

Fundamentos de Sistemas de Operação

Unix Windows NT Netware Macos DOS/V/S Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

*Escalonamento do CPU:
tentando satisfazer “todo o mundo”*

Um mundo menos simples... (1)

□ Assumindo o que de facto acontece...

- Não sabemos previamente o tempo de execução de um processo
- Os processos vão chegando para serem executados e, quando em execução, executam operações de I/O e gastam CPU. Contudo, há aqueles que,
 - fazem muito mais I/O que usam CPU, em grandes volumes: **I/O bound**
 - usam muito mais CPU que I/O, em grande quantidade: **CPU bound**
 - e os que são completamente “mistos” (tipicamente **interactivos**)

Um mundo menos simples... (2)

□ Queremos um escalonador que

- Não prejudique os *interactivos* (*bom TR*) [Já reparou que por vezes (espero que não muitas) carrega nas teclas (*Word?*) e os caracteres demoram a aparecer?]
- Tenha bom débito (*despache rapidamente os processos*) e mantenha o CPU **produtivamente** ocupado)
- Não sacrifique **demasiado** os CPU-bound (*i.e.*, o favorecimento dos *interactivos* são prejudicar seriamente os outros)

O escalonador MLFQv1 (3)

- **Regra 1:** Processos novos começam com prioridade máxima
 $P(A_{novo}) = N-1$
- **Regra 2:** Executa-se sempre o processo de maior prioridade
 $P(A) > P(B) \Rightarrow E(A)$
- **Regra 3:** Se dois processos têm a mesma prioridade, executam-se alternadamente
 $P(A) = P(B) \Rightarrow$ Rodar entre $E(A), E(B)$
- **Regra 4a:** Se a fatia $[T_i, T_{i+1}]$ não foi integralmente consumida
 $P(x_{T_{i+1}}) = P(x_{T_i})$
- **Regra 4b:** senão,
 $P(x_{T_{i+1}}) = P(x_{T_i}) - 1$

O escalonador MLFQv1: problemas

- *Starvation*
 - Se houver muitos processos *interactivos*, os *CPU-bound* podem nunca vir a ser executados
- *Mudança de “perfil”*
 - Se um processo tem um “mix” de fases *interactivas* seguidas de *CPU-bound* (CB), pode levar uma “eternidade” a sair da fase CB e regressar à *interactiva*... e o utilizador desespera
- *Programador “chico-esperto”*
 - Pode programar inserindo “milimetricamente” no código eventos bloqueantes de muito curta duração antes do fim dos timeslices, de forma a manter o processo em alta prioridade ☺

O escalonador MLFQv2

- Resolução do problema de *Starvation* (e...)
 - Ideia base: **periodicamente, todos os processos têm a sua prioridade aumentada até ao máximo**
 - Com esta ideia, resolvem-se de facto dois problemas: o de starvation e o de mudança de comportamento I/CB/.../I/...
 - Mas... com que periodicidade? Todos os sistemas têm parâmetros que têm valores “por omissão” (default values) e permitem o administrador de sistema (no Unix, root) alterar esses valores (tuning system parameters).
A menos que sejam especialistas, essas tentativas podem dar muito maus resultados ☺

O escalonador MLFQv3 (1)

- Resolução do problema do “programador chico esperto”
 - A resolução do problema requer uma contabilização mas rigorosa (apurada ou fina) do uso de CPU
 - Nos versões MLFQv1 e v2, bastava verificar se um processo, quando terminava o timeslice, o tinha gasto por inteiro, ou não. Essa informação servia para alterar, ou não, a sua prioridade; e depois, era descartada (não guardada para futuro).
- Solução (v3)
 - Contabilizar o tempo gasto durante um timeslot. Se consumiu todo, descer a prioridade; se não, guardar o consumo e manter a prioridade. Da próxima vez, adicionar o novo consumo ao guardado e, se este ultrapassou o máximo, então baixar a sua prioridade.

O escalonador MLFQv3 (2)

- **Regra 1:** Processos novos começam com prioridade máxima
 $P(A_{novo}) = N-1$
- **Regra 2:** Executa-se sempre o processo de maior prioridade
 $P(A) > P(B) \Rightarrow E(A)$
- **Regra 3:** Se dois processos têm a mesma prioridade, executam-se alternadamente
 $P(A) = P(B) \Rightarrow$ Rodar entre $E(A), E(B)$
- **Regra 4:** Se o somatório dos consumos é menor que o timeslice, $P(x_{T_{i+1}}) = P(x_{T_i})$; senão, $P(x_{T_{i+1}}) = P(x_{T_i}) - 1$ e somatório $\leftarrow 0$
- **Regra 5:** Com uma periodicidade S , $P(x) = N - 1$

MLFQ: conclusões (1)

□ *Pontos Positivos*

- Não é necessário um oráculo
- Jobs interactivos têm desempenho idêntico ao oferecido por um SJF, e jobs CPU-bound progridem (apesar de menos prioritários)

□ *Pontos menos positivos*

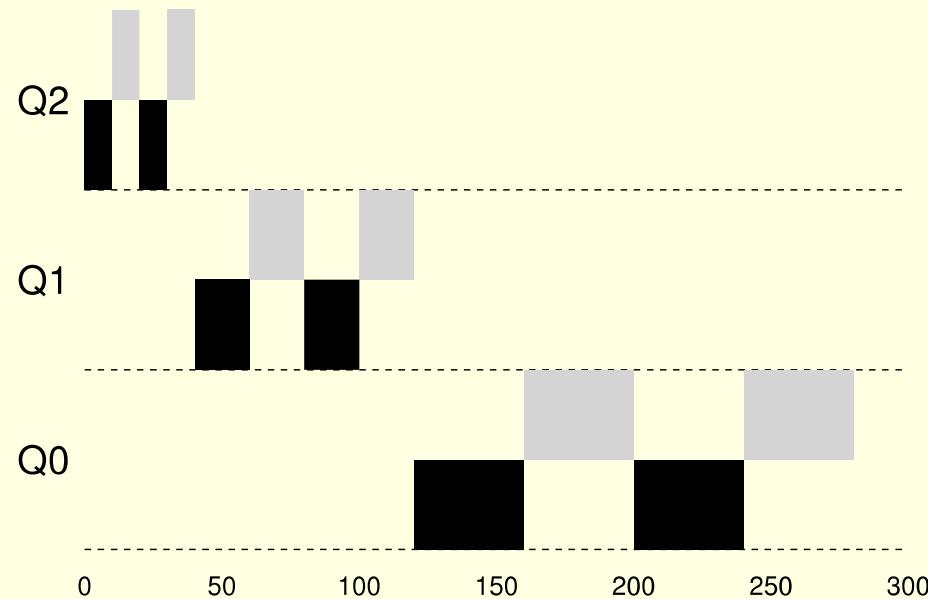
- Quantas filas?
- Que periodicidade para o boost (aumento) da prioridade?

MLFQ: conclusões (2)

□ *Pontos menos positivos*

(continuação)

- *Timeslices iguais em todas as filas? [alguns SOs usam um timeslice maior para as menores prioridades]*



MLFQ: conclusões (3)

□ “Tuning” sofisticado

- *Todos os sistemas têm parâmetros que têm valores “por omissão” (default values) e permitem o administrador de sistema (no Unix, root) alterar esses valores (tuning system parameters).*

A menos que se tenha m conhecimento aprofundado, essas tentativas podem dar muito maus resultados ☺

□ Variantes do MLFQ usadas em vários SOs

- *Windows, Sun/Oracle Solaris, BSD/FreeBSD Unix*

MLFQ: “Prática em papel” + Demo

- **Simulador `mlfq.py`** (*download do site do livro OSTEP*)
 - Desenhe um “diagrama temporal” que mostre a evolução de 3 processos escalonados num MLFQv3 com as seguintes características:
 - 3 filas, timeslices iguais em todas, 10 ms
 - Sem boost
 - Allotment para descida de prioridade = 1 (i.e., 10ms)
 - Processos chegam todos ao mesmo tempo
 - Job 0: duração=17ms, I/Os a cada 7ms
 - Job 1: duração= 8ms, I/Os a cada 3ms
 - Job 2: duração=10ms, I/Os a cada 4ms
 - Duração dos I/Os: 5ms

MLFQ: “Prática em papel”+ Demo

