Relatório de Sistemas Operacionais: Gerenciamento de Processos - Escalonamento

Bruno Costa Souto[1], Felipe Augusto Araújo da Cunha[2]

[1]Universidade Federal do Rio Grande do Norte (UFRN)

[2]Centro de Ensino Superior do Seridó (CERES).

Professor João Batista Borges Neto

Resumo. *Este trabalho teve como objetivo a implementação, execução e análise comparativa de diferentes algoritmos de escalonamento de processos utilizando um simulador em linguagem C, com o apoio de scripts em Python para geração e visualização dos resultados. Foram analisados os algoritmos FIFO, SJF, LJF, PRIO_STATIC, PRIO_DYNAMIC e PRIO_DYNAMIC_QUANTUM. Os resultados obtidos permitiram compreender melhor o funcionamento e os impactos de cada política de escalonamento, reforçando a importância de escolher o algoritmo mais adequado ao perfil de carga do sistema.*

1. Introdução

Em um sistema operacional, o escalonador é o componente responsável por selecionar um processo da fila de prontos para ser executado pelo processador. Dessa forma, é possível organizar a execução dos processos de maneira justa, maximizando a utilização do processador e o desempenho geral do sistema, ao mesmo tempo em que se busca minimizar o tempo de execução, o tempo de espera e o tempo de resposta dos processos.

O escalonador pode ser classificado em três níveis:

- Longo prazo: entra em ação quando um novo processo é criado, decidindo quando ele poderá entrar na fila de prontos.

- Médio prazo: está ligado à gerência de memória e atua no mecanismo de *swapping* (troca de processos entre a memória principal e secundária).

- Curto prazo: é o escalonador que decide qual processo utilizará o processador em um dado instante de tempo.

Com base nesses conceitos, surgem os algoritmos de escalonamento, que implementam a lógica necessária para determinar qual processo será executado. Esses algoritmos buscam bons tempos médios, sem favorecer exclusivamente um único critério, visando manter uma variação equilibrada entre os tempos de execução. O

objetivo deste relatório é implementar e analisar comparativamente diferentes algoritmos de escalonamento de processos, com base em seus tempos médios de espera.

2. Desenvolvimento

Para o desenvolvimento dos escalonadores, foi utilizado um Simulador de Escalonador, fornecido como base para a construção dos algoritmos. O algoritmo *First-In First-Out* (FIFO) já veio implementado em linguagem C, acompanhado de diversas bibliotecas auxiliares. Os demais algoritmos, incluindo o FIFO, foram descritos em um documento em formato PDF, que tinha como objetivo orientar a implementação dos algoritmos e posteriormente a elaboração deste relatório. Com base nessas orientações, os algoritmos foram desenvolvidos também em linguagem C, utilizando a estrutura do simulador. Para a geração dos gráficos, utilizou-se a linguagem Python, com as bibliotecas matplotlib e pandas, que processaram os resultados dos escalonadores a partir de arquivos .txt.

Foram implementados os seguintes algoritmos:

- Preemptivos:
- *First-In First-Out* (FIFO) → **Já fornecido pelo simulador**
- *Shortest Job First* (SJF)
- *Longest Job First* (LJF) → **Algoritmo teste (Inverso do SJF)**
- Não Preemptivos:
- Prioridade Estática (PRIO)
- Prioridade Dinâmica (PRIO_DYNAMIC)
- Prioridade Estática com Quantum (PRIO_STATIC_QUANTUM)

A partir da execução desses algoritmos, foram obtidos os tempos médios de espera (TME) para diferentes quantidades de processos. As execuções seguiram o seguinte procedimento:

- Para cada algoritmo, foram realizados 10 testes para cada quantidade de processos, variando de 10 a 100, com incrementos de 10 processos.
- Isso resultou em um total de 100 execuções por algoritmo.

Com os dados coletados, foram calculados:

- O Valor Médio do TME para cada par (algoritmo x quantidade de processos).
- A Variância do TME entre as amostras desse mesmo par.

Esses valores foram utilizados para a construção de gráficos, onde:

- A linha azul representa o Valor Médio do TME.
- A linha vermelha representa a Variância.

A descrição dos algoritmos nos tópicos seguintes são de acordo com o PDF de orientação, citado anteriormente, e também slides de explicação do professor João Borges.

2.1. FIFO

O algoritmo First-In First-Out (FIFO) já foi implementado na base fornecida pelo simulador. Ele funciona executando os processos na ordem em que chegam à fila de prontos. Cada processo é executado até que libere explicitamente o processador, realize uma chamada de sistema (ficando bloqueado) ou termine sua execução. Após a execução ou liberação, o próximo processo da fila é selecionado. Não há preempção nesse algoritmo, ou seja, o processo em execução permanece utilizando o processador até que ele próprio o libere. A seguir, é apresentado o gráfico com os resultados obtidos para o algoritmo FIFO, mostrando o tempo médio de espera (linha azul) e sua variância (linha vermelha) conforme o número de processos aumenta.

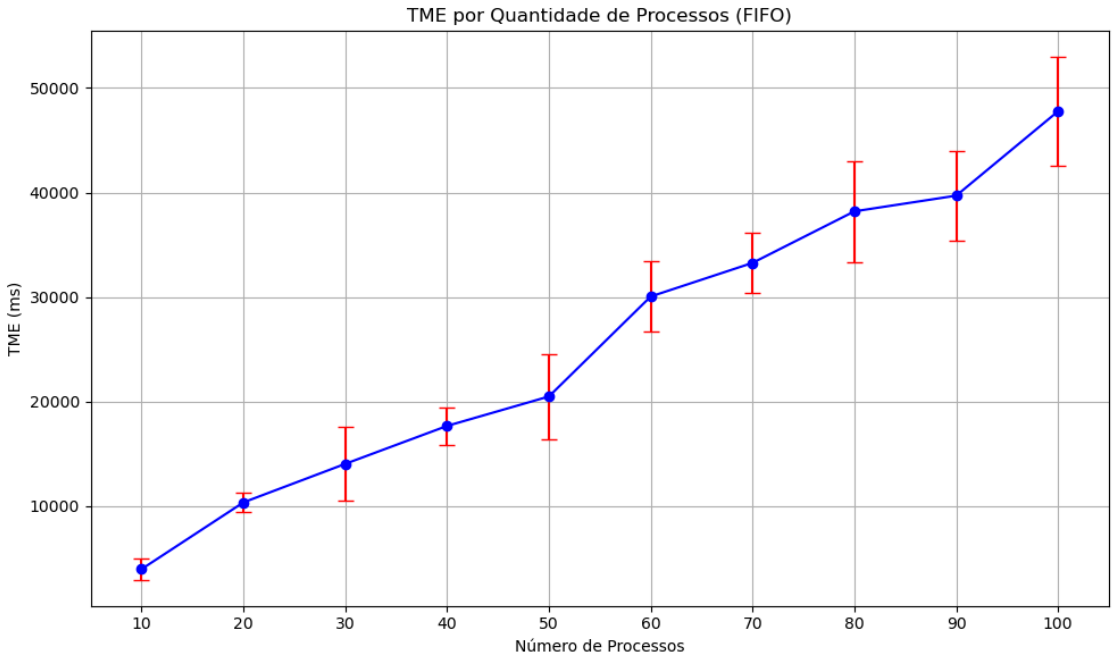

Gráfico 1 - Algoritmo FIFO

Com base no gráfico é notável perceber que, por ser um algoritmo não preemptivo e baseado apenas na ordem de chegada, o tempo médio de espera cresce proporcionalmente com a carga de processos, o que pode ser observado nos dados apresentados.

2.2. SJF

Esse algoritmo funciona selecionando da fila de aptos o processo com menor tempo restante (*remaining_time*) para sua conclusão. Após o término ou liberação do processo, o escalonador seleciona outros processos da fila de aptos que tenho o menor tempo restante, e por assim em diante até a finalização do programa. A seguir, é apresentado o gráfico com os resultados obtidos para o algoritmo SJF, mostrando o tempo médio de espera (linha azul) e sua variância (linha vermelha) conforme o número de processos aumenta.

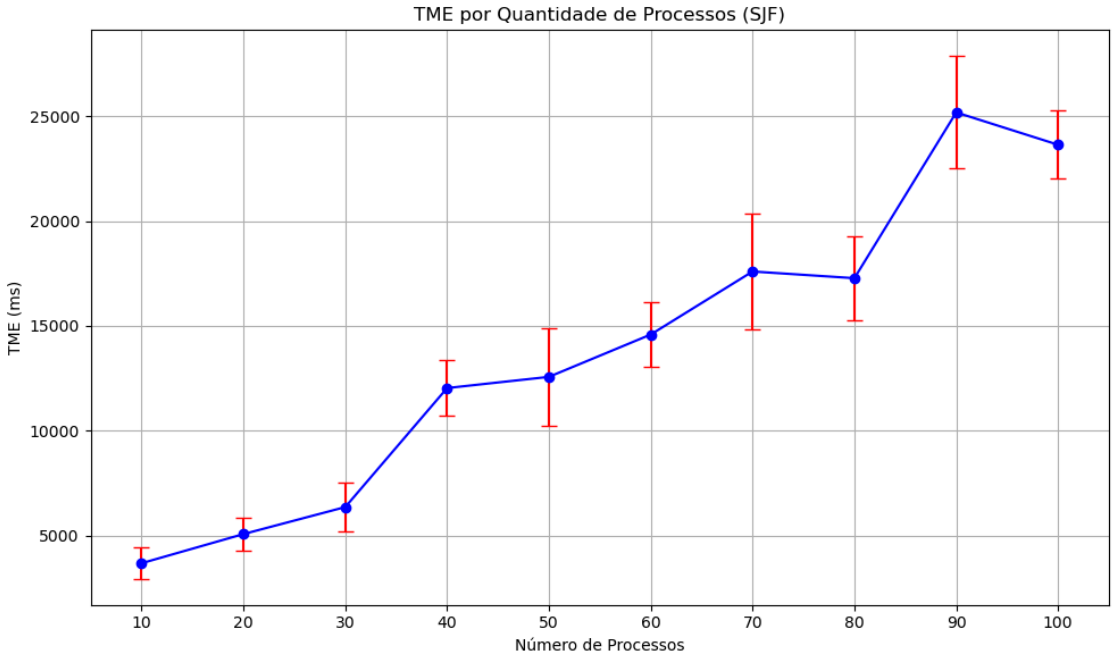

Gráfico 2 - Algoritmo SJF

Observa-se que o algoritmo SJF tende a apresentar tempos médios de espera inferiores em comparação com o FIFO, especialmente em cenários com menor quantidade de processos. Isso ocorre porque o SJF prioriza processos mais curtos, reduzindo o acúmulo de tempo de espera entre os demais. No entanto, a variância pode pode estar mais elevada, uma vez que processos com maior tempo de execução podem acabar esperando mais, dependendo da carga do sistema.

2.3. LJF

Esse algoritmo funciona de maneira contrária ao SJF, selecionando da fila de aptos o processo com maior tempo restante (*remaining_time*) para sua conclusão. Após o término ou liberação do processo, o escalonador seleciona novamente outro processo da fila de aptos que tenha o maior tempo restante, e por assim em diante até a finalização do programa. A seguir, é apresentado o gráfico com os resultados obtidos para o algoritmo LJF, mostrando o tempo médio de espera (linha azul) e sua variância (linha vermelha) conforme o número de processos aumenta.

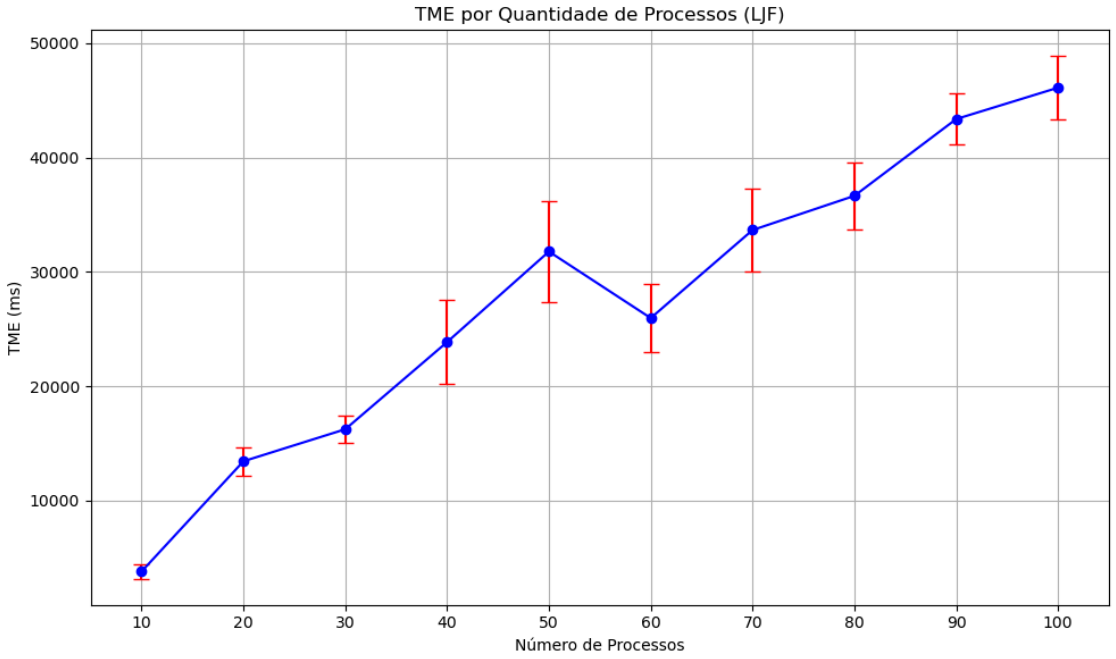

Gráfico 3 - Algoritmo LJF

Com base no gráfico observa-se que, o LJF apresenta um desempenho significativamente inferior ao FIFO. Por priorizar processos com maior tempo de execução, os processos menores tendem a aguardar por longos períodos, resultando em um tempo médio de espera elevado**.** Isso o torna ineficiente em ambientes que exigem tempos de resposta baixos ou justos entre os processos. Devido a esse comportamento, o LJF foi implementado apenas para fins de teste e análise, sendo considerado inviável para aplicação prática em sistemas reais, onde a equidade e o desempenho médio são fatores críticos.

2.4. PRIO_STATIC

O algoritmo PRIO_STATIC utiliza duas filas de prioridade, chamadas ready e ready2, e não há realimentação entre elas. Todos os processos são inseridos inicialmente na fila ready. Após saírem da execução, os processos são redirecionados para uma das filas com base no seu tempo restante de execução (*remaining_time*). Essa decisão é feita com base na variável global MAX_TIME, definida no arquivo main.c e alguns critérios. Sempre que um processo voltar do processador ou da fila de bloqueados, ele retorna para a mesma fila de onde saiu e a seleção do próximo processo para execução é feita com base em probabilidade entre as filas. A seguir, o gráfico mostra o tempo médio de espera (linha azul) e a variância (linha vermelha) para diferentes quantidades de processos.

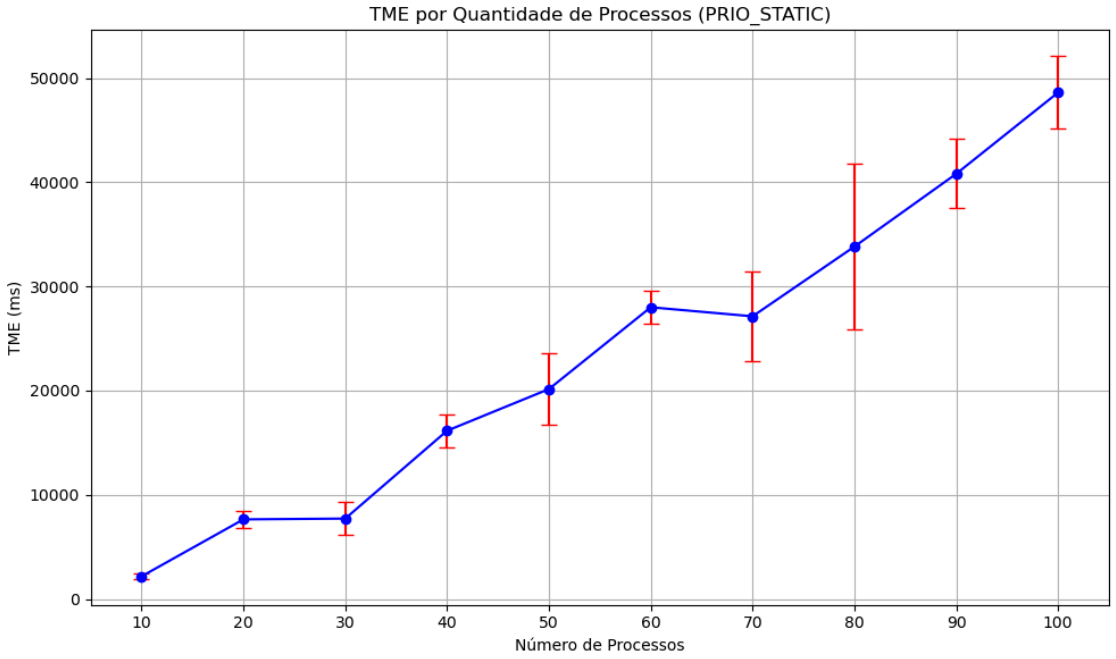

Gráfico 4 - Algoritmo PRIO_STATIC

O gráfico mostra que o PRIO_STATIC, comparado ao FIFO E LJF, consegue manter tempos médios de espera mais baixos e menor variância, graças ao uso de filas separadas por tempo restante de execução. Apesar de não ser tão eficiente quanto o SJF em alguns cenários, ele oferece um equilíbrio melhor entre prioridade e justiça entre os processos.

2.5. PRIO_DYNAMIC

O algoritmo PRIO_DYNAMIC segue parecido com o PRIO_STATIC, no entanto, diferentemente do estático, pode haver realimentação entre as filas, ou seja, um processo pode mudar de fila, prioridade, caso o processo se enquadrar em uma regra destinada a uma fila. Também como estático, todos os processos são inseridos inicialmente na fila ready. As regras para mudança de fila são as seguintes:

- Fila 1 (ready) - FIFO = Processos que saíram por E/S, após voltar de bloqueado, deverão voltar para a primeira fila.
- Fila 2 (ready2) - FIFO = Processos que saíram por preempção, devem retornar para a segunda fila.

O escalonador decide a seleção do próximo processo com base em probabilidade entre as filas. A seguir, o gráfico mostra o tempo médio de espera (linha azul) e a variância (linha vermelha) para diferentes quantidades de processos.

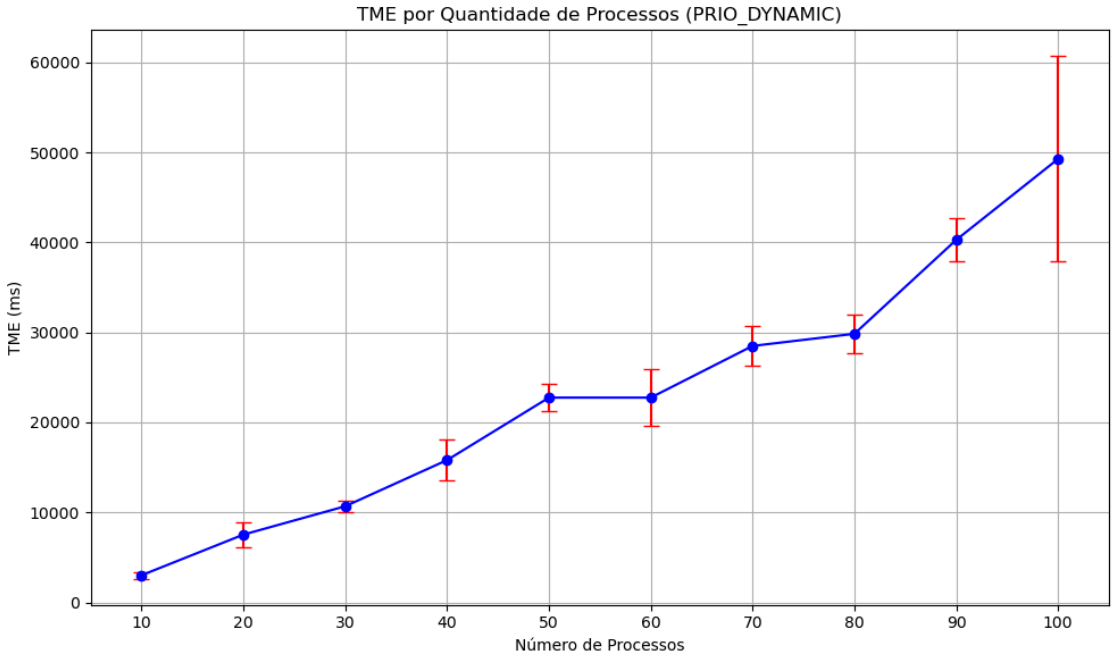

Gráfico 5 - Algoritmo PRIO_DYNAMIC

Analisando o gráfico, observa-se que o algoritmo PRIO_DYNAMIC apresentou um desempenho bastante competitivo, ficando entre os algoritmos com menor tempo médio de espera (TME) geral, atrás apenas do SJF e muito próximo ao PRIO_DYNAMIC_QUANTUM, que ainda vai ser citado neste relatório.

Esse resultado indica que a política de realimentação entre filas, apesar de envolver trocas frequentes de processos entre as filas, conseguiu trazer benefícios ao escalonamento, ao permitir que os processos fossem reavaliados de acordo com seu estado de execução. Mesmo com o custo adicional de controle de filas, o PRIO_DYNAMIC apresentou um desempenho final superior ao FIFO, PRIO_STATIC e LJF, o que reforça seu potencial como uma estratégia eficiente em ambientes com variação constante de processos.

2.1. PRIO_ DYNAMIC_QUANTUM

O algoritmo PRIO_DYNAMIC_QUANTUM segue semelhante com o PRIO_DYNAMIC, também com realimentação entre as filas, no entanto, com a presença de uma nova variável chamada QUANTUM, que está sendo usada não apenas determinando o tempo máximo de execução contínua de um processo antes de sofrer preempção, mas também é usado como critério para mudança de fila. Também como os outros algoritmos de prioridade, todos os processos são inseridos inicialmente na fila ready. As regras para mudança de fila são as seguintes:

- Fila 1 (ready) - FIFO = Processos que utilizaram mais de 50% do seu valor disponível de QUANTUM, deverão voltar para a primeira fila.

- Fila 2 (ready2) - FIFO = Processos que utilizaram menos de 50% do seu valor disponível de QUANTUM, deverão voltar para a segunda fila.

O escalonador decide a seleção do próximo processo com base em probabilidade entre as filas. A seguir, o gráfico mostra o tempo médio de espera (linha azul) e a variância (linha vermelha) para diferentes quantidades de processos.

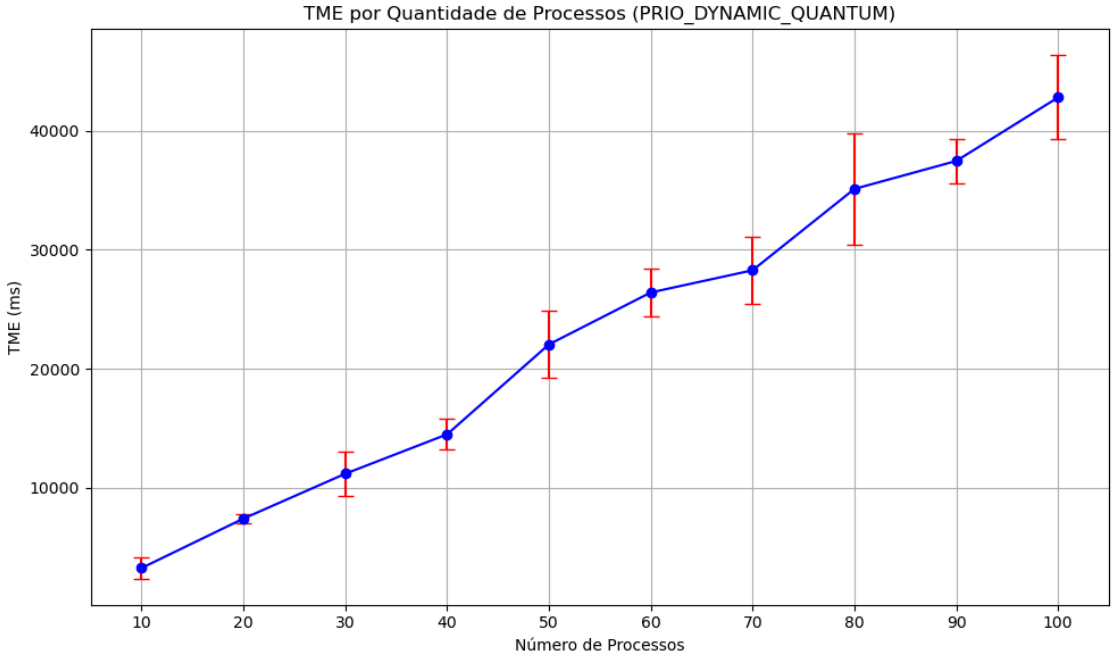

Gráfico 6 - Algoritmo PRIO_DYNAMIC_QUANTUM

Analisando o gráfico do PRIO_DYNAMIC_QUANTUM, observa-se que o algoritmo apresentou um tempo médio de espera (TME) levemente superior ao PRIO_DYNAMIC, ocupando a segunda posição no ranking geral, logo atrás do SJF.

Uma possível explicação para esse comportamento está na introdução do quantum como critério de realimentação, que pode ter aumentado o número de preempções e, consequentemente, o número de trocas de contexto. O desempenho do PRIO_DYNAMIC_QUANTUM foi positivo, superando outros algoritmos como FIFO e PRIO_STATIC. O resultado demonstra que a combinação de prioridade dinâmica com controle por quantum pode ser uma estratégia interessante em determinados contextos, ajudando a ajustar o balanceamento entre processos curtos e longos.

2.2. Comparação Geral

Nos tópicos anteriores foi descrito o funcionamento dos algoritmos e também apresentada a análise gráfica dos resultados obtidos. Algumas comparações parciais já foram feitas ao longo dessas análises, mas, de forma geral, é possível destacar alguns pontos importantes. De acordo com os resultados finais, o algoritmo SJF se destacou como o mais eficiente, apresentando o menor TME médio geral entre todas as

abordagens testadas. Em seguida, os algoritmos PRIO_DYNAMIC_QUANTUM, PRIO_DYNAMIC e PRIO_STATIC também mostraram um desempenho competitivo, com tempos de espera consideravelmente melhores do que o FIFO e o LJF. É importante ressaltar que o algoritmo LJF, que, como mencionado, foi implementado apenas para fins de teste e comparação, uma vez que sua estratégia não é adequada para um ambiente real de escalonamento, gerou tempos altos.

Os algoritmos dinâmicos com realimentação e controle de quantum conseguiram reduzir o tempo médio de espera em comparação ao FIFO e ao PRIO_STATIC. Isso indica que o uso de mecanismos de preempção, somado a políticas de realocação inteligente de filas, pode trazer vantagens em cenários com maior quantidade de processos. A seguir, é apresentado o gráfico comparativo com todos os algoritmos analisados, mostrando o tempo médio de espera (linha azul) e a variância (linha vermelha) em função da quantidade de processos.

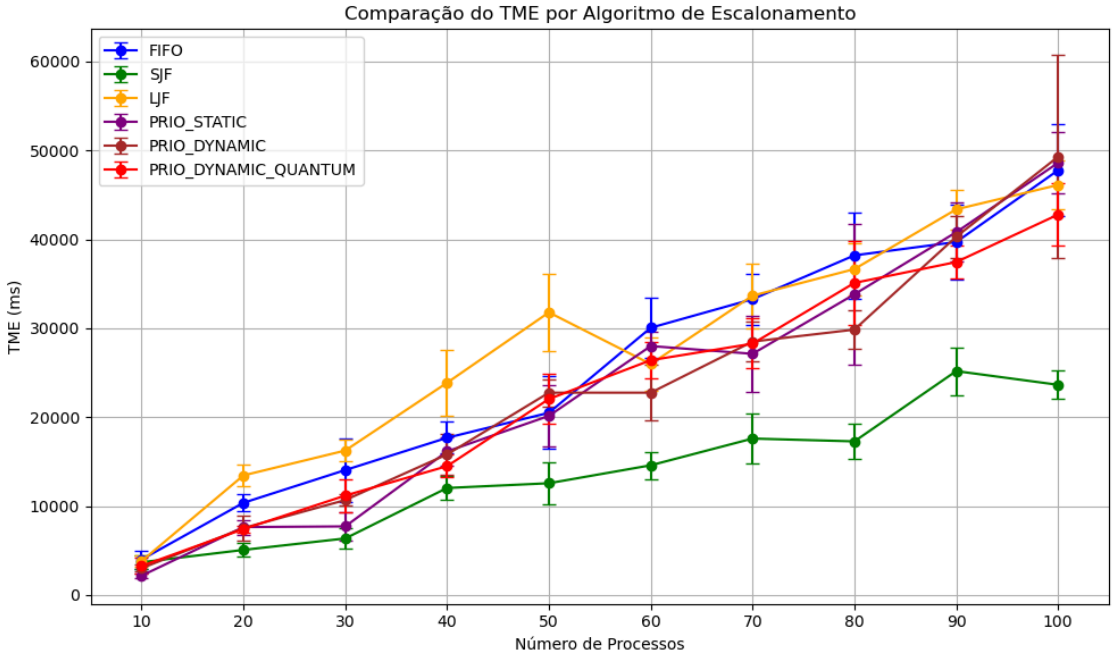

Gráfico 7 - Todos os algoritmos

Para facilitar a visualização comparativa dos resultados, foi criado um script em Python chamado ranking.py, responsável por ranquear os algoritmos com base no menor tempo médio de espera (TME) geral. A seguir, é apresentada uma captura de tela da parte do código responsável pelo cálculo dos valores.

```
resultados = []

for nome_arquivo, nome_escalonador in arquivos_e_escalonadores:
    dados = []
    with open(nome_arquivo, "r") as f:
        for linha in f:
            if "TME:" in linha:
                partes = linha.strip().split()
                tme = float(partes[1])
                n_procs = int(partes[2])
                dados.append((n_procs, tme))

    df = pd.DataFrame(dados, columns=["n_procs", "tme"])
    media_geral = df["tme"].mean()
    resultados.append((nome_escalonador, media_geral))

# Ordena do menor para o maior TME
ranking = sorted(resultados, key=lambda x: x[1])
```

Figura 1 - Código para geração do ranking

Com a execução desse script, foi possível gerar uma tabela ordenada, classificando os algoritmos de acordo com o menor TME médio obtido durante os testes, permitindo uma análise direta de quais algoritmos apresentaram melhor desempenho em relação ao tempo de espera.

Tabela - Ranking de menores TME		
Posição	**Algoritmo**	**TME médio geral**
1°	SJF	13801.50 ms
2°	PRIO_DYNAMIC_QUANTUM	22840.47 ms
3°	PRIO_DYNAMIC	23046.32 ms
4°	PRIO_STATIC	23217.42 ms
5°	FIFO	25557.69 ms
6°	LJF	27494.61 ms

Tabela 1 - Ranking TME

3. Conclusão

A partir da implementação e análise dos diferentes algoritmos de escalonamento de processos, foi possível compreender de forma prática o impacto que cada política de escalonamento tem no desempenho do sistema, especialmente em relação ao tempo médio de espera (TME).

Os resultados mostraram que o algoritmo SJF confirmou sua eficiência teórica, apresentando o menor TME médio geral. Por outro lado, o LJF, como esperado, obteve o pior desempenho, devido seguir uma estratégia indesejada. Um resultado interessante foi o bom desempenho dos algoritmos com prioridades dinâmicas e uso de quantum, que superaram o FIFO e o PRIO_STATIC, demonstrando que a preempção e a realimentação de filas, quando bem configuradas, podem ser vantajosas em termos de tempo de espera.

De modo geral, este trabalho permitiu explorar conceitos importantes sobre escalonamento de processos, além de evidenciar que não existe um algoritmo universalmente melhor, sendo a escolha do escalonador fortemente dependente do tipo de carga de trabalho, características do sistema e objetivos de desempenho desejados.

References

Neto, João. Sistemas Operacionais. [Slide: Escalonamento de Processos]. Universidade Federal do Rio Grande do Norte, 2025. Acesso em: 10 jun. 2025.