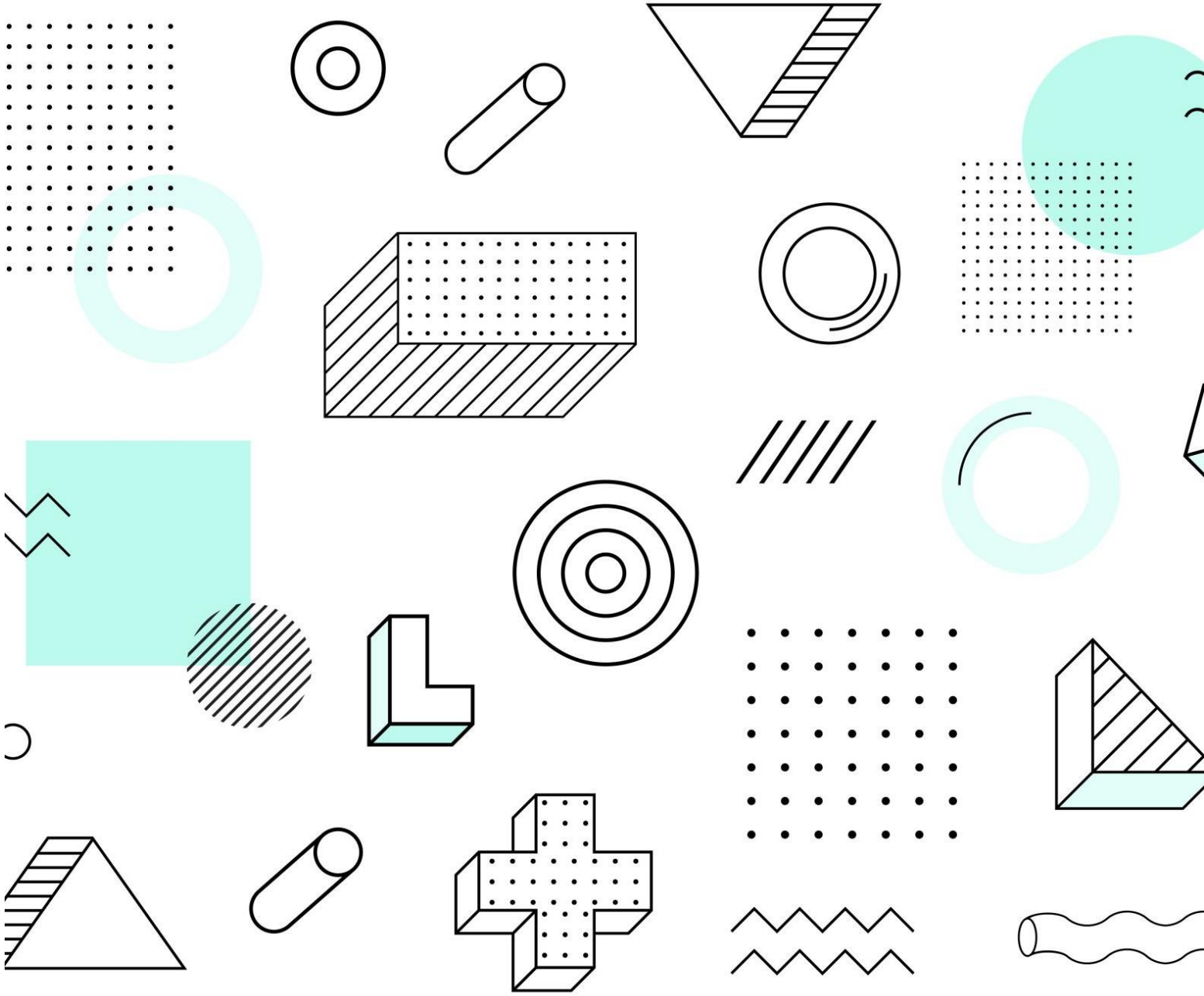
Comparação entre Algoritmos de Escalonamento

FCFS, SRTN e Prioridade

Otávio Garcia Capobianco
NUSP: 15482671



Introdução

- Implementamos um simulador de escalonamento de processos, que pode simular os algoritmos First Come First Served (**FCFS**), Shortest Remaining Time Next (**SRTN**) e **Escalonamento por Prioridade**
- Nesta apresentação, mostraremos resultados de experimentos realizados nesse simulador. Mais especificamente, discutiremos como cada um dos algoritmos acima se comporta no tocante a quantos **prazos de finalização** de processos foram cumpridos, e quantas **preempções** cada um teve que realizar, sob diferentes circunstâncias
- Alguns detalhes da implementação serão abordados para garantir a consistência dos resultados

Especificações

- Para ter uma amplitude interessante nos testes, realizamo-nos em 2 máquinas distintas. Abaixo estão os processadores delas e quantos núcleos cada um possui
  - **Intel Core i5-5350U** com 2 núcleos, permitindo **4 threads simultâneas** através de hyperthreading
  - **Intel Core i7-4790K** com 4 núcleos, permitindo **8 threads simultâneas** através de hyperthreading
  - Ambos os computadores estavam utilizando Linux durante a execução dos programas relevantes
  - Todo o código do simulador implementado foi escrito em C, e os gráficos presentes nesta apresentação foram gerados em Python, utilizando Matplotlib

Determinismo

- Primeiramente, precisamos garantir que, **para uma mesma entrada**, o código sempre **gerará a mesma saída**
- Uma possível causa para problemas neste sentido seria sincronizar completamente a execução das threads com o **tempo real** que tenha passado, utilizando os valores decimais granulares na hora de verificar quanto tempo resta a um processo, quando o mesmo acabou, e assim por diante
  - Para evitar isso, implementamos **contadores inteiros** para guardar quanto tempo se passou. Dessa forma, apesar de deixarmos as threads realizando tarefas por tempo real, em um escopo global, sempre incrementaremos o tempo em 1 segundo
  - Assim, pequenas flutuações de tempo que naturalmente surgiriam ao longo da execução do programa, bem como o curto tempo consumido pelo próprio escalonador ao realizar suas tarefas, são desconsiderados para os resultados e continuidade da execução

Determinismo

- Outros problemas poderiam surgir do fato de usarmos **múltiplas threads** rodando concorrentemente em cada núcleo da CPU para ter maior eficiência
  - Impedimos que isso torne-se um problema de fato pelo uso adequado de **mutexes** e **barreiras de sincronização**.
  - Em todo acesso a variáveis globais, trancamos o mutex associado antes, e nunca há impedimentos para que o mesmo seja destrancado logo depois. Isso, dentre outras minúcias da implementação, garante as 4 propriedades de proteção de seções críticas
  - Temos dois tipos de entidades rodando: um coordenador principal (escalonador) e as threads. A cada iteração do contador de tempo, o escalonador aguarda todas as threads atualmente rodando chegarem ao fim da execução daquele ciclo (instante de tempo). Quando isso ocorre, elas sinalizam e aguardam a sinalização de que o escalonador acabou suas tarefas e que um novo ciclo pode iniciar
- Com isso, cada ação ocorre em **seções compartimentalizadas** em que ela pode acontecer, levando à previsibilidade e determinismo dos resultados

Escalonamento por Prioridade

- Este foi o único dos algoritmos utilizados que considera todas as informações fornecidas na entrada: t0, dt e deadline, ao tomar decisões
- A estrutura básica utilizada é a de um **round-robin com quantums variáveis**, que dependem da urgência para finalizar cada processo
- Para cada núcleo de processamento, é mantida uma fila de prontos. A cada instante de tempo, o escalonador verifica se algum processo novo chegou, ou se o quantum de algum dos que estavam rodando acabou
- Quando processos chegam, escolhe-se para ele o núcleo com a menor quantia de trabalho total, dentre processos prontos e rodando. Caso algum processo esteja rodando, o recém-chegado é adicionado à fila; caso contrário, ele roda
- Caso algum quantum tenha acabado, o próximo da correspondente fila de prontos (caso exista) começa a rodar em seu lugar. Em outras palavras, ocorre **preempção**

Escalonamento por Prioridade

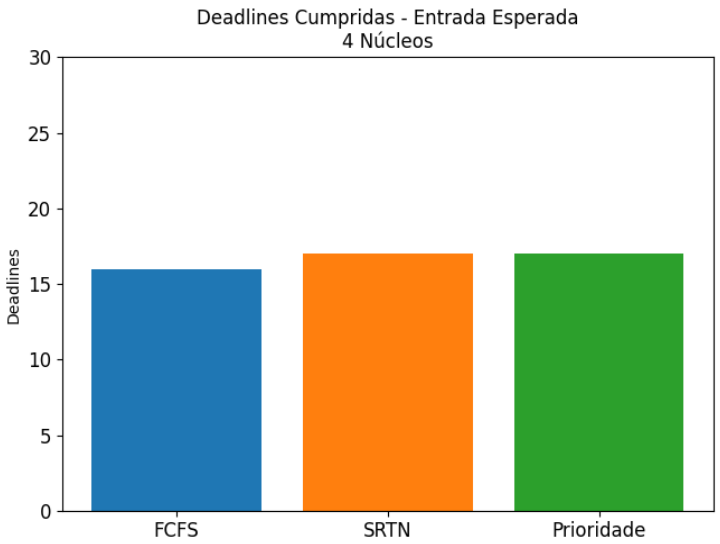
- No momento em que um processo começa a rodar, deve ser escolhido um quantum para ele, de modo a tentar cumprir a deadline
- Assim, escolhemos um valor de prioridade de forma diretamente proporcional ao **tempo restante** para aquele processo, e inversamente proporcional à **distância entre o tempo atual e a deadline** a ser cumprida. Para tal, definimos o quantum por:

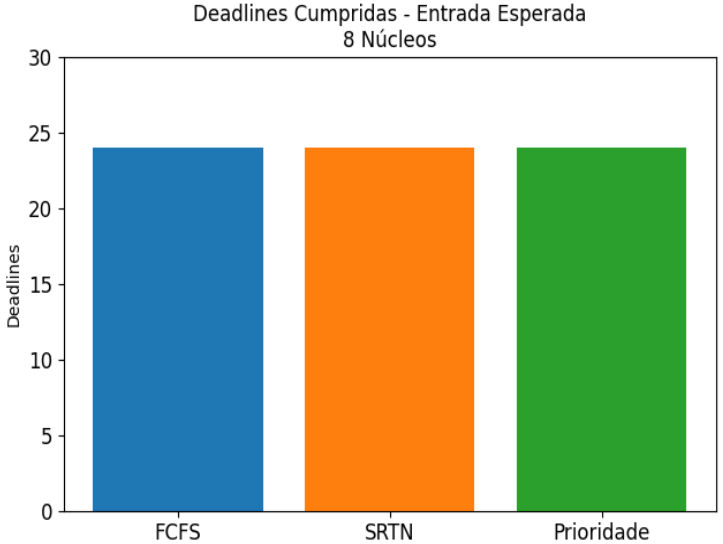
```
max{1, (execução_restante * 10) / (deadline – tempo_atual)}
```

- Note que o escalonador se adapta constantemente a quão urgente um dado processo é, pois a cada vez que o quantum for decidido, uma nova situação atual é considerada
- Portanto, nosso escalonamento por prioridade tem características próximas de escalonadores que buscam interatividade, revezando processos com uma certa constância em situações esperadas, e busca **cumprir o máximo de deadlines** que puder

Experimentos – Deadlines Cumpridas

Entrada Esperada



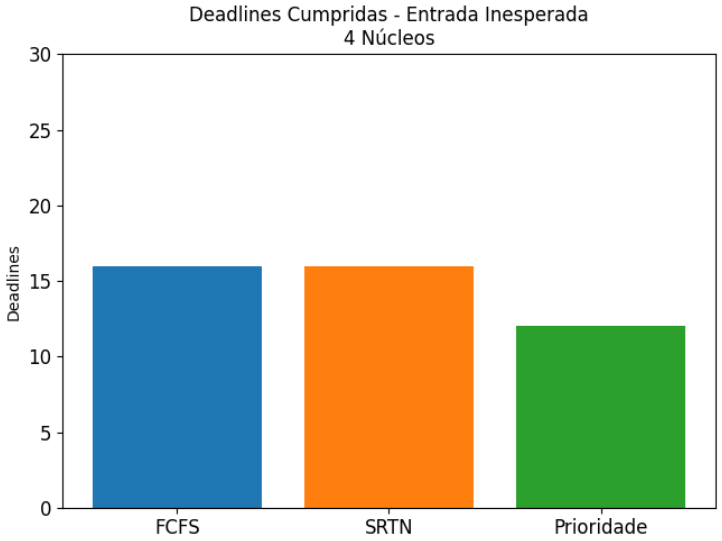


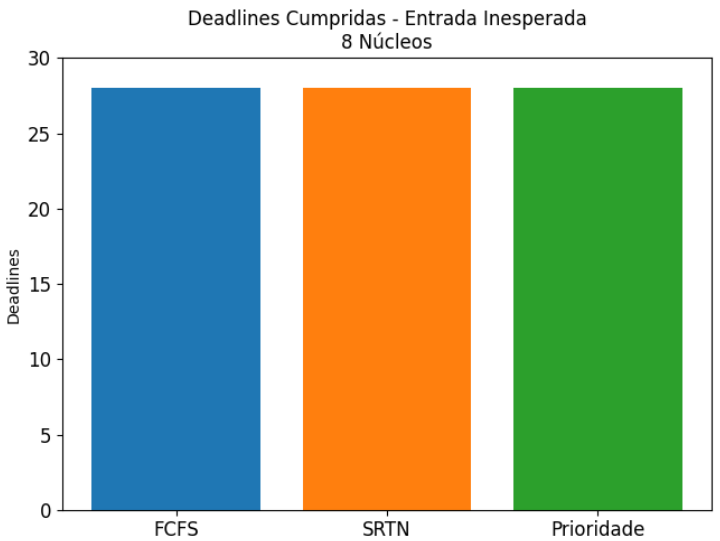
Foi utilizado um arquivo com entradas “aleatórias”, **sem um critério definido**, para verificar o caso médio. O arquivo contém **24 processos** ao todo, totalizando 120 segundos de tempo de execução

- Para a máquina de 4 cores, o desempenho neste aspecto foi bem similar entre os algoritmos. O FCFS concluiu um processo a menos a tempo, e o SRTN e escalonamento por Prioridade a mesma quantia, totalizando **17 deadlines**. Entretanto, os **maiores processos não conseguiram rodar** por tempo suficiente no SRTN, mas muitos deles cumpriram suas deadlines no por prioridade, sugerindo um critério de justiça menos enviesado neste último
- Para a de 8 cores, qualquer arquivo de entrada que tenha um desempenho decente na máquina menos potente consegue ser completamente concluído a tempo, e foi o que ocorreu. **Todos os processos** tiveram suas deadlines cumpridas

Experimentos – Deadlines Cumpridas

Entrada Inesperada

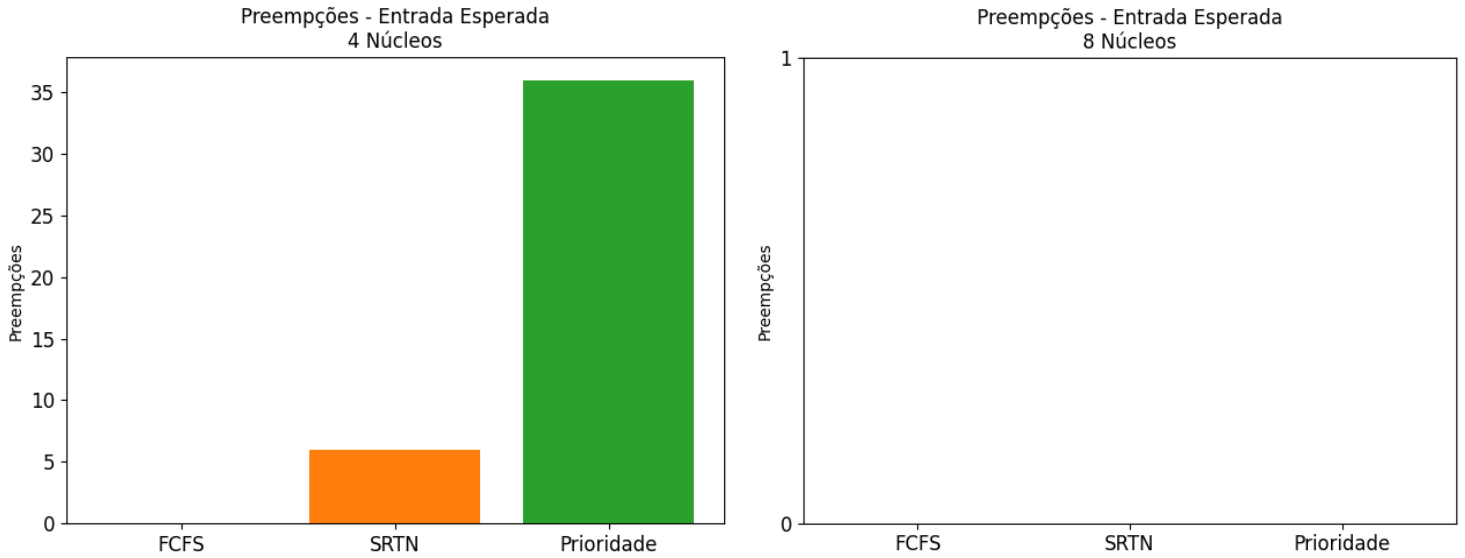




Foi utilizado um arquivo com processos tais que, ao chegar, seu tempo de execução seria maior do que o restante do atualmente rodando no núcleo. Por isso, o **SRTN se comporta igual ao FCFS**. O arquivo contém **28 processos** ao todo, totalizando 100 segundos de tempo de execução

- Para a máquina de 4 cores, o desempenho do **FCFS e do SRTN foi o mesmo, cumprindo 16 deadlines** por conta da criação de um arquivo para que isso ocorresse. Ademais, escolhemos casos em que o escalonador por prioridade quase acaba alguns processos antes da deadline, mas a **perde por poucos segundos, cumprindo 12**. Esses resultados são inesperados, dado que, em média, esperamos que o FCFS seja o pior, por não levar nenhuma métrica além do tempo de chegada em conta, e o por prioridade o melhor
- Para a de 8 cores, ocorre o mesmo que na entrada esperada. **Todas as 28 deadlines são cumpridas**

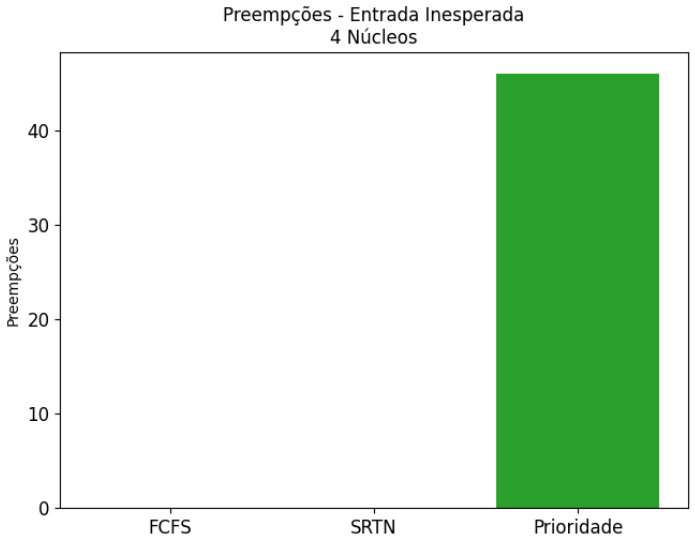
Experimentos – Preempções
Entrada Esperada

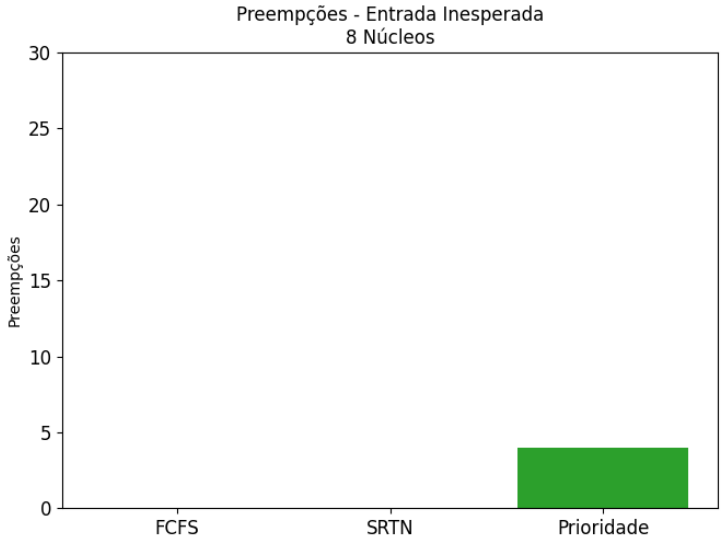


Mesmo arquivo de trace das deadlines esperadas, com 24 processos e 120 segundos totais

- Para 4 cores, o FCFS não faz **nenhuma preempção**, pois por definição ele não tem preempções, o SRTN faz **6 preempções** e o escalonador por prioridade faz 7 vezes mais, com **36 preempções**. Isto é esperado, pois o por prioridade faz uma preempção a cada fim de quantum, para cada núcleo, e o SRTN o faz apenas sob circunstâncias mais específicas
- Para 8 cores, não há **preempção em nenhum dos algoritmos**. Isso indica que, no caso "aleatório" dessa entrada, mesmo com alguns processos de maior duração, nunca ocorreu o caso em que um quantum acaba com algum processo aguardando na fila de prontos, ou de chegar um processo mais curto que dispute um core no SRTN. Novamente, vemos que 8 núcleos de processador conseguem executar os processos desta entrada com eficiência máxima

Experimentos – Preempções
Entrada Inesperada





Mesmo arquivo de trace das deadlines inesperadas, com 28 processos e 100 segundos totais

- Para 4 cores, a diferença mais evidente vindo da entrada esperada é que, aqui, o **SRTN não realiza preempções**. Isso ocorre pois, como já mencionado, sempre que um processo chega, ao comparar seu tempo de execução com o tempo restante do núcleo a que foi atribuído, seu tempo é maior. Assim, os **processos rodam ininterruptamente do início ao fim, tal qual fazem no FCFS**, onde não há preempção. O por prioridade segue o esperado, com 46 preempções
- Para 8 cores, o **mesmo padrão visto na entrada esperada** é observado aqui. A única diferença é que o por prioridade realizou 4 preempções, em decorrência de, no modo arbitrário de escolher os processos de entrada, termos **agrupado tempos de chegada em um intervalo curto**. Ainda assim, são poucas preempções, deixando claro, novamente, a suficiência de 8 núcleos

Conclusão

- De modo geral, os algoritmos **seguiram na prática o que sabíamos sobre eles** na teoria. A presença de muitos núcleos de processamento acaba por normalizar os resultados de certo modo, dadas as restrições no número de processos e tempo de execução total, fator já visível com 4 núcleos e óbvio com 8, mas muitos dos resultados ainda permitem observações contundentes
- Claramente, o uso de **multithreading** para esta tarefa **tem benefícios** que superam em muito o pequeno overhead de gerenciamento de variáveis de controle
- Diferentes escolhas de arquivos de entrada podem gerar resultados vastamente distintos, e **é impossível projetar um algoritmo de escalonamento que seja ideal para toda situação** possível. Compreender as diferentes nuances de cada um deles é essencial caso suas tarefas envolvam criar ou escolher um destes algoritmos sob circunstâncias específicas. E, de fato, a implementação desta simulação permitiu uma visão mais ampla sobre a natureza dessas nuances.