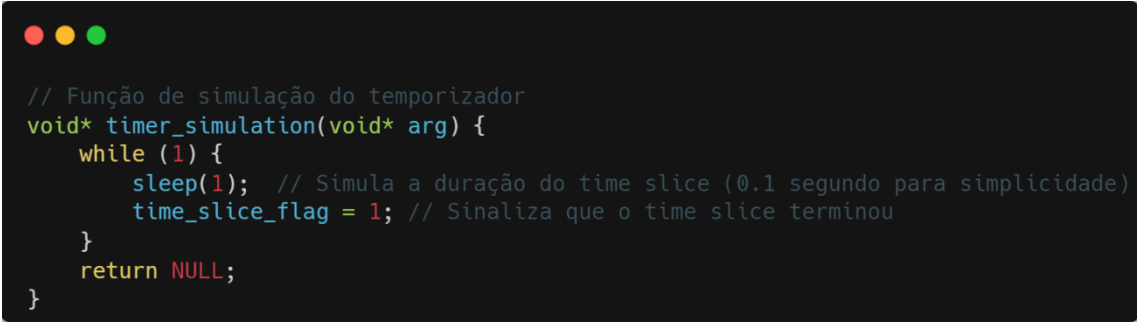
Figura 3. Demonstração do resultado final obtido pelo algoritmo do EDF

## 5. Códigos importantes

Nesta seção dos códigos importantes da aplicação, listamos os códigos essenciais para o correto funcionamento de ambos os algoritmos implementados.

### 5.1. Algoritmo Round-Robin com prioridade

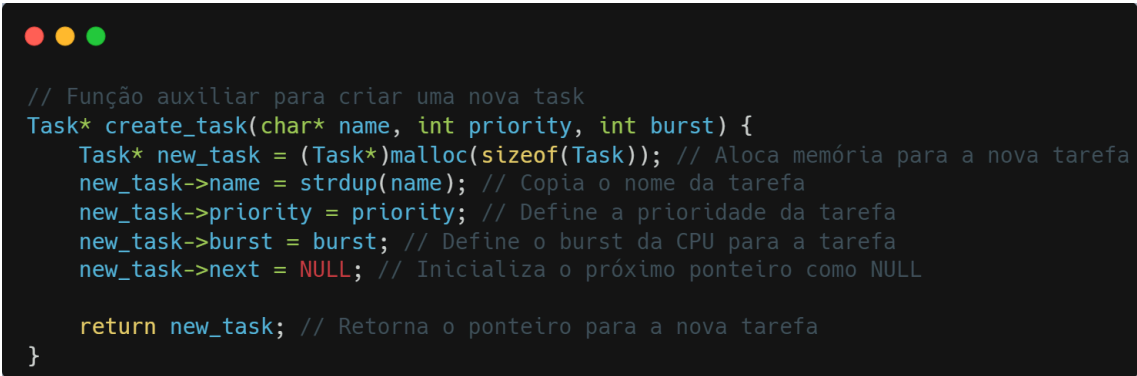
Esta função cria uma simulação de temporizador que define um flag (time\_slice\_flag) a cada segundo, sinalizando o fim de um time slice. Este flag é utilizado para controlar um timer de simulação das tarefas.



```
// Função de simulação do temporizador
void* timer_simulation(void* arg) {
    while (1) {
        sleep(1); // Simula a duração do time slice (0.1 segundo para simplicidade)
        time_slice_flag = 1; // Sinaliza que o time slice terminou
    }
    return NULL;
}
```

Figura 4. Função timer\_simulation do arquivo schedule\_rr\_p.c

Função auxiliar para a criação de novas tasks, ele aloca a memória necessária e atribui os dados de nome, prioridade e burst.



```
// Função auxiliar para criar uma nova task
Task* create_task(char* name, int priority, int burst) {
    Task* new_task = (Task*)malloc(sizeof(Task)); // Aloca memória para a nova tarefa
    new_task->name = strdup(name); // Copia o nome da tarefa
    new_task->priority = priority; // Define a prioridade da tarefa
    new_task->burst = burst; // Define o burst da CPU para a tarefa
    new_task->next = NULL; // Inicializa o próximo ponteiro como NULL

    return new_task; // Retorna o ponteiro para a nova tarefa
}
```

Figura 5. Função create\_task do arquivo schedule\_rr\_p.c

Esta função adiciona uma nova tarefa à fila correspondente à sua prioridade, mantendo a ordem de chegada dentro da mesma prioridade.

```
// Adiciona uma task à lista
void add(char* name, int priority, int burst) {
    Task* new_task = create_task(name, priority, burst); // Cria uma nova tarefa
    Task** queue = &priority_queues[priority]; // Obtem a fila correspondente à prioridade

    // Adiciona a nova tarefa à fila
    if (*queue == NULL) {
        *queue = new_task; // Se a fila está vazia, a nova tarefa é a primeira
    } else {
        Task* temp = *queue;
        while (temp->next != NULL) { // Percorre até o final da fila
            temp = temp->next;
        }
        temp->next = new_task; // Adiciona a nova tarefa ao final da fila
    }
}
```

**Figura 6. Função add do arquivo schedule\_rr\_p.c**

Esta função percorre as filas de prioridade em ordem crescente, retornando a primeira tarefa disponível para execução.

```
// Função auxiliar para obter a próxima task a ser executada
Task* get_next_task() {
    for (int i = MIN_PRIORITY; i <= MAX_PRIORITY; i++) {
        if (priority_queues[i] != NULL) {
            Task* task = priority_queues[i]; // Obtém a primeira tarefa da fila
            priority_queues[i] = task->next; // Atualiza a fila para apontar para a próxima tarefa
            task->next = NULL; // Desconecta a tarefa da fila
            return task; // Retorna a tarefa
        }
    }
    return NULL; // Retorna NULL se não houver mais tarefas
}
```

**Figura 7. Função get\_next\_task do arquivo schedule\_rr\_p.c**



Esta função coordena o escalonamento das tarefas, utilizando a função de temporização para simular o tempo e gerenciando a fila de tarefas prontas para execução.

```
// Invoca o escalonador
void schedule() {
    pthread_t timer_thread;
    pthread_create(&timer_thread, NULL, timer_simulation, NULL); // Cria um thread para a simulação do temporizador
    pthread_detach(timer_thread); // Desanexa o thread

    int time = 0; // Inicializa o tempo total de execução
    Task* task = get_next_task(); // Obtém a primeira tarefa a ser executada

    while (task != NULL) {
        while (task->burst > 0 && !time_slice_flag) {
            // Espera até que o time slice termine
            usleep(1000); // Aguarda 1 milissegundo
        }
        if (time_slice_flag) {
            time_slice_flag = 0; // Reseta o flag do time slice

            int run_time = task->burst > TIME_QUANTUM ? TIME_QUANTUM : task->burst; // Calcula o tempo de execução
            task->burst -= run_time; // Atualiza o burst da tarefa

            printf("Tempo: %d\n", time);
            run(task, run_time); // Printa a task

            time += run_time; // Atualiza o tempo total de execução

            if (task->burst > 0) {
                // Se a tarefa não terminou, adiciona de volta à fila
                add(task->name, task->priority, task->burst);
                free(task->name); // Libera o nome da tarefa
                free(task); // Libera a memória da tarefa
            } else {
                // Se a tarefa terminou, imprime a mensagem e libera a memória
                printf("Task %s finalizada.\n", task->name);
                free(task->name); // Libera o nome da tarefa
                free(task); // Libera a memória da tarefa
            }
        }

        task = get_next_task(); // Obtém a próxima tarefa a ser executada
    }
}
```

Figura 8. Função schedule do arquivo schedule\_rr\_p.c

Arquivo schedulers.h com a declaração da fila global de aptos para cada prioridade.

```
#ifndef SCHEDULERS_H
#define SCHEDULERS_H

#include "task.h"

#define MIN_PRIORITY 1 // Define a prioridade mínima como 1
#define MAX_PRIORITY 10 // Define a prioridade máxima como 10

// Declaração de uma fila global de aptos para diferentes prioridades
extern Task *priority_queues[MAX_PRIORITY];

#endif // SCHEDULERS_H
```

Figura 9. Arquivo schedulers.h

## 5.2. Algoritmo EDF (Earliest Deadline First)

Similar à função no RR\_p, esta função define um flag a cada segundo para controlar o time slice.

```
// Função de simulação de timer
void* timer_simulation(void* arg) {
    while (1) {
        sleep(1); // Simula a duração de uma parte de tempo (1 segundo para simplicidade)
        time_slice_flag = 1; // Sinaliza que uma parte de tempo terminou
    }
    return NULL;
}
```

Figura 10. Função timer\_simulation do arquivo schedule\_edf.c

Esta função cria uma nova tarefa com um atributo adicional: deadline.

```
// Função auxiliar para criar uma nova tarefa
Task* create_task(char* name, int priority, int burst, int deadline) {
    Task* new_task = (Task*)malloc(sizeof(Task));
    new_task->name = strdup(name); // Copia o nome da tarefa
    new_task->priority = priority; // Define a prioridade
    new_task->burst = burst; // Define o tempo de burst
    new_task->deadline = deadline; // Define a deadline
    new_task->next = NULL; // Inicializa o próximo ponteiro como NULL
    return new_task;
}
```

Figura 11. Função create\_task do arquivo schedule\_edf.c

Esta função adiciona uma nova tarefa à fila correspondente à sua prioridade, ordenando as tarefas por deadline.

```
// Adiciona uma tarefa à lista, inserindo-a com base na deadline
void add(char* name, int priority, int burst, int deadline) {
    Task* new_task = create_task(name, priority, burst, deadline);
    Task** queue = &priority_queues[priority]; // Obtém a fila de tarefas para a prioridade dada

    // Insere a tarefa na posição correta com base na deadline
    if (*queue == NULL || (*queue)->deadline > deadline) {
        new_task->next = *queue;
        *queue = new_task;
    } else {
        Task* temp = *queue;
        while (temp->next != NULL && temp->next->deadline <= deadline) {
            temp = temp->next;
        }
        new_task->next = temp->next;
        temp->next = new_task;
    }
}
```

Figura 12. Função add do arquivo schedule\_edf.c

Esta função, como no RR\_p, percorre as filas de prioridade para obter a próxima tarefa, mas aqui as filas estão ordenadas por deadline.

```
// Função auxiliar para obter a próxima tarefa a ser executada
Task* get_next_task() {
    for (int i = MIN_PRIORITY; i <= MAX_PRIORITY; i++) {
        if (priority_queues[i] != NULL) {
            Task* task = priority_queues[i]; // Obtém a tarefa da frente da fila
            priority_queues[i] = task->next; // Atualiza a fila removendo a tarefa obtida
            task->next = NULL;
            return task;
        }
    }
    return NULL; // Retorna NULL se não houver tarefas
}
```

Figura 13. Função `get_next_task` do arquivo `schedule_edf.c`

Esta função gerencia a execução das tarefas, verificando deadlines e reordenando as filas conforme necessário.

```
// Invoca o escalonador
void schedule() {
    pthread_t timer_thread;
    pthread_create(&timer_thread, NULL, timer_simulation, NULL); // Cria uma thread para a simulação do timer
    pthread_detach(timer_thread); // Desvincula a thread para que ela rode em segundo plano

    int time = 0; // Inicializa o tempo total de execução
    Task* task = get_next_task(); // Obtém a primeira tarefa a ser executada

    while (task != NULL) {
        while (task->burst > 0 && !time_slice_flag) {
            // Espera até que a flag da parte de tempo seja setada
            usleep(1000); // Espera por 1 milissegundo
        }
        if (time_slice_flag) {
            time_slice_flag = 0; // Reseta a flag

            int run_time = (task->burst < TIME_QUANTUM) ? task->burst : TIME_QUANTUM; // Calcula o tempo de execução
            int new_time = time + run_time; // Atualiza o tempo total de execução

            if (new_time > task->deadline) {
                // Verifica se a tarefa não pode ser completada antes da deadline
                printf("Task %s não conseguiu completar antes da deadline.\n", task->name);
                free(task->name);
                free(task);
            } else {
                run(task, run_time); // Executa a tarefa pelo tempo calculado
                printf("Tempo: %d\n", time);
                time = new_time;
                task->burst -= run_time; // Atualiza o tempo de burst restante

                printf("Task %s - Burst restante: %d\n", task->name, task->burst);

                if (task->burst <= 0) {
                    // Se a tarefa estiver completa
                    printf("Task %s finalizada.\n", task->name);
                    free(task->name);
                    free(task);
                } else {
                    // Reinsere a tarefa na fila se ainda houver tempo de burst restante
                    add(task->name, task->priority, task->burst, task->deadline);
                }
            }
        }
        task = get_next_task(); // Obtém a próxima tarefa a ser executada
    }
}
```

Figura 14. Função `schedule` do arquivo `schedule_edf.c`

## 6. Resultados obtidos com a implementação

A implementação dos algoritmos de escalonamento Round-Robin com Prioridade (RR\_p) e Earliest Deadline First (EDF) foi avaliada com base em arquivos de texto fornecidos, que continham as tarefas e suas respectivas informações.

### 6.1. Round-Robin com Prioridade (RR\_p)

No algoritmo Round-Robin com Prioridade, as tarefas foram executadas conforme suas prioridades e fatias de tempo (quantum). Os prints do terminal mostram claramente como cada tarefa foi alternada e, quando necessário, reintroduzida na fila.

Este método garantiu que todas as tarefas recebessem uma oportunidade justa de execução, o que é uma de suas maiores vantagens. Além disso, o RR\_p é simples de implementar e eficaz para evitar que qualquer tarefa sempre ocupe a CPU. No entanto, uma desvantagem notável do RR\_p é que tarefas de alta prioridade podem causar starvation em tarefas de prioridade mais baixa, especialmente em sistemas com muitas tarefas de alta prioridade.

Além disso, a escolha do tamanho do quantum pode impactar significativamente o desempenho: um quantum muito grande pode reduzir a responsividade do sistema, enquanto um quantum muito pequeno pode aumentar a sobrecarga de troca de contexto.

### 6.2. Earliest Deadline First (EDF)

Para o algoritmo Earliest Deadline First, as tarefas foram ordenadas e executadas com base em seus deadlines. Os resultados mostraram que o EDF é eficaz em atender tarefas com restrições temporais críticas, priorizando aquelas com deadlines mais próximas.

No entanto, os prints também revelaram uma limitação importante: nem todas as tarefas conseguiram concluir seu tempo de execução antes do deadline. Isso ocorreu devido à alta carga de tarefas e prazos apertados, resultando na falha de algumas tarefas em cumprir seus deadlines.

Essa característica do EDF pode ser problemática em sistemas onde a não conclusão de tarefas dentro do prazo pode levar a falhas críticas.

### 6.3. Tabelas e gráficos

Abaixo, as duas tabelas fornecem uma visão detalhada do desempenho das tarefas em diferentes algoritmos de escalonamento, e o gráfico de Gantt ilustra as deadlines associadas às tarefas da segunda tabela.

A primeira tabela apresenta dados obtidos pelo uso do algoritmo de escalonamento Round-Robin com prioridades. A tabela inclui as tarefas (Task), suas respectivas prioridades (Prioridade), o tempo de burst inicial (Tempo de burst inicial), e o tempo total de espera (Tempo total de espera). Por exemplo, a T1 com prioridade 1 tem um tempo de burst inicial de 50 e um tempo total de espera de 0, enquanto a T9 com prioridade 5 tem um tempo total de espera de 450. Esse tempo total de espera reflete o tempo que cada tarefa aguarda antes de começar a ser executada, somado ao tempo de execução das tarefas que a precederam.

A segunda tabela é baseada no algoritmo de Earliest Deadline First (EDF), que prioriza as tarefas de acordo com a proximidade de suas deadlines. As colunas incluem

as tarefas (Task), suas prioridades (Prioridade), o tempo de burst inicial (Tempo de burst inicial), a deadline (Deadline), e se a deadline foi cumprida (Deadline cumprido?). Por exemplo, a T1 com prioridade 1 e um tempo de burst inicial de 10 tem uma deadline de 50 e a cumpre, enquanto a T2, com prioridade 2 e um tempo de burst inicial de 20, não cumpre sua deadline de 60.

O gráfico de Gantt visualiza as deadlines das tarefas da segunda tabela, mostrando de forma clara e cronológica quando cada tarefa deve ser concluída. Este tipo de gráfico ajuda a visualizar a ordem de execução e a eficácia do algoritmo EDF em cumprir as deadlines. Por exemplo, se T2 aparece no gráfico e não cumpre a deadline de 60, isso ficará evidente no gráfico, ajudando a identificar possíveis melhorias ou ajustes necessários no escalonamento das tarefas.

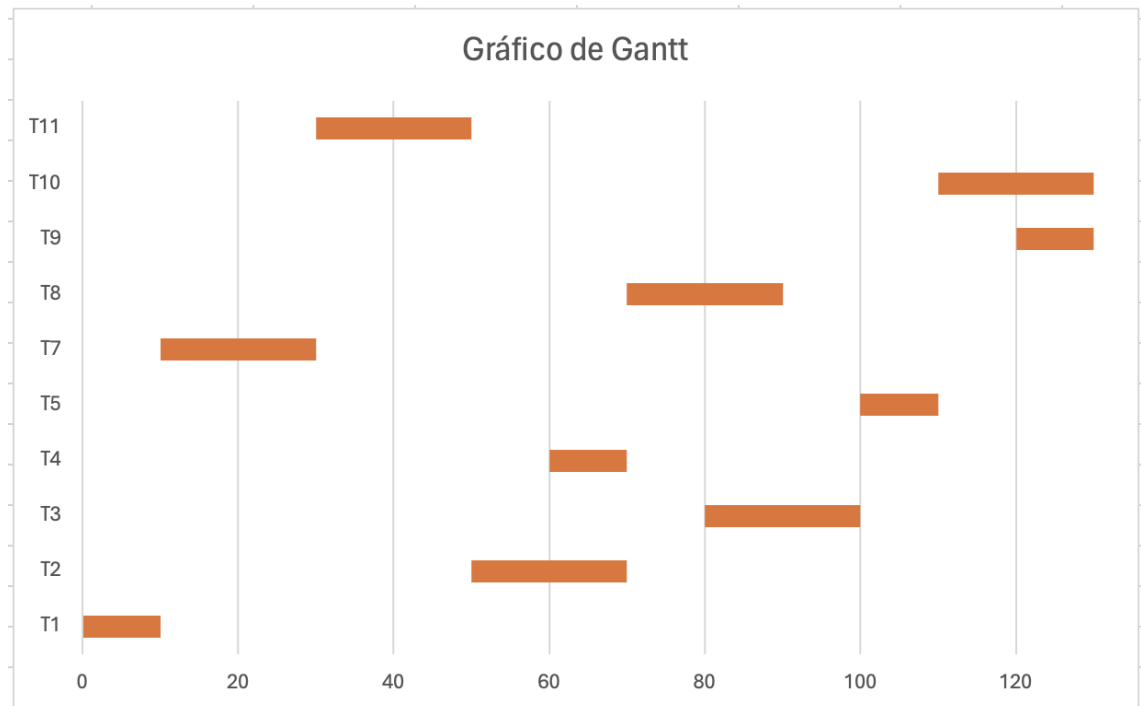
**Tabela 1. Tempo de espera por task no algoritmo de Round-Robin por prioridade**

Task	Prioridade	Tempo de burst inicial	Tempo total de espera
T1	1	50	0
T2	2	50	150
T3	3	50	250
T4	2	50	160
T5	4	50	350
T7	1	50	10
T8	3	50	260
T9	5	50	450
T10	4	50	360
T11	1	50	20

**Tabela 2. Verificação de deadline cumprido por task no algoritmo de EDF**

Task	Prioridade	Tempo de burst inicial	Deadline	Deadline cumprido?
T1	1	10	50	Sim
T2	2	20	60	Não
T3	3	20	100	Sim
T4	2	10	70	Sim
T5	4	10	110	Sim
T7	1	20	50	Sim
T8	3	20	80	Não
T9	5	10	130	Sim
T10	4	20	110	Não
T11	1	20	50	Sim

**Gráfico 1. Gráfico de Gantt demonstrando as tasks e suas deadlines**



## 7. Conclusão

As simulações realizadas destacaram as vantagens e desvantagens de ambos os algoritmos. O Round-Robin com prioridade se mostrou eficaz em distribuir de forma igualitária o tempo de CPU, mas pode levar à starvation de tarefas de baixa prioridade.

Por outro lado, o EDF demonstrou sua capacidade de gerenciar tarefas com deadlines rigorosos, embora com a desvantagem de nem sempre conseguir cumprir todos os prazos em sistemas com alta carga.

Esses resultados sugerem que a escolha do algoritmo de escalonamento deve considerar o contexto específico e os requisitos do sistema em questão. Para sistemas onde a equidade é crucial, o RR\_p pode ser mais adequado. Já em sistemas onde o cumprimento de deadlines é essencial, o EDF pode ser preferível, apesar de suas limitações.

## Referências

VIEL, Felipe. SO-Codes. Disponível em: <<https://github.com/VielF/SO-Codes/>>. Acesso em: 24 maio de 2024.

VIEL, Felipe. Aula 7 - Escalonamento. Disponível em: <<https://private-zinc-3e1.notion.site/Aula-7-Escalonamento-31736ced893046228a857375d1f9dfab>>. Acesso em: 24 maio de 2024.