UNIPAR

IAN KALLEL ARROYO LÓS

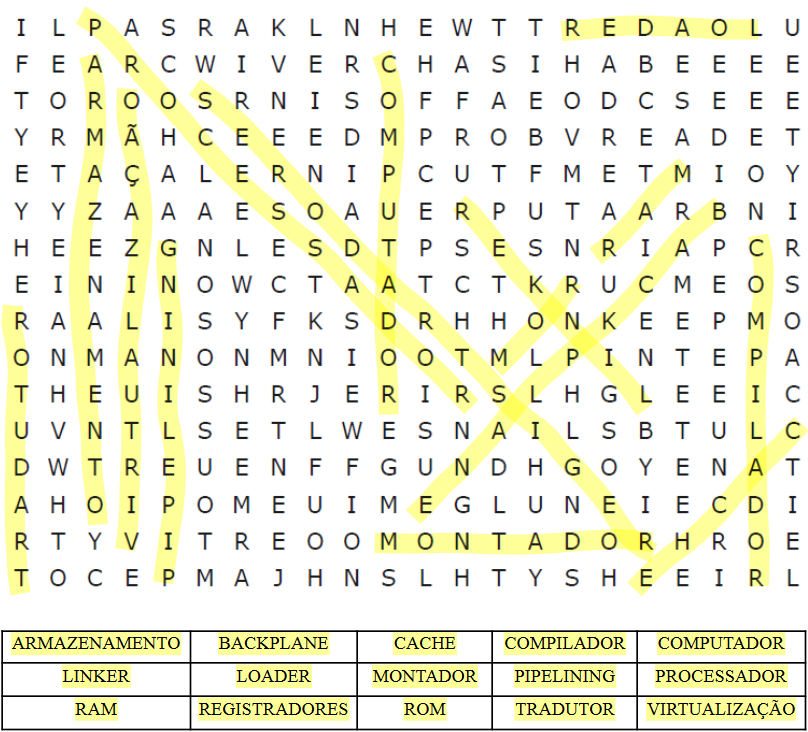
**INTEGRAÇÃO ENTRE HARDWARE E SOFTWARE**: Otimização Recursos Computacionais em Sistemas Reais

Toledo – PR

2025

Atividade 1:





Atividade 2:

Você pode acessar os códigos completos no GitHub neste link: <https://github.com/KallelGaNewk/unipar/tree/39f538d6800107ad68b8cd3b271612812c0fcffa/Arquitetura%20de%20Computadores%20e%20Sistemas%20Operacionais/APO%201>

Simulações de Threads usando Python, o primeiro algoritmo que implementei foi First-In, First-Out (FIFO), pode ver a implementação abaixo.

# fifo.py

# Implementação do escalonamento FIFO (First In, First Out)

import time

# Definição dos tempos de execução (em milissegundos)

tempos\_de\_execucao = [5, 10, 3]

class Processo:

    def \_\_init\_\_(self, id, tempo\_total):

        self.id = id

        self.tempo\_total = tempo\_total

    def executar(self):

        print(f"Processo {self.id} iniciando execução por {self.tempo\_total}ms...")

        time.sleep(self.tempo\_total / 1000)  # Converte ms para segundos

        print(f"Processo {self.id} finalizado após {self.tempo\_total}ms de execução!")

        return self.tempo\_total

def escalonador\_fifo(processos):

    print("Iniciando simulação FIFO/FCFS")

    tempo\_total\_execucao = 0

    tempo\_espera\_total = 0

    for processo in processos:

        print(f"Processo {processo.id} esperou {tempo\_total\_execucao}ms")

        tempo\_espera\_total += tempo\_total\_execucao

        # Executa o processo do início ao fim sem interrupção

        tempo\_execucao = processo.executar()

        tempo\_total\_execucao += tempo\_execucao

    tempo\_espera\_medio = tempo\_espera\_total / len(processos)

    print("\nEstatísticas da simulação:")

    print(f"Tempo total de execução: {tempo\_total\_execucao}ms")

    print(f"Tempo médio de espera: {tempo\_espera\_medio}ms")

# Criar processos

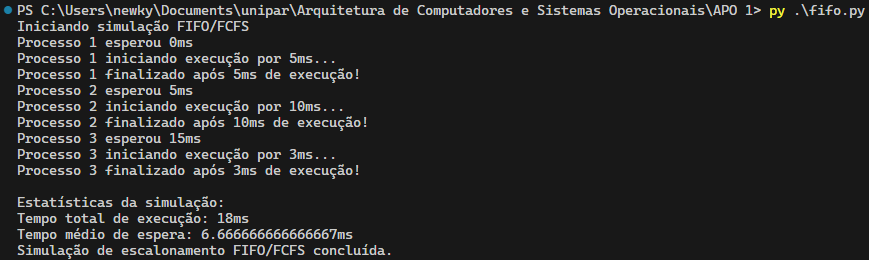
processos = [Processo(i + 1, tempo) for i, tempo in enumerate(tempos\_de\_execucao)]

# Iniciar o escalonador FIFO

escalonador\_fifo(processos)

print("Simulação de escalonamento FIFO/FCFS concluída.")

Saída do fifo.py no terminal:



O jeito que esse algoritmo funciona, a primeira thread que é criada é a que sai por primeiro, e a segunda thread sai por segundo, e assim por diante. É a mais simples de implementar, já que já quando criar, executamos ela. Funciona como uma fila de banco, as primeiras que chegam, são as primeiras que são atendidas.

O próximo algoritmo que implementei, foi de prioridade, a implementação está abaixo.

# prioridade.py

# Implementação do escalonamento por prioridade de threads em Python

import time

# Cria uma Classe para representar uma Thread

class Processo:

    def \_\_init\_\_(self, thread\_id, tempo, prioridade):

        # Definimos aqui os atributos da Thread, para que possamos acessá-los posteriormente

        self.thread\_id = thread\_id

        self.tempo = tempo

        self.prioridade = prioridade

    def tarefa(self):

        print(f"Thread {self.thread\_id} iniciada, executando por {self.tempo}ms...")

        time.sleep(self.tempo / 1000) # Simula o tempo de execução da thread

        print(f"Thread {self.thread\_id} finalizada!")

        return self.tempo

# Tempo de execução para cada thread (em milissegundos)

tempos\_de\_execucao = [5, 10, 3]

# Prioridade de cada thread (quanto menor, maior a prioridade)

prioridades = [1, 0, 2]

# Cria as threads com base nos tempos de execução e prioridades

threads = [Processo(i+1, tempo, prioridades[i]) for i, tempo in enumerate(tempos\_de\_execucao)]

# Ordena as threads por prioridade

threads.sort(key=lambda x: x.prioridade, reverse=True)

total\_time = 0

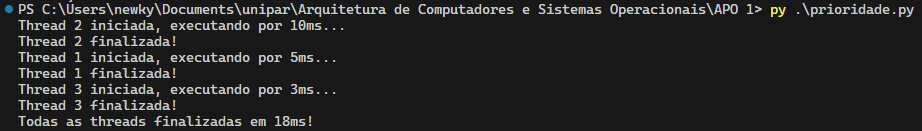
# Inicia as threads

for thread in threads:

    total\_time += thread.tarefa()

print(f"Todas as threads finalizadas em {total\_time}ms!")

Saída de prioridade.py no terminal:



A diferença entre o algoritmo de escalonamento por Prioridade e o algoritmo First-In-First-Out está no fato de que, no modelo baseado em Prioridade, os processos que requerem execução imediata são classificados com níveis de prioridade elevados, como por exemplo, o valor 0 neste contexto específico.

Utilizando a analogia da fila bancária, este mecanismo se parece com um sistema de atendimento preferencial, no qual clientes em condições especiais, como idosos ou pessoas com deficiência, recebem o atendimento imediato, independente na sequência de chegada.

A próxima implementação, é do Round Robin, a seguir.

# round-robin.py

# Implementação do escalonamento Round Robin

import time

# Queue é tipo um array, mas com operações específicas para filas

from queue import Queue

# Definição dos tempos de execução (em milissegundos)

tempos\_de\_execucao = [5, 10, 3]

quantum = 1  # Quantum de tempo em ms

class Processo:

    def \_\_init\_\_(self, id, tempo\_total):

        self.id = id

        self.tempo\_restante = tempo\_total

        self.tempo\_total = tempo\_total

    def executar(self, tempo):

        # Checa se o tempo a ser executado é menor que o tempo restante

        # Se for, executa pelo tempo restante, senão, executa pelo tempo do quantum

        tempo\_executado = min(tempo, self.tempo\_restante)

        time.sleep(tempo\_executado / 1000)

        return tempo\_executado

    def finalizado(self):

        return self.tempo\_restante <= 0

def escalonador\_round\_robin(processos, quantum):

    fila = Queue()

    # Adiciona todos os processos à fila

    for processo in processos:

        fila.put(processo)

    # Executa até que todos os processos sejam finalizados

    while not fila.empty():

        processo\_atual = fila.get()

        # Executa o processo pelo tempo do quantum ou pelo tempo restante

        tempo\_executado = processo\_atual.executar(quantum)

        print(

            f"Processo {processo\_atual.id} executando por {tempo\_executado}ms... (Restante: {processo\_atual.tempo\_restante}ms)"

        )

        processo\_atual.tempo\_restante -= tempo\_executado

        # Verifica se o processo foi finalizado

        if not processo\_atual.finalizado():

            # Se não foi finalizado, coloca de volta na fila

            fila.put(processo\_atual)

        else:

            print(

                f"Processo {processo\_atual.id} finalizado após {processo\_atual.tempo\_total}ms de execução total!"

            )

# Criar processos

# Esse syntax parece estranho, mas é equivalente a dar append em uma lista dentro do for loop

processos = [Processo(i + 1, tempo) for i, tempo in enumerate(tempos\_de\_execucao)]

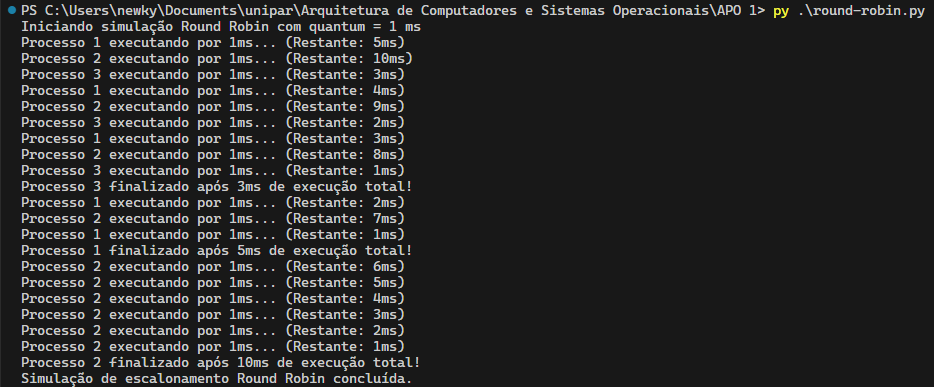
# Iniciar o escalonador Round Robin

print("Iniciando simulação Round Robin com quantum =", quantum, "ms")

escalonador\_round\_robin(processos, quantum)

print("Simulação de escalonamento Round Robin concluída.")

Saída de round-robin.py no terminal:



A diferença para as outras duas, seria que o CPU processa cada Thread em fatias de tempo, nesse código ele foi definido na variável ‘quantum’. Cada processo pode executar por esse tempo por vez, e depois é trocado para o próximo na fila, e assim por diante, até concluir o que a Thread precisava fazer. Pode ser comparado a compartilhar um único videogame para três crianças em uma tarde, todas jogam em rodadas por um tempo, salva o jogo que ela estava jogando, e a outra abre outro jogo, e assim por diante.

Comparando as três implementações, usando 3 processos de 10ms, 5ms, e 3ms cada, todas terminam em 18ms, o diferencial é como cada uma executa, dando prioridades para processos críticos, ou fazendo todas executarem ao mesmo tempo.

No First-In, First-Out, os outros processos precisam esperar o anterior, como ali no código, processo 2 precisa esperar o processo 1, que demora 5ms, e depois disso processor 2 pode executar, demorando 10ms, e por fim, processo 3 precisa esperar pelo processo 1 e 2, totalizando 15ms de espera.

Agora no Prioridade, vamos imaginar que processo 2 é de alta prioridade, ele é executado primeiro por 10ms, e após isso processo 1 é executado por 5ms, totalizando 15ms de espera para o processo 3, já que não era tão importante.

E no Round Robin, todos já começam executando um pouco de cada vez, processo 3 termina primeiro após 3ms seguido pelo processo 1, que demorou 5ms. Analisando, a resposta do código, a diferença entre o processo 3 e o 1, é apenas 3ms, diferente do resto.