Escalonamento: Fila com retorno Multi-camada

- *Questão inicial*
- *Fila com retorno Multi-camada (MLFQ)*
- *Regras básicas*
- *Primeira tentativa*
  - *Exemplo 1*
  - *Exemplo 2*
  - *Exemplo 3*
  - *Problemas com a primeira tentativa*
- *Segunda tentativa*
- *Terceira tentativa*
- *Sintonizar o MLFQ*
- *Exemplos de implementação*
- *Questões*

Questão inicial

OBJETIVO:
COMO REALIZAR O ESCALONAMENTO SEM TODO O CONHECIMENTO ?

Como desenhar um escalonador que minimize o **tempo de resposta** para tarefas interativas e, em simultâneo, o **tempo de retorno**, sem que, *à priori*, se conheça a dimensão da tarefa?

THE CRUX:
HOW TO SCHEDULE WITHOUT PERFECT KNOWLEDGE?

How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without *a priori* knowledge of job length?

Fila com retorno multi-nível (MLFQ)

- **Fila com Retorno Multi-nível (*Multi-level Feed-back Queue – MLFQ*)**
  - O problema que se pretende resolver:
    - Otimizar o tempo de retorno T_{ar}(*turn-around t*ime);
      - O SO não sabe quanto tempo demora uma determinada tarefa, conhecimento necessário para aplicar **SJF** e/ou **STCF**.
    - Responder às necessidade de um utilizador interativo, minimizando o tempo de resposta T_r;
      - O **RR** (r*ound-robin)* reduz o tempo de resposta mas não consegue dar resposta ao elevado tempo de retorno.

MLFQ: Regras básicas

- Um número distinto de filas, cada fila com diferentes prioridades:
  - Uma tarefa executável (***ready to run***) encontra-se numa das filas.
- **MLFQ** utiliza as prioridades para decidir que tarefa deve correr num determinado momento:
    - Uma tarefa com a prioridade mais alta (i.e. um trabalho na fila mais alta) é a escolhida para ser executada;
    - Se duas tarefas têm a mesma prioridade elas são executadas em RR.
- Estabelecemos as **regras básicas 1 e 2:**

  Regra 1: Se p*rioridade(A) > prioridade(B)*, A é executado

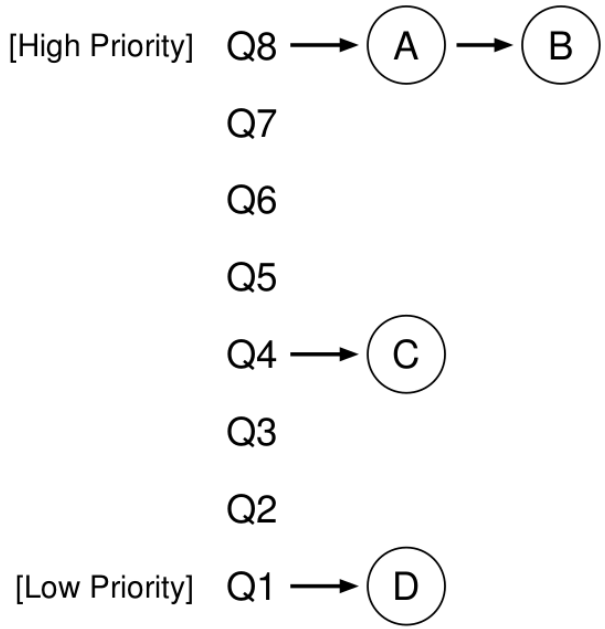
  Regra 2: Se p*rioridade(A) == prioridade(B)*, A e B são executados em **RR** (***round-robin***).

MLFQ: Regras básicas

- A chave do bom funcionamento está na forma como se estabelecem prioridades:
  - Em vez de dar uma prioridade fixa, **MLFQ** atribui prioridades às tarefas baseando-se no *comportamento observado:*
    - Se a tarefa é muito interativa, **MLFQ** mantém prioridade alta.
    - Se a tarefa usa intensamente o **CPU**, **MLFQ** reduz a prioridade.

MLFQ: regras básicas

- Duas tarefas, ***A*** e ***B*** têm a prioridade máxima.
  - O escalonador alterna entre o ***A*** e ***B***.
  - Mas o que acontece ao ***C*** e ao ***D***? Nunca são executados?
- Ter-se-á que mudar a prioridade das tarefas ao longo tempo.

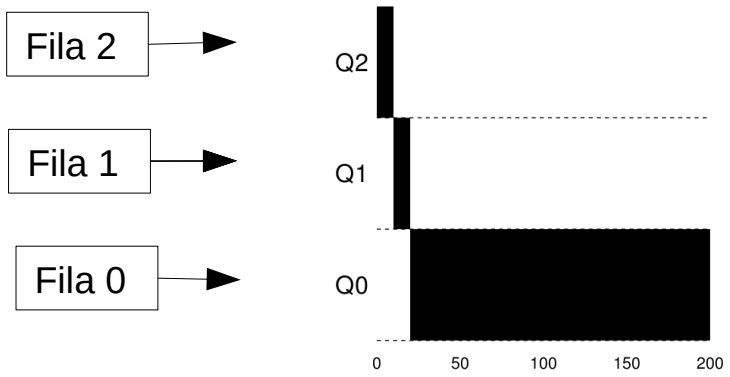


Primeira tentativa: Como mudar prioridade?

- **Regras para mudar prioridade:**
  - **Regra 3**: Quando uma tarefa se torna executável é colocada na fila com maior prioridade;
  - **Regra 4a**: Se uma tarefa utiliza toda a fatia de tempo que lhe é atribuída, a sua prioridade é reduzida *i.e.* muda para uma fila mais a baixo.
  - **Regra 4b**: Se uma tarefa abandona o CPU sem utilizar a totalidade da fatia de tempo atribuída, mantém a sua prioridade *i.e.* mantém-se na mesma fila.

Exemplo 1: Uma única tarefa demorada

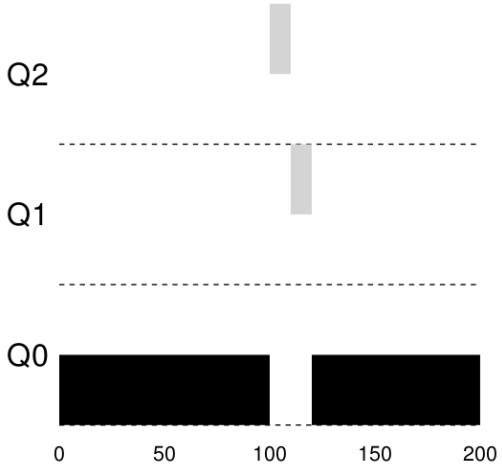
- **Exemplo 1**: Uma única tarefa que usa o CPU de forma demorada
  - A tarefa entra com a máxima prioridade , fila *Q2*.
  - No final da fatia de tempo (10 ms), o escalonador reduz a prioridade e passa para *Q1*.
  - E finalmente muda-se para *Q0* onde permanece.



Posição e tempo de utilização do CPU

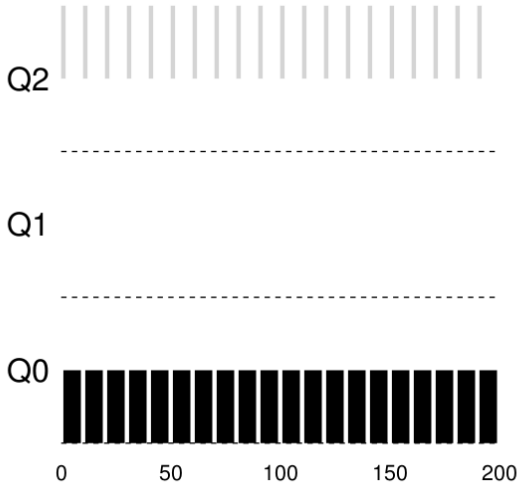
Exemplo 2: Adicionar tarefa de curta duração

- ***A*** é uma tarefa que utiliza CPU de forma intensiva;
- ***B*** é uma tarefa rápida interativa;
  - ***A*** fica executável primeiro. O que acontece quando ***B*** se torna executável?
    - ***A*** corre na fila com menor prioridade.
    - ***B*** está disponível ao fim de T =100.
    - ***B*** muda de prioridade depois de executar a fatia completa de tempo.



Exemplo 3: O que se passa com a E/S?

- Segundo a regra **4b**, se um processo desiste do CPU antes de terminar a sua fatia de tempo, mantém o mesmo nível de prioridade.
  - O objetivo desta regra é o de garantir que tarefas que executam muitas E/S não sejam penalizadas por isso;
  - A figura mostra um exemplo onde a tarefa B necessita de CPU apenas por 1 ms, antes de executar uma E/S, em competição com a tarefa A executa um trabalho demorado em ***background*** (***long-running batch job***)



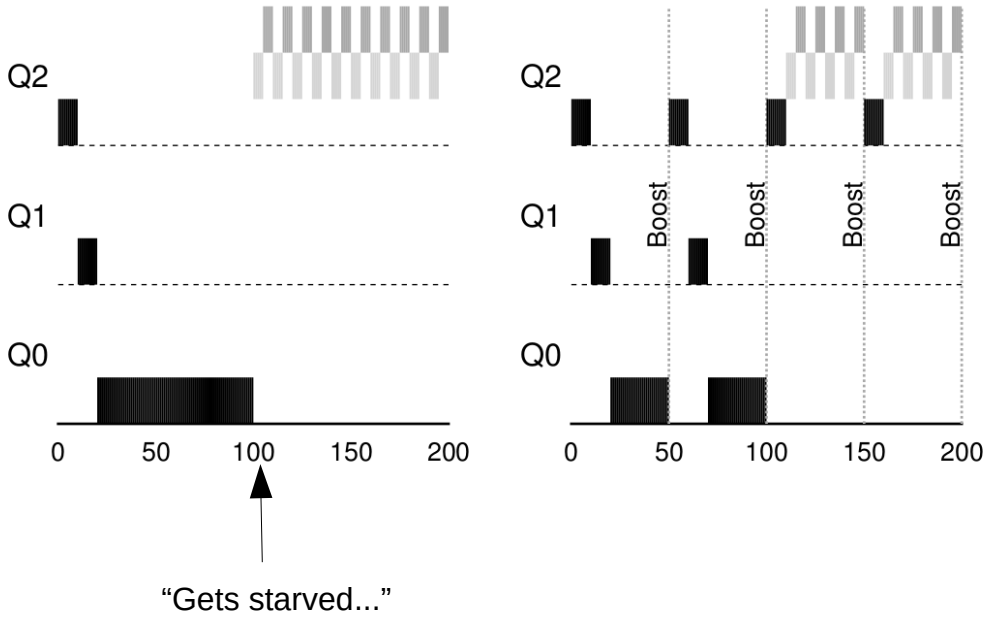
Problemas com primeira tentativa

- O problema da inanição (**starvation**)
  - Se existem demasiadas tarefas interativas, elas poderão consumir todo o CPU e, dessa forma as outras tarefas nunca poderão receber tempo de CPU (ficam inativas).
- Um utilizador pode reescrever o seu programa para enganar o escalonador (***game the scheduler***):
  - Antes de terminar a fatia de tempo, lança uma operação de E/S, renunciando ao CPU.
  - Mantem-se na mesma fila, ganhando mais tempo de CPU.
  - Quando executa bem esta estratégia, pode monopolizar o tempo de CPU.
- Um programa pode mudar de comportamento ao longo tempo:
  - Numa etapa pode usar recursos de CPU.
  - Noutra etapa pode tornar-se interativo:
    - Em determinada etapa pode não ser tratado como uma tarefa interativa!

Segunda tentativa: Um impulso na prioridade

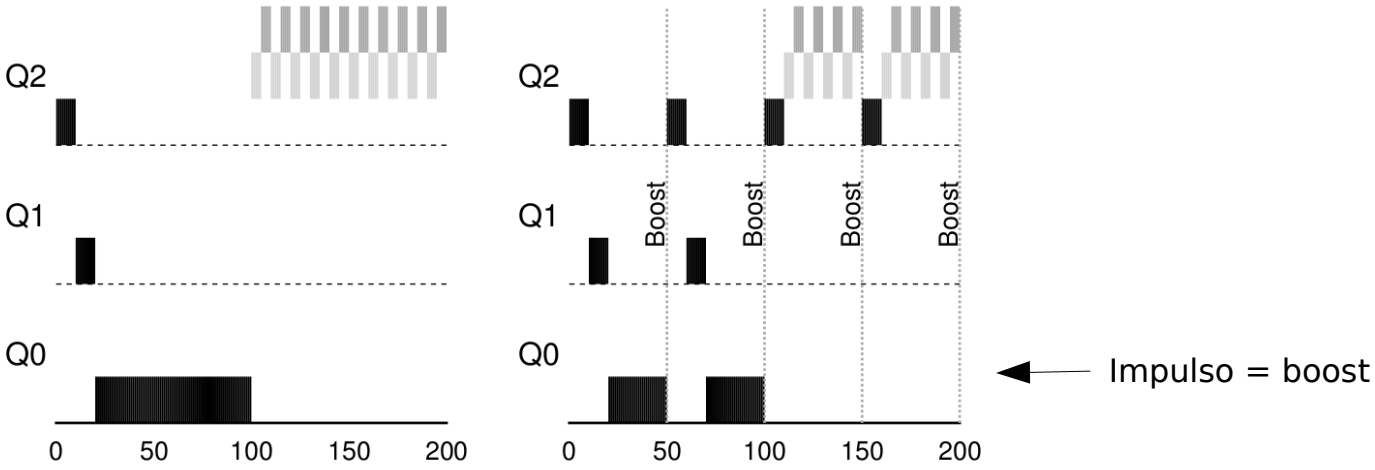
- Modificamos as regras tentando enviar o problema da inanição (*starvation*):
  - Impulsionar, periodicamente, a prioridade de todos os trabalhos no sistema:

    Regra 5: Ao fim de determinado periodo **S**, movem-se todas as tarefas no sistema para a fila com mais prioridade (mais alta).



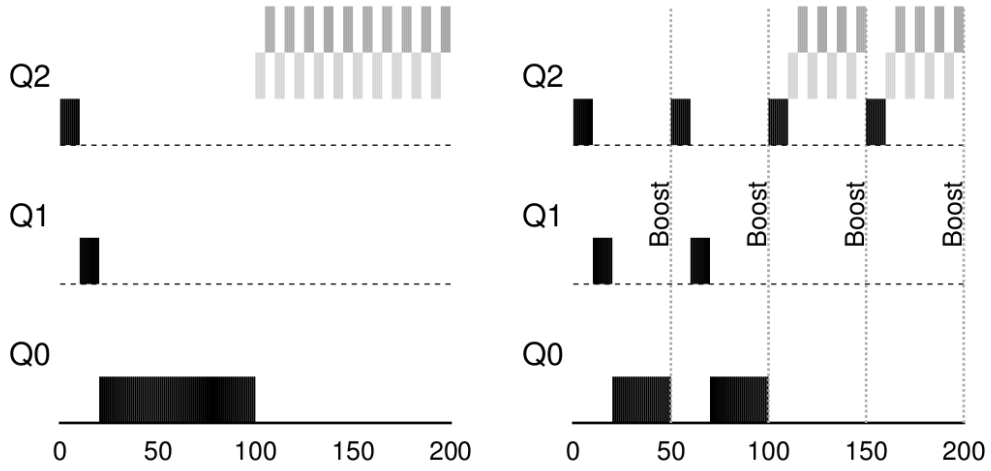
Segunda tentativa: Um impulso na prioridade

- Esta estratégia resolve dois problemas:
  - Deixa de existir inanição.
  - Se uma tarefa se transforma em interativa, o escalonador irá considerá-la como as outras tarefas interativas, a partir do momento que esta se mover para a fila com maior prioridade.



Segunda tentativa: Um impulso na prioridade

- É difícil encontrar valor certo para o impulso:
  - O que acontece se for demasiado grande?
  - E se for demasiado pequeno?



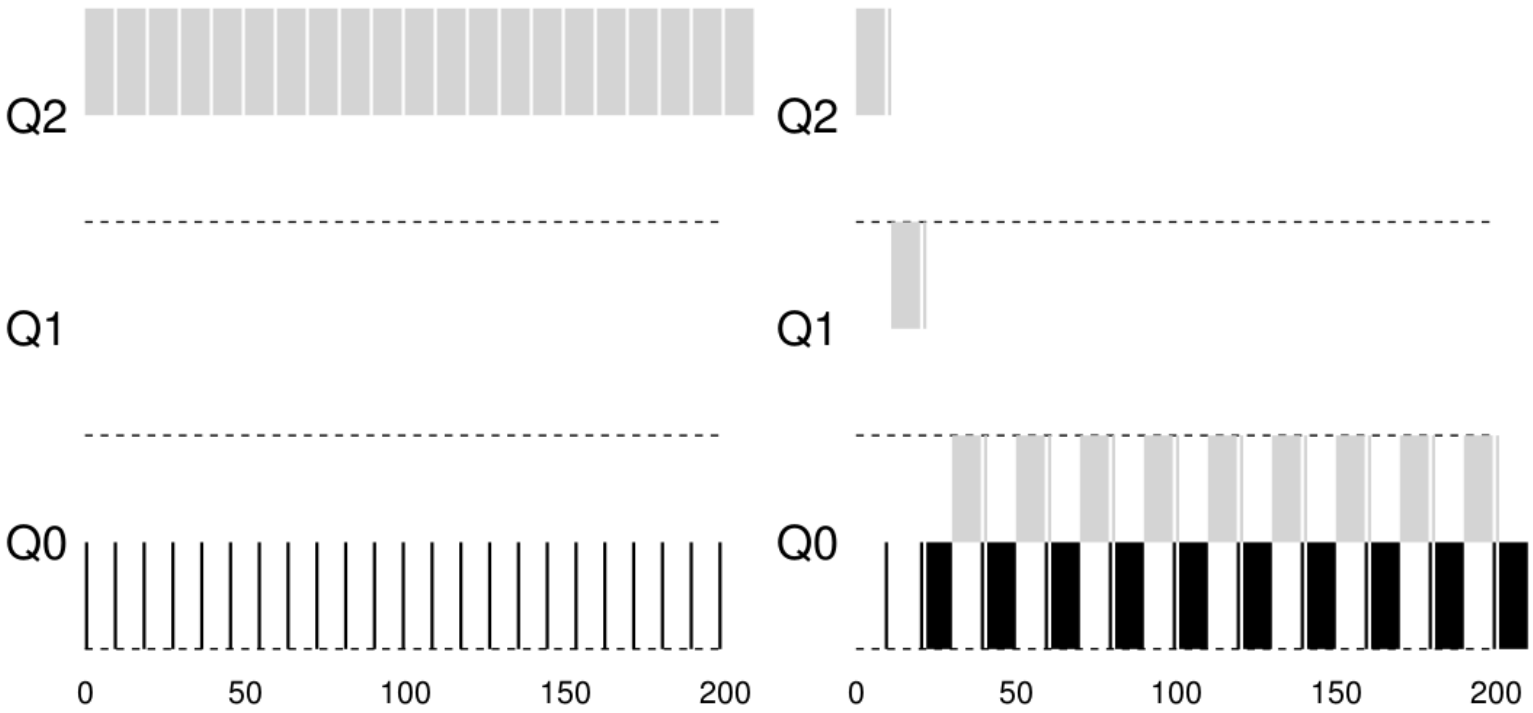
Terceira tentativa: Melhorar a contagem do tempo

- Como garantir que o escalonador não é enganado?
  - O problema está nas tarefas que não completam a sua fatia de tempo, mantendo a sua prioridade (Regras **4a** e **4b**);.
- A solução é realizar a contabilização do tempo de CPU para cada nível do MLFQ:
    - O MLFQ deverá guardar informação sobre qual parte da fatia de tempo é usada por um trabalho (mesmo após ter perdido o CPU).
    - Sempre que uma tarefa termina essa fatia de tempo, é movida para um nível mais baixo.
- Reescreve-se a **Regra 4**:

  Sempre que um trabalho utilize toda a fatia de tempo num determinado nível de prioridade, independentemente de quantas vezes tenha cedido o CPU, a sua prioridade é reduzida, movendo-se para a fila mais a baixo.

Terceira tentativa: Melhorar a contagem do tempo

- À esquerda aplicam-se as **regras 4a 4b** enquanto à direita aplica-se a nova **regra 4**:
  - O trabalho mais claro vai perdendo prioridade à medida que esgota o tempo de CPU;



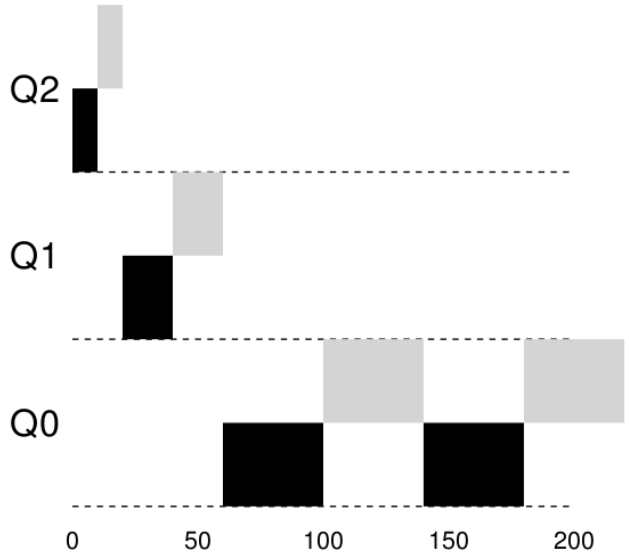
Sintonizar o MLFQ

- **Como parametrizar o escalonador?**
  - Quantas filas deve ter?
  - Quanto deve ser a fatia de tempo em cada fila *?
  - De quanto em quanto tempo a prioridade deve ser impulsionada de forma a evitar *starvation* e ter em atenção mudanças de comportamento dos processos?
- **Variantes do MLFQ**
  - A fatia de tempo varia de dimensão entre filas.
    - **As filas com maior prioridade têm fatias de tempo mais pequenas**
      - Como estão direcionadas para tarefas interativas, faz sentido ter uma fatia de tempo mais pequena (até 10 ms).
    - **As filas com menor prioridade funcionam ao contrário** (até 100 ms)

* variante da MLFQ estudada.

Sintonizar MLFQ

- O exemplo mostra dois trabalhos com diferentes fatias de tempo:
  - 10 ms para a fila com maior prioridade;
  - 20 ms para a fila do meio;
  - 40 ms para a fila com menor prioridade;



Questões

- Explique em que situação um trabalho muda de fila no MLFQ.
- Explique o que entende por inanição de um processo ou trabalho?
- Explique o que pode acontecer se o tempo associado ao “boost” for curto de mais? E se for longo de mais?
- Por que razão lhe parece que atribuir fatias de tempo (time slices) diferentes para filas com prioridades diferentes, é uma boa ideia?
- Suponha que um SO usa um MLFQ que não tem o mecanismo de “boost”. Explique que problemas podem resultar assumindo que tem processos que usam muito o CPU e outros que são muito interativos.
- Explique como é que um programador pode “abusar” do SO, colocando o seu processo em execução em detrimento de outros processos e que mecanismo é adicionado ao MLFQ para evitar esta situação?

Sintonizar o MLFQC

- As constantes VOO-DOO (Lei de Ousterhout)

TIP: AVOID VOO-DOO CONSTANTS (OUSTERHOUT'S LAW)

Avoiding voo-doo constants is a good idea whenever possible. Unfortunately, as in the example above, it is often difficult. One could try to make the system learn a good value, but that too is not straightforward. The frequent result: a configuration file filled with default parameter values that a seasoned administrator can tweak when something isn't quite working correctly. As you can imagine, these are often left unmodified, and thus we are left to hope that the defaults work well in the field. This tip brought to you by our old OS professor, John Ousterhout, and hence we call it **Ousterhout's Law**.

DICA: Evitar as constantes VOO-DOO (lei de Ousterhout)

Evitar, sempre que possível, as constantes voo-doo é uma boa estratégia . No entanto, como no exemplo anterior, nem sempre é possível.

Pode-se tentar garantir que o sistema aprenda ele próprio a escolher um bom valor, mas essa solução também não é simples.

O resultado frequente: Um ficheiro de configuração com parâmetros por defeito que um administrador de sistema ajusta quando necessário. Mas que, muitas vezes não é modificado – sugestão apresentada pelo professor de Sistemas Operativos, John Ousterhout.

Implementação

- Solaris MLFQ
  - Fácil de configurar
  - Fornece conjunto de tabelas que determina exatamente como as prioridades de um processo é alterada ao longo do seu período de existência, qual é a dimensão de um *time slice* e de quanto em quanto tempo a prioridade de uma tarefa sofre um *boost*.
  - Valores por defeito: 60 filas; aumento da dimensão do time-slice de 20 milisegundos até algumas centenas de miliseguntos e prioridade boost cada 1 segundo.
  - FreeBSD (ver https://www.freebsd.org/)

Questões

- Por que razão o MLFQ tem diferentes filas onde coloca os diferentes processos que pretendem ser executados?
- Se um processo muito interativo tivesse uma prioridade baixa, que efeito tinha no comportamento do Sistema Operativo?
- Por que razão é necessário existir o *boost*? O que tem o boost a ver com o efeito de *starvation*?
- Se o tempo de intervalo entre dois impulsos (boost) for pequeno, o que pode acontecer à resposta do Sistema Operativo? E se for muito grande?
- Qual é a vantagem de aumentar o número de filas num MLFQ?
- Por que razão os sistemas têm dimensões de “slice” diferentes nas diferentes filas? As dimensões são maiores para uma fila com mais ou com menos prioridade? Porquê?