FACULDADE DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

# Fundamentos de Sistemas de Operação

LEI - 2023/2024

Vitor Duarte
Mª. Cecília Gomes

1

## Aula 21

- Escalonamento: MLFQ, multiprocessadores
- OSTEP: cap. 8, 10



2

# RR: Vantagens vs Desvantagens

- Sobreposição de IO com CPU
  - Melhora o uso dos recursos e o tempo para terminar um conjunto de processos: **melhor uso de recursos e melhor débito de tarefas**
  - No fim de cada IO o processo passa a ready → pode executar
- RR com time-slice no uso do CPU
  - Permite partilhar o CPU entre todos os processos *ready*, protegendo dos CPU-bound: **melhor *response time***
  - Tem custos com a troca de contextos
- O tempo de conclusão de todas as tarefas tende a ser atrasado
  - Num sistema com vários tipos de processos, os curtos e os IO-bound ficam prejudicados: **pior *turnaround time***
- Não é necessário saber o futuro (as tarefas podem entrar a meio)

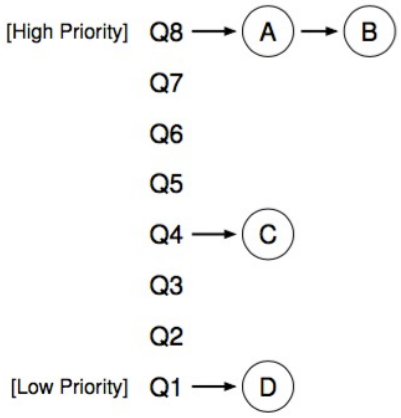


# Prioridade ao IO-bound?

- Objectivos:
  - Melhorar o *turnarround time* dos processos, especialmente os IO-bound (interatividade é IO)
  - Garantir bom *response time* na interação com utilizadores. Evitar injustiças e impedir *starvation*
- Se um processo é IO-bound deve ter prioridade no uso do CPU?
  - De qualquer modo vai deixar de usar o CPU brevemente... → *Shortest Job First*
- Como saber o perfil de cada processo?
  - Não sabemos o futuro mas podemos usar o passado recente como indicador do futuro
  - Se terminou IO dar-lhe oportunidade de passar à frente dos que têm estado a usar o CPU

# Multi-Level Queue

- Em vez de uma fila de processos READY, várias filas, cada uma para diferente prioridade
  - Cada processo READY está numa só fila
- O escalonador atribui o CPU ao primeiro da fila com maior prioridade (usa RR se vários)
  - necessita de políticas para fazer subir e descer processos nas prioridades
- **Pretende-se agora:**
  - **melhorar a resposta para processos interativos/IO**
  - **e o turnaround para processos de curta duração**





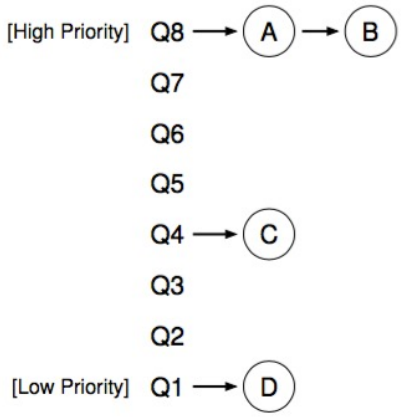

# Feedback para mudar prioridades

- O escalonador vai mudando os processos de prioridade em função do comportamento passado (feedback)
- **Objetivos:**
  - **Melhorar o *turnaround time* para processos de curta duração** *(como em SJF)*
  - **Melhorar a resposta para processos interativos/IO**
  - **Garantir melhor justiça e ausência de *starvation***
- Princípios orientadores:
  - Se usaram o time-slice completamente devem ser CPU bound; senão, devem ser IO bound
  - Diminuir a prioridade dos processos longos e consumidores de muito CPU
  - Manter/aumentar a dos mais curtos ou que se bloqueiam muito e usam pouco CPU

# Multi-Level Feedback Queue

- Regras básicas:

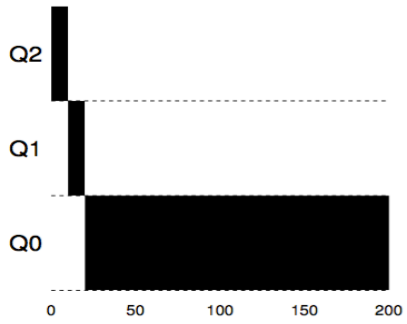
1. O escalonador atribui o CPU ao primeiro da fila com maior prioridade (em RR)
2. Cada processo começa na prioridade máxima
3. Se usa o time-slice completo, **diminui** um nível
4. Se não usa o time-slice completo, **continua** onde está
5. ...quando aumentar a prioridade?



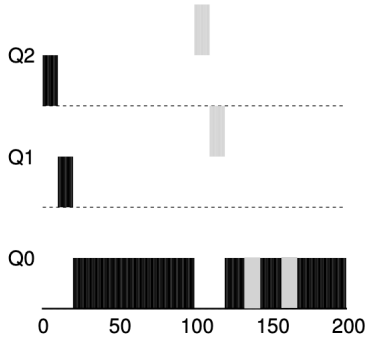



# Exemplo

- Um só processo CPU bound, com 3 prioridades:



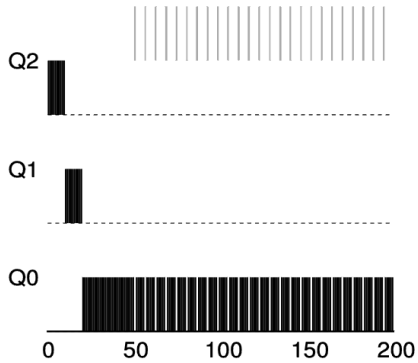
- Entra um novo processo no instante 100:




  - o novo começa com mais prioridade → semelhante a SJF/STCF
  - se também for longo, usa completamente os time-slices, acaba por ficar com a prioridade igual ao 1º → RR



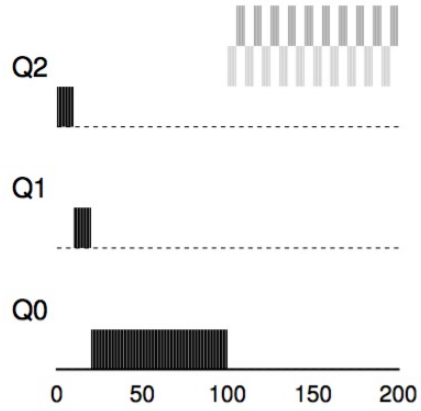
- Entra um processo que não usa todo o time-slice (p.e. IO):




  - mantém a sua prioridade → melhor resposta

## Starvation de CPU-bound vs IO-bound

- Pode levar a *starvation* de processos longos, que usem muito CPU face muitos com IO / interativos
  - também, um programador pode tirar partido disto para ter mais prioridade

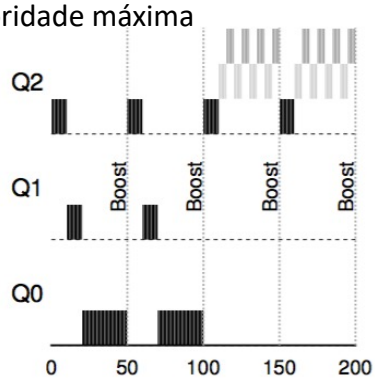





## Aumentando a prioridade

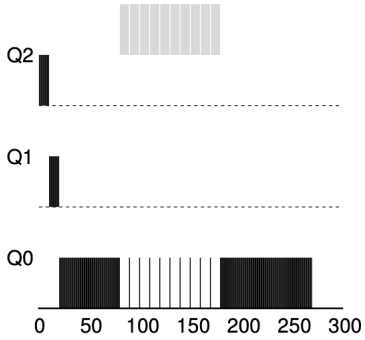
- Como evitar starvation?
- Como lidar com processos longos que mudem para IO-bound/interativos?

5. Nova regra: a cada intervalo (muitos *time-slices*), todos passam à prioridade máxima



# Fazendo batota

- Se contando os time-slices a intervalos fixos
  - o processo faz um IO pequeno mesmo antes do fim do time-slice
- Se contando os time-slices a partir do início de cada escalonamento
  - o processo anterior executa em time-slices alternados, deixando o outro slice para todos os outros processos!

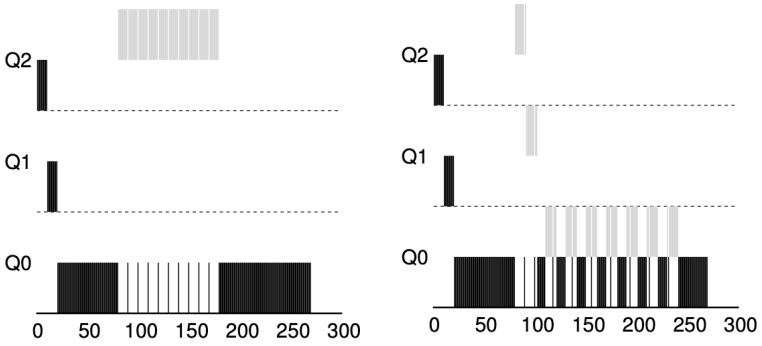





# Evitando batota

- Contabilizando o time-slice como continuação do anterior desse processo

Nova 4. a prioridade é diminuída depois de usar o acumulado de um slice (ou n*t-slice), independente do nº de bloqueios

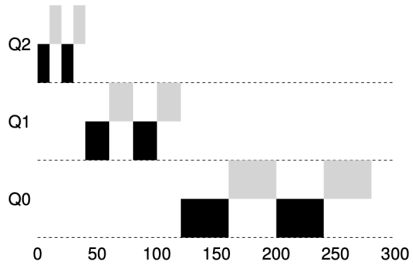


# Mais afinações

- Os IO-bound/interativos precisam de slices menores; os cpu-bound necessitam de mais tempo mas executam menos

→Pode dar *time-slices* maiores em prioridades mais baixas (e vice-versa)

- reduz as trocas de contexto




- Pode reservar a prioridade mais alta para processos especiais
  - processos importantes do SO
  - gestor da interface gráfica



# Multiprocessadores

- Cuidados a ter em multiprocessadores (multicores)
  - procurar ter cada CPU sempre ocupado
  - quando um cpu fica livre escolher um thread/processo READY para executar, como antes
  - o escalonador tem de ser *multithread safe* – pode ter de estar a correr para escolher dois ou mais threads para dois ou mais cpu que ficaram livres
    - pode ter um problema no desempenho do escalonador em arquitecturas com muitos cpu/cores
  - O *overhead* das trocas de contexto pode depender do *thread* escolhido
    - trocas entre *threads* do mesmo processo podem beneficiar da mesma tabela de páginas, conteúdo da TLB e das caches → **cache affinity**

# Uma só fila READY

- Como antes, uma fila (ou uma MLFQ)
  - lidar com concorrência: um só kernel-thread para escalonar todos os cpus; ou vários com controlo de concorrência...
  - problemas de escalabilidade com aumento de cpu/cores
  - pode ter má cache-affinity – os threads/processos podem ir alternado entre cpu, sem nunca aproveitarem caches e TLBs
- Variante: o escalonador contempla que threads do mesmo processo mantenham o mesmo CPU
  - pode estar acessível ao programador:

  `pthread_setaffinity_np`



# Multiplas filas READY

- Tipicamente uma fila por CPU/core
  - escalonador pode ter um kernel-thread por cada fila, para cada cpu
  - melhor de escalabilidade com aumento de cpu/cores
  - procurar que threads do mesmo processo sejam atribuídos à mesma da fila e cpu
  - melhor cache-affinity – os threads/processos tendem a manter o cpu
- Necessita de um mecanismo para equilibrar a carga entre cpus (load balancer)
  - um cpu não pode ficar sem nada para fazer ou ter muito menos processos que os outros
  - Variante: migra processos entre filas/cpu para equilibrar a carga, por exemplo, **work stealing**