Experimento 1: Entendendo o MLFQ Básico

Pra começar, rodei este comando: `python3 mlfq.py -n 3 -Q 5,10,20 -B 30 -l 0,20,0:2,8,0:4,12,0 -c`. Com isso, configurei um MLFQ com 3 filas de prioridade (de 0 a 2). A fila 2 tinha um quantum de 5 unidades de tempo, a fila 1 tinha 10, e a fila 0, 20. Ah, e programei um "boost" de prioridade pra rolar a cada 30 unidades de tempo. Criei três jobs na mão, sem nenhuma operação de entrada/saída.

a) Qual foi o tempo de resposta e turnaround de cada job?

As "Estatísticas finais" do simulador me deram todos os números bonitinhos:

- **Job 0:**
  - **Tempo de resposta:** 0 (ele começou a rodar assim que chegou, na hora zero).
  - **Tempo de turnaround:** 35 (terminou lá no tempo 35).
- **Job 1:**
  - **Tempo de resposta:** 3 (chegou no tempo 2, mas só foi começar no tempo 5; então, 5−2=3).
  - **Tempo de turnaround:** 26 (chegou no tempo 2 e terminou no tempo 28; 28−2=26).
- **Job 2:**
  - **Tempo de resposta:** 6 (chegou no tempo 4, mas a execução dele só iniciou no tempo 10; logo, 10−4=6).
  - **Tempo de turnaround:** 36 (chegou no tempo 4 e concluiu no tempo 40; 40−4=36).

b) Algum job desceu de prioridade? Se sim, quando?

Sim, e foi bem legal de ver! Todos os jobs acabaram descendo de prioridade. Isso porque a duração de execução de cada um deles era maior do que o quantum da fila de prioridade mais alta (que era a fila 2, com quantum de 5).

- **Job 0** (duração 20):
  - Ele começou na **PRIORIDADE 2**, no tempo 0.
  - Rodou por 5 unidades de tempo (do tempo 0 ao 4).
  - No tempo 15, o Job 0 voltou a ser executado, mas aí já estava na **PRIORIDADE 1**. Isso significa que ele foi rebaixado depois de usar todo o quantum dele na prioridade 2 (lá no tempo 4, depois de gastar 5 ticks). Daí, ele esperou na fila de prioridade 1 até ser escalonado novamente.
- **Job 1** (duração 8):
  - Começou na **PRIORIDADE 2**, no tempo 5.
  - Rodou por 5 unidades de tempo (do tempo 5 ao 9).
  - No tempo 25, o Job 1 retornou à execução, mas agora na **PRIORIDADE 1**. Ele desceu de prioridade depois de esgotar o quantum dele na prioridade 2 (no tempo 9, depois de usar 5 ticks).
- **Job 2** (duração 12):
  - Começou na **PRIORIDADE 2**, no tempo 10.

    - Rodou por 5 unidades de tempo (do tempo 10 ao 14).
    - No tempo 28, o Job 2 voltou a ser executado, mas já na **PRIORIDADE 1**. Ele também desceu de prioridade depois de esgotar seu quantum na prioridade 2 (no tempo 14, após consumir 5 ticks).

c) O boost ocorreu? E o que ele fez?

Sim, o boost rolou direitinho no tempo 30, exatamente como mostra a linha `[ tempo 30 ] BOOST (a cada 30)` no log de rastreamento.

O impacto do boost foi **restaurar a prioridade de todos os jobs que estavam ativos pra fila de prioridade mais alta (Prioridade 2)**. Deixa eu te mostrar o que vi:

- No tempo 29, o Job 2 estava sendo executado na **PRIORIDADE 1**.
- No tempo 30, *bum*, o boost acontece.
- Logo no tempo 30, imediatamente após o boost, o Job 0 (que também estava na **PRIORIDADE 1** desde o tempo 15) foi agendado e voltou a ser executado na **PRIORIDADE 2**.
- Quando o Job 0 terminou (no tempo 35), o Job 2 foi agendado de novo e passou a rodar na **PRIORIDADE 2** (no tempo 35), mesmo tendo sido interrompido da **PRIORIDADE 1** no tempo 29.

Isso deixa bem claro o papel fundamental do boost: ele não deixa que jobs "longos" que caíram pra prioridades mais baixas fiquem presos lá pra sempre (isso se chama *starvation*). Ao jogá-los de volta para a fila de maior prioridade, o boost garante uma nova chance de execução mais rápida, o que acaba trazendo mais justiça ao escalonamento com o tempo.

Experimento 2: Mudando os Quanta

Pra essa rodada, eu mudei os quanta e rodei este comando: `python3 mlfq.py -n 3 -Q 2,4,8 -B 30 -l 0,20,0:2,8,0:4,12,0 -c`. Agora, os novos quanta eram: fila 2: 2, fila 1: 4, e fila 0: 8. O boost continuou a cada 30 unidades de tempo.

a) O tempo de resposta dos jobs aumentou ou diminuiu?

- Pro Job 0, o tempo de resposta continuou 0, porque ele ainda é o primeiro a chegar e começa a rodar imediatamente.
- Mas pros Jobs 1 (chegou em t=2) e 2 (chegou em t=4), o tempo de resposta **diminuiu pra 0**. Isso é ótimo! Significa que eles começaram a executar assim que se tornaram elegíveis, mesmo com o Job 0 sendo interrompido. No Experimento 1, eles tiveram que esperar mais tempo pelo Job 0.

b) Teve mais ou menos trocas de contexto?

Nossa, teve **muito mais trocas de contexto**!

No Experimento 1 (quando os quanta eram maiores):

- O Job 0 rodava por 5 ticks, depois o Job 1 por 5 ticks, e depois o Job 2 por 5 ticks.

No Experimento 2 (com quanta menores, 2,4,8), basta olhar o rastreamento:

- `[ tempo 0 ] Executando JOB 0 na PRIORIDADE 2 [ TICKS 1 RESTANTE 19 ]`
- `[ tempo 1 ] Executando JOB 0 na PRIORIDADE 2 [ TICKS 0 RESTANTE 18 ]` (Job 0 foi interrompido depois de apenas 2 ticks)
- `[ tempo 2 ] Executando JOB 1 na PRIORIDADE 2 [ TICKS 1 RESTANTE 7 ]`
- `[ tempo 3 ] Executando JOB 1 na PRIORIDADE 2 [ TICKS 0 RESTANTE 6 ]` (Job 1 foi interrompido depois de 2 ticks)
- `[ tempo 4 ] Executando JOB 2 na PRIORIDADE 2 [ TICKS 1 RESTANTE 11 ]`
- `[ tempo 5 ] Executando JOB 2 na PRIORIDADE 2 [ TICKS 0 RESTANTE 10 ]` (Job 2 foi interrompido depois de 2 ticks)
- `[ tempo 6 ] Executando JOB 0 na PRIORIDADE 1` (Job 0 de volta, mas agora na fila 1)

Como o quantum da fila de maior prioridade (Prioridade 2) é de só 2 ticks, qualquer job que tente usar a CPU é interrompido muito mais rápido. Isso faz com que os jobs se revezem na CPU de forma mais acelerada e, consequentemente, o número de trocas de contexto aumenta bastante.

c) O que aconteceu com os jobs curtos?

O Job 1 (duração 8) é o job mais curto nesse grupo.

- **Melhora no Tempo de Resposta:** O tempo de resposta dele caiu de 3 pra 0. Isso foi ótimo! Mesmo com outros jobs presentes, ele conseguiu começar a rodar quase que imediatamente depois que chegou, o que é um benefício direto de ter quanta menores e mais trocas de contexto.
- **Transição de Prioridade Mais Rápida (e retorno):** Ele desceu de prioridade mais rapidamente, mas também teve a chance de ser agendado mais cedo nas filas de prioridade mais alta.
- **Finalização Rápida:** O Job 1 terminou no tempo 28, o que resultou num tempo de turnaround muito bom pra ele (26 unidades de tempo). No Experimento 1, ele também terminou no tempo 28, mas a percepção da resposta inicial é melhor neste experimento.

Experimento 3: Starvation e o Impacto do Boost (Parte 1)

Aqui, o comando foi: `python3 mlfq.py -n 3 -Q 2,4,8 -B 10000 -l 0,60,0:2,60,0:4,60,0:6,5,0 -c`. Nesse cenário, coloquei três jobs bem longos (duração 60) e um job curto (Job 3, duração 5). O intervalo do boost estava super longo (10000 unidades de tempo), o que, na prática, significava que o boost não aconteceria enquanto esses jobs estivessem rodando.

a) O job mais curto sofreu starvation?

O Job 3 era o mais curto, com 5 unidades de tempo de duração e chegando no tempo 6. Analisando o rastreamento da execução, eu vi o seguinte:

- O Job 3 começou a rodar na **PRIORIDADE 2** no tempo 6.
- Ele executou por 2 ticks (`[ tempo 6 ] Executando JOB 3 na PRIORIDADE 2 [ TICKS 1 RESTANTE 4 ]` e `[ tempo 7 ] Executando JOB 3 na PRIORIDADE 2 [ TICKS 0 RESTANTE 3 ]`).
- No tempo 8, o Job 0 (que é um dos jobs longos) foi agendado na **PRIORIDADE 1**.
- O Job 3 só voltou a ser executado no tempo 20 (`[ tempo 20 ] Executando JOB 3 na PRIORIDADE 1`). Ele tinha sido rebaixado pra **PRIORIDADE 1**.
- Ele rodou por 3 ticks (do tempo 20 ao 22).
- No tempo 23, o Job 3 **FINALIZOU**.

Minha conclusão: O Job 3 não sofreu uma starvation *prolongada*, mas o tempo que ele levou pra concluir foi bem atrasado pelos jobs mais longos.

Vamos dar uma olhada nos números:

- Tempo de Chegada do Job 3: 6
- Tempo de Início da Execução do Job 3: 6 (imediatamente)
- Tempo de Conclusão do Job 3: 23

Se pensarmos que um job que dura 5 unidades poderia, idealmente, terminar lá pelo tempo 11 (6+5), um tempo de conclusão de 23 mostra que ele teve que esperar bastante. Ele foi interrompido várias vezes e acabou descendo de prioridade. Se não houvesse boost, ou se a duração dos jobs longos fosse ainda maior, ou se o job curto tivesse chegado mais cedo, aí sim ele teria sofrido uma starvation *severa*.

Nesse caso específico, os quanta curtos (2,4,8) até que permitiram que o Job 3 rodasse um pouco na prioridade mais alta, fosse rebaixado e, eventualmente, conseguisse um pedaço de tempo na fila 1 pra terminar. No entanto, ele ainda foi "punido" pela presença dos jobs longos que consumiram uma boa parte da CPU antes que ele pudesse concluir. Se a política fosse tipo *First-Come, First-Served*, por exemplo, ele teria esperado muito mais.

A definição clássica de starvation é um processo que nunca roda ou roda por um tempo muito pequeno. Aqui, ele rodou, mas com atrasos consideráveis, o que é um sinal de que a falta de um boost frequente afetou negativamente o desempenho dele.

b) O que mudou quando eu incluí o boost frequente?

A mudança mais significativa que notei foi a **justiça no acesso à CPU e a prevenção da starvation para jobs que desceram de prioridade**.

- **Prevenção de Starvation (para jobs longos e curtos):** No cenário sem boost frequente (`-B 10000`), os jobs longos (0, 1, 2) caíram rapidinho para a prioridade mais baixa (0) e ficaram por lá, rodando em round-robin com um quantum maior. O Job 3, mesmo sendo curto, teve que esperar que esses jobs caíssem o suficiente pra ter sua vez. Mas com o boost a cada 20 unidades de tempo, todos os jobs, incluindo os longos, são "resetados" pra prioridade mais alta (Prioridade 2). Isso significa que:
    - Jobs longos (0, 1, 2) não ficam "presos" na fila de prioridade mais baixa indefinidamente; eles têm chances periódicas de serem executados com um quantum menor na fila superior, e depois descem de novo.

    - O Job 3 (curto), mesmo se fosse interrompido, seria rapidamente movido de volta para a fila de alta prioridade com o próximo boost, garantindo que ele sempre tivesse chances de ser executado rapidamente. Embora, nesse caso específico, ele tenha terminado mais tarde (tempo 47 contra 23), o mecanismo de boost garante que ele não será esquecido.
- **Aumento das Trocas de Contexto:** Como todos os jobs são constantemente "promovidos" pra prioridade mais alta, a concorrência na fila de prioridade 2 (com quantum de 2) aumenta a cada ciclo de boost. Isso gera mais trocas de contexto e mais *overhead* para o escalonador, mas o objetivo é a justiça.
- **Comportamento mais imprevisível para jobs curtos em um cenário com muitos jobs longos:** Curiosamente, nesse caso, o tempo de turnaround do Job 3 foi maior (41 contra 17). Isso acontece porque, com o boost frequente, os três jobs longos também estão sendo constantemente promovidos de volta para a prioridade mais alta. Ou seja, o Job 3, que é curto, agora tem que competir repetidamente com três jobs longos que também estão na prioridade máxima após cada boost. Sem o boost, os jobs longos desceram de prioridade rapidamente, dando mais espaço pro Job 3 na fila 1 terminar. Com o boost, eles voltam a ser "altamente prioritários" a cada 20 tempos, o que pode atrasar a finalização do job curto que precisa de poucas unidades de tempo na fila de maior prioridade.

A principal mudança é uma tentativa de garantir que nenhum job seja completamente ignorado, introduzindo periodicamente a chance de todos competirem pelas prioridades mais altas. Isso é uma estratégia de "envelhecimento" (ou *aging*) pra evitar a fome.

Experimento 4: Testando -S e -I

E pra fechar, eu rodei: `python3 mlfq.py -n 3 -Q 3,6,10 -B 50 -l 0,60,5:0,60,5 -S -I -c`. Nesse experimento, eu tinha dois jobs idênticos (duração 60, e I/O a cada 5 unidades de tempo). O que mudou é que ativei os parâmetros `-S` (Mantém prioridade após I/O) e `-I` (Enfileira no topo após I/O).

a) Qual job teve menor turnaround?

Olhando as "Estatísticas finais", eu vi que:

- Job 0: turnaround 145
- Job 1: turnaround 148

O **Job 0** teve o menor tempo de turnaround (145), sendo só um pouquinho mais rápido que o Job 1 (148). A diferença é pequena porque os dois jobs são iguais em todas as características (duração, frequência de I/O) e chegam ao mesmo tempo (tempo 0). Essa pequena variação pode ser por causa da ordem interna que o simulador usa pra escolher entre jobs de mesma prioridade quando há vários prontos (tipo um FIFO na fila). Nesse caso, o Job 1 foi o primeiro a ser executado (`[ tempo 0 ] Executando JOB 1...`), mas o Job 0 conseguiu terminar um pouco antes.

b) Que diferença os parâmetros -S e -I provocaram?

Esses parâmetros `-S` e `-I` impactam *muito* o comportamento de jobs que fazem I/O, especialmente quando usados juntos:

- **-S (Permanece na prioridade após I/O? True):** Normalmente, se um job usa seu quantum e é interrompido ou começa uma I/O, os "ticks" que ele usou na CPU são zerados, e ele pode até ir pra uma fila de prioridade mais baixa se esgotou o quantum. Com o `-S`, se um job começa uma operação de I/O antes de usar todo o seu quantum, ele não é penalizado perdendo prioridade. Quando ele volta da I/O, ele mantém a prioridade que tinha quando iniciou a operação. Isso é crucial pra jobs interativos, já que eles fazem I/O o tempo todo.
- **-I (Enfileira no topo após I/O? True):** Esse foi o mais impactante. Quando um job termina a operação de I/O, ele é *imediatamente* movido pra fila de maior prioridade (Prioridade 2, nesse caso) e vai pro *início* dessa fila. Isso coloca ele numa posição super privilegiada pra ser agendado rapidamente.

A Diferença que Percebi: A combinação `-S` e `-I` cria um viés fortíssimo a favor de jobs que fazem muita E/S. Cada vez que um job com I/O termina sua operação, ele é "recompensado" com a prioridade máxima, garantindo que ele pegue a CPU rapidamente de novo. Deu pra ver isso várias vezes no log:

Exemplo:

- `[ tempo 8 ] INÍCIO DE E/S do JOB 1`
- `[ tempo 13 ] FIM DE E/S do JOB 1`
- `[ tempo 13 ] Executando JOB 1 na PRIORIDADE 1 [ TICKS 5 RESTANTE 54 ]` (O Job 1 voltou pra Prioridade 1 e imediatamente começou a rodar de novo, mesmo tendo sido interrompido do quantum da Prioridade 2 no tempo 2)

Mesmo tendo um quantum na Fila 2 de 3 unidades, ambos os jobs raramente completavam 3 ticks na Prioridade 2 antes de caírem pra Prioridade 1 (porque o Job 0 ou Job 1 já estava lá, e os quanta são curtos, 3,6,10). No entanto, o "FIM DE E/S" constantemente os coloca no topo de alguma fila alta (geralmente Prioridade 1 ou 2, dependendo do estado do sistema e do boost). Os boosts também rolam, reforçando essa promoção.

No fundo, esses parâmetros garantem que jobs que fazem I/O não sejam "punidos" por ficarem parados esperando I/O e sejam tratados como jobs de alta prioridade que precisam de acesso rápido à CPU pra continuar o trabalho deles.

c) O comportamento do escalonador foi justo?

Olha, "justiça" em escalonamento é um conceito meio relativo. Pra esse cenário específico, eu diria:

- **Sim, foi justo para os jobs que fazem muita E/S:** Os dois jobs são idênticos e fazem E/S o tempo todo. O escalonador, com `-S` e `-I`, garante que eles tenham acesso à CPU de forma responsiva depois de cada operação de E/S. Eles competem de forma justa entre si, e a política os favorece pra evitar que sejam deixados de lado, o que é importante pra interatividade em sistemas operacionais.
- **Potencialmente Injusto para jobs que só usam CPU (puramente "CPU-bound"):** Se eu tivesse um terceiro job que só fizesse cálculos (sem nenhuma E/S), ele

provavelmente sofreria uma starvation severa ou demoraria uma eternidade pra terminar. Isso acontece porque os jobs com I/O intenso, com esses parâmetros `-S` e `-I`, seriam constantemente promovidos pra prioridade mais alta e acabariam monopolizando a CPU, deixando pouco tempo pra um job que só calcula e não tem essa "passagem VIP" de volta pro topo da fila.

Então, a "justiça" aqui depende do objetivo. Pra um sistema que prioriza a responsividade de E/S, essa configuração é justa. Mas pra um sistema que quer distribuir o tempo de CPU de forma igualitária entre todos os tipos de jobs, essa configuração seria injusta para os jobs que não fazem E/S.