UNIVERSIDADE DO VALE DE ITAJAÍ

Curso de Engenharia de Computação

Disciplina de Sistemas Operacionais

Orientador Felipe Viel

PEDRO JOSÉ GARCIA

THIAGO ZIPPER MELATO

SISTEMAS OPERACIONAIS - AVALIAÇÃO M2

ESCALONAMENTO

Itajaí

2025

SUMÁRIO

PROJETO

"Implementação de diferentes escalonadores".

INTRODUÇÃO

Neste trabalho, o objetivo foi implementar e demonstrar diversos tipos de escalonadores, buscando compará-los em relação ao tempo de espera, resposta e conclusão de tarefas, além da quantidade de trocas de contexto, esses escalonadores são: *Round Robin* (RR), *Round Robin* de prioridade, *Earliest Deadline First* (EDF) e *Round Robin* de prioridade com envelhecimento (*aging*).

Para esse projeto foram disponibilizados diversos arquivos pelo Professor Felipe Viel em seu repositório do *GitHub*, esses arquivos fornecem uma estrutura para seguir no momento de implementar os escalonadores, podendo serem alterados desde que haja motivo para isso.

ENUNCIADO

Para consolidar o aprendizado sobre os escalonadores, você deverá implementar dois algoritmos de escalonadores de tarefas (taks) estudos em aula. Os escalonadores são o Round-Robin (RR), o Round-Robin com prioridade (RR_p), EDF (Earliest Deadline First - link de explicação) e Prioridade aplicando o conceito de Aging (envelhecimento). Para essa implementação, são disponibilizados os seguintes arquivos (Link Github):

- 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 e RR_p, em projetos diferentes, além do EDF e Prioridade com Aging.

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), aplicando o mesmo conceito de RR clássico. Nesse trabalho, considere a maior prioridade como sendo 1.

- Você deve considerar mais filas de aptos para diferentes prioridades. Acrescente duas taks para cada prioridade criada. Não serão aceitos trabalhos com uma única fila reorganizada com algoritmos de ordenação.

- 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, schedule_edf.c e schedule_pa.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.

- Você deverá, via uma thread, simular o funcionamento de um timer em hardware. Esse thread irá fazer a simulação do tempo e gerará a flag de estouro do tempo (para o slice). Além disso, para o algoritmo EDF (link de explicação) será necessário avaliar o deadline das tasks e verificar qual das tasks está com o menor deadline. Logo, para esse algoritmo, você deverá manter o registro de deadline e timestamp de início para saber se ela aumenta ou não de prioridade. O mesmo acontece com o algoritmo de Prioridade por Aging, porém você terá que definir o qual é limiar de envelhecimento, ou seja, qual é a quantidade de tempo que a tarefa passou sem executar que faz ela aumentar de prioridade.

IMPLEMENTAÇÃO

Para este trabalho foram utilizados os códigos na linguagem C, disponibilizados pelo professor Felipe Viel para as partes de leitura de tarefas, estruturas de dados e exibição de informações. O foco do desenvolvimento se voltou então à implementação dos algoritmos de escalonamento, além de ajustes aos arquivos já fornecidos.

A adição de tarefas à fila é feita com a instanciação de *structs Task*, aos quais são atribuídos os parâmetros extraídos pelo *driver*, mesmo que estes não sejam usados pelo escalonador em particular. As tarefas são adicionadas a listas encadeadas, mantidas como variáveis globais. Uma variável global também é usada para rastreio dos identificadores de tarefa.

```
 void add(char *name, int priority, int burst, int deadline) {
   Task *newTask = (Task *)malloc(sizeof(Task));
   if (newTask == NULL) {
      fprintf(stderr, "Error: Failed to allocate memory for task [%s].\n",
name);
      return;
   }

   newTask->name = strdup(name);
   if (newTask->name == NULL) {
      fprintf(stderr, "Error: Failed to assign name to task [%s].\n",
name);
      free(newTask);
      return;
   }

   newTask->tid = nextTid;
   newTask->priority = priority;
   newTask->burst = burst;
   newTask->deadline = deadline;

   nextTid++;

   insert(&taskList, newTask);
}
```

Para os escalonadores que fazem uso de prioridade, as tarefas são inseridas em múltiplas listas de acordo com seu grau de prioridade, o número total de listas é definido pela quantidade de graus disponíveis. É feita a validação da prioridade fornecida para confirmar se está dentro da faixa prevista.

```
struct node *taskLists[MAX_PRIORITY - MIN_PRIORITY + 1] = {NULL};
```

```
   if (priority < MIN_PRIORITY || priority > MAX_PRIORITY) {
      fprintf(stderr, "Error: Priority [%d] for the task [%s] out of range
(%d-%d).\n",
            priority, name, MIN_PRIORITY, MAX_PRIORITY);
```

```
        return;
    }
```

As funções de escalonamento mantêm-se ativas enquanto houver tarefas a serem processadas. O escalonador RR clássico obtém a tarefa no final da lista e cria uma *thread* para executá-la pela duração do *quantum* (nessa aplicação, 10 unidades de tempo). Se ainda houver tempo de *burst* restante após a execução, a tarefa é reinserida no início da lista, e o escalonador busca uma nova tarefa no final. O trecho abaixo mostra a função responsável pelo escalonamento do RR.

```
void schedule(){
    pthread_t timer_tid;
    ThreadArgs args;

    while (taskList != NULL) {
        // get the task from the end of the list
        struct node *listEnd = end(taskList);
        Task *task = listEnd->task;

        // remove the task from the list and run it
        delete(&taskList, task);

        args.task = task;
        args.slice = QUANTUM;
        if (pthread_create(&timer_tid, NULL, run, &args) != 0) {
        perror("Error: Failed to create timer thread.");
        return;
        }
        pthread_join(timer_tid, NULL); // wait for the timer thread to finish

        if (task->burst > 0) {
        // task is not completed, reinsert it at the head of the list
        insert(&taskList, task);
        }
        else {
        // task is completed, free its resources
        free(task->name);
        free(task);
        }
    }
}
```

O escalonador RR de prioridade percorre as diferentes listas e seleciona a de maior prioridade que possui tarefas a serem processadas.

```
    while (1) {
        struct node **taskList;
        int hasTasks = 0;
        // get the task from the highest-priority list with tasks
        for (int i = 0; i <= (MAX_PRIORITY - MIN_PRIORITY); i++) {
        if (taskLists[i] != NULL) {
              taskList = &taskLists[i];
              hasTasks = 1;
```

```
            break;
      }
      }
      if (!hasTasks) {
      return; // no tasks to run, exit the scheduler
      }
```

Já o escalonador RR de prioridade com *aging* trabalha de forma muito parecida, porém utilizando uma variável extra importante que fica na *struct* da *task*: *time_since_last_run*, essa variável armazena quanto tempo faz desde que a *task* recebeu tempo de processador pela última vez, caso a esse tempo ultrapasse o threshold definido, a *task* recebe uma promoção de prioridade.

```
if (other_task->time_since_last_run >= AGING_THRESHOLD) {
     if (other_task->priority > MIN_PRIORITY) {

    delete(&taskLists[p_idx], other_task);
    other_task->priority--;
    other_task->time_since_last_run = 0;

    int new_priority_idx = other_task->priority - MIN_PRIORITY;

    insert(&taskLists[new_priority_idx], other_task);
    }
}
```

O escalonador *Earliest Deadline First* possui uma implementação um pouco diferente, ele exigiu a adição de um novo parâmetro na função *add()*, o "*deadline*". Esse parâmetro é utilizado para definir a prioridade de uma tarefa, quanto menor o *deadline*, maior se torna a prioridade, e claro, esse *deadline* diminui com o passar do tempo.

```
while (temp != NULL) {
   if (temp->task->deadline <= earliestNode->task->deadline) {
      earliestNode = temp;
   }
   temp = temp->next;
}

Task *task = earliestNode->task;
```

As threads que executam as tarefas subtraem o *slice* do tempo de burst da tarefa, mas se o *burst* for menor que o *slice* ou se nenhum *slice* for fornecido (no caso do EDF), não ocorre a preempção e a tarefa é executada até o fim.

```
void *run(void *args) {
      // cast arguments
      ThreadArgs *threadArgs = (ThreadArgs *)args;
```

```
	Task *task = threadArgs->task;
	int slice = threadArgs->slice;

	// if the slice is larger than the burst time or no valid burst is
provided, run for the burst time
	if (task->burst < slice || slice < 0) {
	slice = task->burst;
	}

	printf("Running task = [%s] [%d] [%d] [%d] for %d
units.\n",task->name, task->priority, task->burst, task->deadline, slice);

	task->burst -= slice; // reduce the burst time by the time slice

	return NULL;
}
```

RESULTADOS SIMULADOS

A tabela de tarefas (*schedule.txt*), baseada no arquivo *edf-schedule_pri.txt* disponibilizado pelo professor, foi utilizada para testar os quatro escalonadores.

Tabela 1 - Tarefas a serem executadas.

Nome	Prioridade	Tempo de *burst*	*Deadline*
T1	1	10	50
T2	2	20	60
T3	3	20	100
T4	2	5	70
T5	4	10	110
T6	1	20	50
T7	3	20	80
T8	5	10	130
T9	4	25	110
T10	5	15	50

Fonte: Elaborada pelos autores, com base em materiais do professor Felipe Viel.

A sequência de execução de tarefas, para o escalonador RR com um *quantum* de 10 unidades de tempo:

Tabela 2 - Sequência de execução do escalonador RR.

Tempo	Nome	*Burst restante*
0-10	T1	0
10-20	T2	10
20-30	T3	10
30-35	T4	0
35-45	T5	0
45-55	T6	10
55-65	T7	10
65-75	T8	0
75-85	T9	15

85-95	T10	5
95-105	T2	0
105-115	T3	0
115-125	T6	0
125-135	T7	0
135-145	T9	5
145-150	T10	0
150-155	T9	0

Fonte: Elaborada pelos autores.

Para o RR com prioridade, mantendo o *quantum* de 10 unidades de tempo:

Tabela 3 - Sequência de execução do escalonador RR com prioridade.

Tempo	Nome	*Burst restante*
0-10	T1	0
10-30	T6	0
30-40	T2	10
40-45	T4	0
45-55	T2	0
55-65	T3	10
65-75	T7	10
75-85	T3	0
85-95	T7	0
95-105	T5	0
105-130	T9	0
130-140	T8	0
140-155	T10	0

Fonte: Elaborada pelos autores.

Para o escalonador EDF:

Tabela 4 - Sequência de execução do escalonador EDF.

Tempo	Nome	Tempo de estouro do *deadline*
0-10	T1	-
10-30	T6	-
30-45	T10	-
45-65	T2	5
65-70	T4	-
70-90	T7	10
90-110	T3	10
110-120	T5	10
120-145	T9	35
145-155	T8	25

Fonte: Elaborada pelos autores.

Por fim, a sequência do escalonador RR com prioridade e *aging* é a seguinte:

Tabela 4 - Sequência de execução do escalonador RR com prioridade e *aging*.

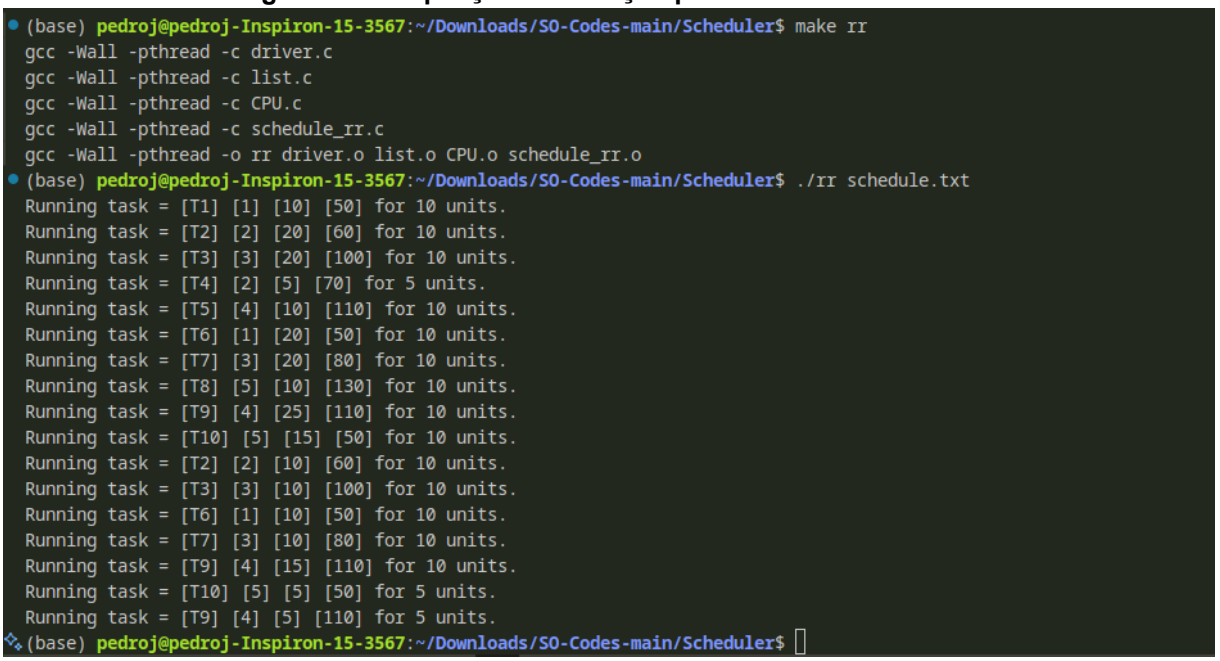
Tempo	Nome
0-20	T6
20-30	T1
30-35	T4
35-55	T2
55-75	T3
75-95	T7
95-105	T5
105-130	T9
130-145	T10
145-155	T8

Fonte: Elaborada pelos autores.

RESULTADOS DA IMPLEMENTAÇÃO

O projeto foi compilado e executado em quatro diferentes configurações para cada escalonador. O resultado para o RR clássico é mostrado a seguir:

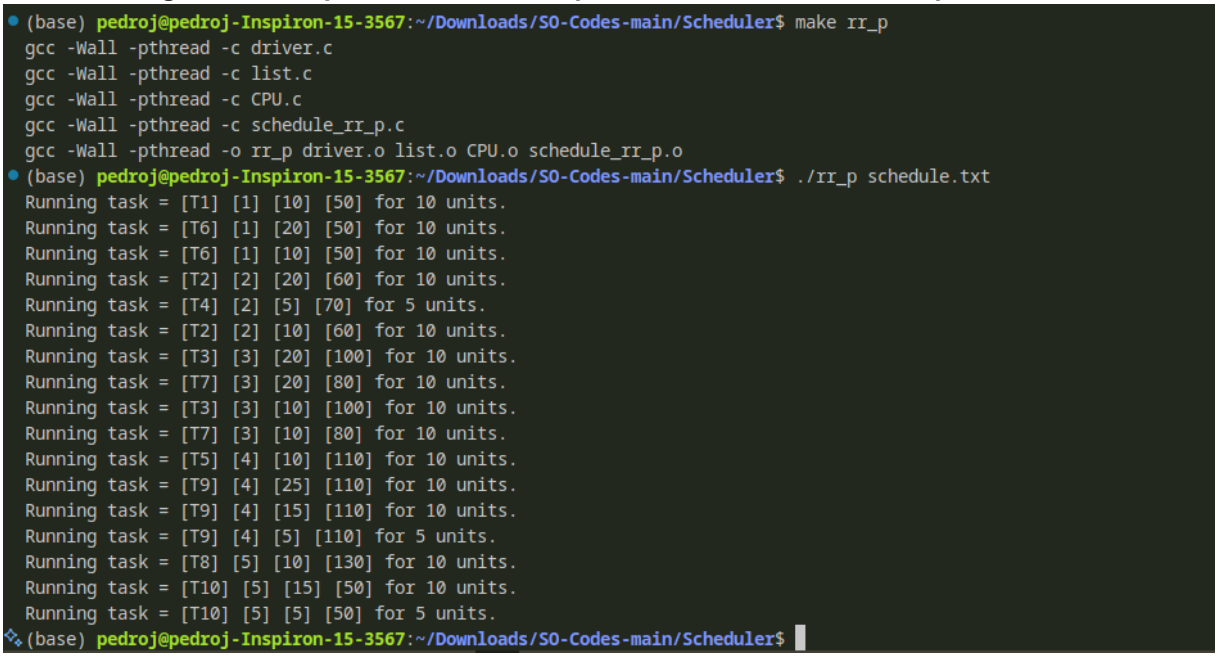
Figura 1 - Compilação e execução para o escalonador RR.



Fonte: Visual Studio Code.

Escalonador RR com prioridade:

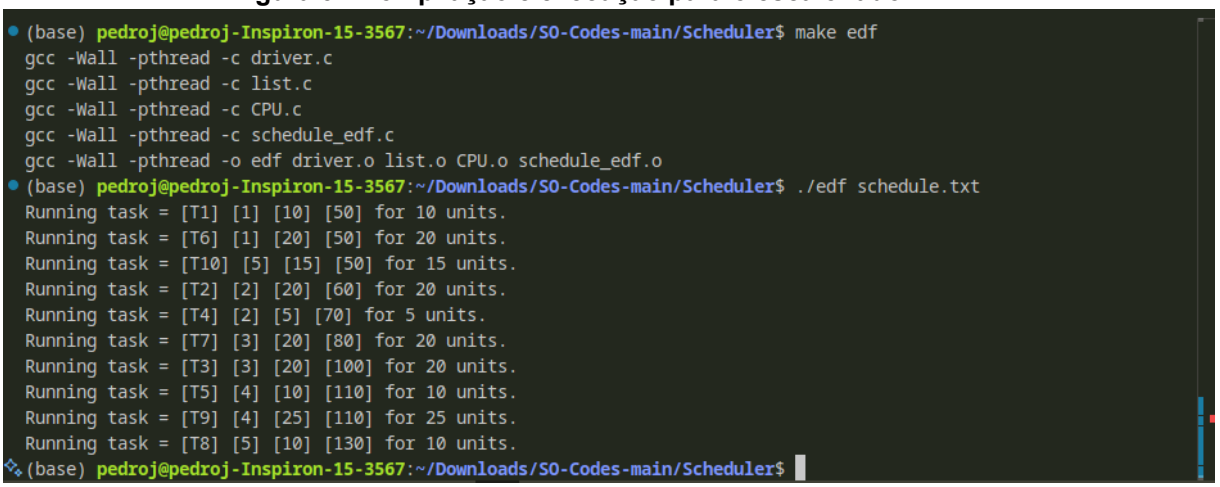
Figura 2 - Compilação e execução para o escalonador RR com prioridade.



Fonte: Visual Studio Code.

Escalonador EDF:

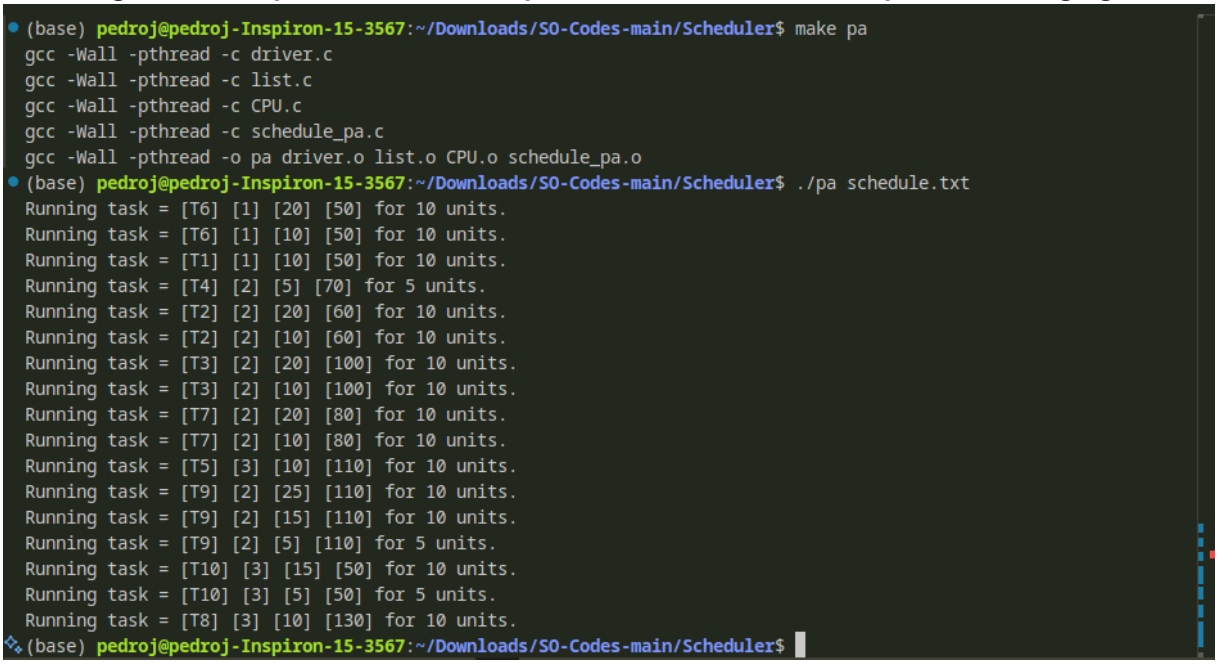
Figura 3 - Compilação e execução para o escalonador EDF.



Fonte: Visual Studio Code.

Escalonador RR com prioridade e *aging*:

Figura 4 - Compilação e execução para o escalonador RR com prioridade e *aging*.



Fonte: Visual Studio Code.

DISCUSSÃO

Este projeto foi importante de algumas formas diferentes, um dos pontos foi a utilização de códigos prontos para servir de base para a implementação, algo que normalmente não é explorado em outras disciplinas, mas é extremamente recorrente no mercado de trabalho.

Fora esta característica também foi interessante analisar os diferentes tipos de escalonadores, conhecendo os pontos fortes e fracos de cada um, facilitando na escolha de um método para o caso de ser necessário implementar um escalonador em alguma aplicação real ou também na programação de um sistema para fazer um uso correto de prioridades e melhorar o seu desempenho.

REFERÊNCIAS

GARCIA, Pedro José; MELATO, Thiago Zipper. Scheduler [repositório GitHub]. 2025. Disponível em: https://github.com/pedro-fixingstuff/Scheduler. Acesso em: 27 maio 2025.

GEEKSFORGEEKS. *Earliest Deadline First (EDF) CPU Scheduling Algorithm.* Disponível em: https://www.geeksforgeeks.org/earliest-deadline-first-edf-cpu-scheduling-algorithm/. Acesso em: 27 maio 2025.

VIEL, Felipe. SO-Codes [repositório GitHub]. 2025. Disponível em: https://github.com/VielF/SO-Codes/tree/main/Scheduler. Acesso em: 13 abr. 2025.

VIEL, Felipe. *Aula 7 – Escalonamento*. 2025. Disponível em: https://private-zinc-3e1.notion.site/Aula-7-Escalonamento-31736ced893046228a857375d1f9dfab. Acesso em: 13 abr. 2025.