

# Fiscalização da fila

Veja como usar o Dtrace para observar as classes de escalonamento do OpenSolaris.

por Marcelo Arbore e José Damico

ANÁLISE



Os primeiros computadores nasceram com o objetivo de fazer cálculos repetitivos, que demorariam tempo demais se feitos por seres humanos. Nesta época, o algoritmo de escalonamento era simples: execute o próximo trabalho ao fim do atual. Hoje, os sistemas computacionais compartilham tempo, usuários e serviços. É comum encontrar situações em que temos dois ou mais processos competindo pelo mesmo processador. Neste caso, cabe ao sistema operacional decidir a ordem em que os processos serão executados. O componente do sistema operacional responsável por esta tarefa é chamado de escalonador, e o algoritmo utilizado é chamado de algoritmo de escalonamento.

Algumas das funções fundamentais dos sistemas operacionais modernos são o gerenciamento e o escalonamento dos processos dentro

dos processadores disponíveis. O objetivo principal é manter a justiça. Entretanto, como na política, existe divergência de opiniões, pois nos deparamos com necessidades, objetivos e métricas diferentes. Por este motivo, existe uma extensa variedade de algoritmos disponíveis. Este artigo analisa algumas das implementações clássicas existentes no sistema operacional de código aberto OpenSolaris por meio da ferramenta de instrumentação dinâmica *Dtrace*.

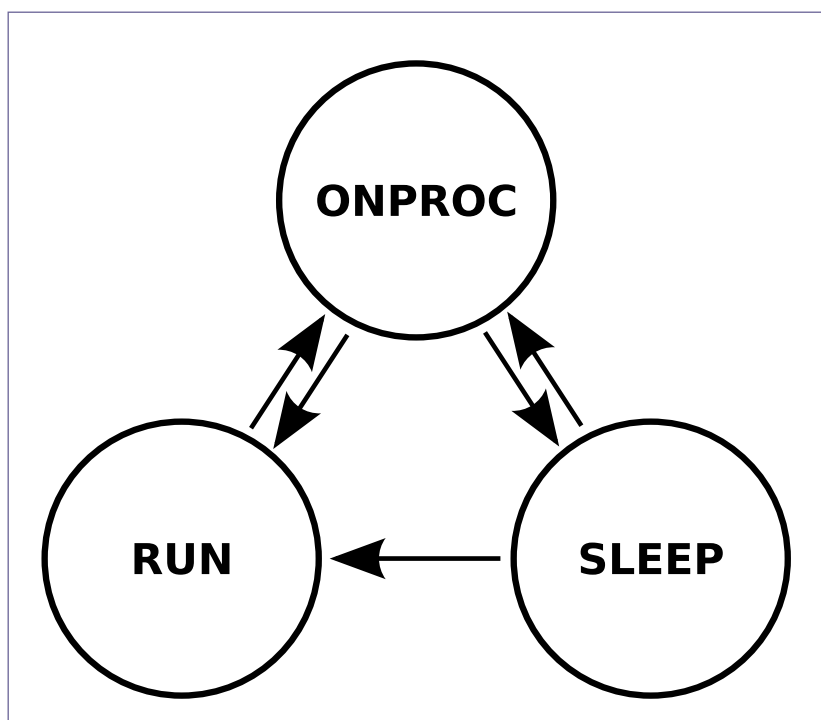
## Observabilidade

Com o aumento do tamanho e da complexidade dos sistemas operacionais modernos, torna-se cada vez mais complicada a análise, a depuração ou mesmo a compreensão didática do que está ocorrendo no que se refere ao kernel e aos usuários. A maioria dos softwares existentes para depuração se mostram

ineficientes, principalmente pelo custo de desempenho, e também incompletos, pelas restrições de segurança que o kernel impõe aos seus processos e, conseqüentemente, aos depuradores. A leitura do código-fonte é, além de complexa, pouco eficiente do ponto de vista prático.

Existem diversas ferramentas de observabilidade em sistemas Unix, mas elas sofrem, além dos problemas já listados, de baixa flexibilidade (servem para propósitos específicos e apresentam saídas padronizadas) e baixa interoperabilidade (não conversam entre si).

Em 2002 foi lançado o primeiro protótipo da ferramenta *DTrace*. O *Dtrace* não sofre as restrições de segurança dos métodos convencionais e tem bom desempenho porque está diretamente inserido no kernel do sistema operacional. Utilizando-se de blocos dinamicamente



**Figura 1** Fluxo de estados de *threads* no sistema OpenSolaris.

mente alocáveis, ele não desperdiça preciosos ciclos de processamento quando desligado. Isto permite um extenso número de pontos de instrumentação (atualmente na ordem de dezenas de milhares) que podem ser ativados individualmente. Desta maneira, observa-se padrões globais ou específicos do sistema de uma maneira uniforme.

Por meio de scripts na linguagem de alto nível *D* (semelhante a *C*), é possível fazer perguntas arbitrárias ao sistema sobre praticamente qualquer padrão de comportamento deste. O *DTrace* foi implementado inicialmente no Solaris, e hoje já está portado para OpenSolaris, MacOS, FreeBSD e existem projetos para outros sistemas.

### Listagem 1: Script para verificar alterações de prioridade

```
#!/usr/sbin/dtrace -s
sched:::change-pri
{
    @[stringof(args[0]->pr_clname)] =
        lquantize(args[2] - args[0]->pr_pri, 50, 50, 5);
}
```

### Listagem 2: Saída sem script fominha

```
value  ---- Distribution ---- count
-15   |                               0
-10   | @@@@@@@@@@                    3
-5    |                               0
0     |                               0
5     |                               0
10    | @@@@@@@@@@@@@@@@             0
15    |                               0
```

## Escalonamento no OpenSolaris

No OpenSolaris, cada *thread* possui uma prioridade global, e isto determina quão brevemente ela, dentre todas as *threads* executáveis do sistema, será executada.

Tipicamente, uma *thread* encontra-se nos estados *RUN*, *ONPROC* ou *SLEEP* (figura 1). *RUN* representa prontidão para execução (esperando a sua prioridade ser a maior na fila de execução). *ONPROC* simboliza as *threads* selecionadas para execução no presente momento e *SLEEP* é o estado no qual elas aguardam um evento de sincronia, como o término de uma operação de entrada/saída.

O escalonador é responsável por gerenciar todos os estados de todas as *threads* do sistema, assim como a troca sincronizada entre eles. As principais funções do escalonador do OpenSolaris são:

- ◆ gerência de filas: inserir e remover *threads* das filas de execução e de *SLEEP*;
- ◆ seleção de *thread*: selecionar, dentre todas as *threads* no estado *RUN*, qual será a próxima a ser executada;
- ◆ seleção do processador: escolher em qual processador uma *thread* será executada;
- ◆ troca de contexto: trocar o ambiente de execução de um processador para que este possa executar uma *thread* diferente.

Diferentes situações requerem diferentes algoritmos de escalonamento. Isto acontece porque as áreas de aplicação possuem objetivos distintos. Não existe um algoritmo único otimizado para todos os sistemas. Os algoritmos de escalonamento implementados no OpenSolaris são:

- ◆ Timeshare (TS, compartilhamento de tempo)
- ◆ Interactive (IA, interativo)
- ◆ Fair Share (FSS, parcela justa)

- ▶ Fixed Priority (FX, prioridade fixa)
- ▶ Real Time (RT, tempo real)
- ▶ System (SYS, sistema)

Analisaremos agora o comportamento de alguns destes algoritmos. A referência completa pode ser encontrada na documentação oficial.

## Timeshare (TS)

A ideia central do timeshare é tentar compartilhar o tempo da CPU de maneira uniforme entre as threads. É claro que cada processo possui a sua demanda própria por processamento. Por este motivo, os ajustes de prioridade são feitos com base no tempo gasto pela thread à espera do processador ou em execução.

O TS é estruturado com múltiplas filas de execução (uma por prioridade). Dentro da fila, os processos comportam-se de maneira similar ao algoritmo *Round-Robin* (rodízio simples).

Este algoritmo de escalonamento possui um tempo inicial padrão – chamado de *quantum* – que ele atribui a todos os processos. Por um lado, se um processo tem a tendência de usar seu quantum inteiro, o escalonador se encarrega de reduzir sua prioridade. Por outro lado, se o processo tende a não usar seu quantum por completo, cessando o processamento ou parando em algum mecanismo de sincronia, o algoritmo timeshare aumenta a sua prioridade com o intuito de valorizar o pedido deste por processamento.

Suponhamos um processo faminto por processamento, digamos:

```
$ while true; do let a=0; done
```

É fácil perceber que este script na linha de comando entra em *loop* infinito enquanto faz uma atribuição inútil.

Vejamos agora um script em linguagem D (**listagem 1**), retirado da

### Listagem 3: Saída com script fominha

| value | ----- Distribution ----- | count |
|-------|--------------------------|-------|
| -15   |                          | 0     |
| -10   | @@@@@@@@@@@@@@@@         | 120   |
| -5    |                          | 0     |
| 0     | @@@@@@@@@@@@             | 49    |
| 5     |                          | 0     |
| 10    |                          | 0     |
| 15    |                          | 0     |
| 25    |                          | 0     |
| 30    |                          | 0     |
| 35    |                          | 0     |
| 40    |                          | 0     |
| 45    | @@@@                     | 25    |
| >=50  |                          | 0     |

documentação, que verifica todas as alterações de prioridade nas threads do sistema.

O script agrupa pelo nome da classe – no nosso caso, TS – e pede para agregar linearmente o valor resultante da diferença entre as prioridades nova e velha. Ao acionar o script da **listagem 1** por alguns segundos, com o script fominha anterior em execução, são gerados resultados como os das **listagens 2 e 3**.

Eles mostram uma distribuição do valor da variação de prioridade (à esquerda) e o número de ocorrências daquela mudança (à direita).

Fica visível que o escalonador age no sentido de diminuir a prioridade de threads no sistema. O grupo de controle (execução sem o script fominha) levanta a suspeita de que o culpado pelas mudanças de prioridades é o processo do terminal que está em loop infinito.

### Listagem 4: Script para contar o tempo

```
01 #!/usr/sbin/dtrace -s
02
03 #pragma D option quiet
04
05 long int tson, tsoff;
06
07 sched::on-cpu
08 /pid == $1 && tid == $2/
09 {
10     trace(probename);
11     tson = timestamp;
12     printf("    tempo fora = %d nsec\n", timestamp-lsoff);
13 }
14 sched::change-pri
15 /args[1]->pr_pid == $1 && args[0]->pr_lwpid == $2/
16 {
17     trace(probename);
18     printf("    delta pri = %d\n", args[2]-args[0]->pr_pri);
19 }
20 sched::off-cpu
21 /args[1]->pr_pid == $1 && args[0]->pr_lwpid == $2/
22 {
23     trace(probename);
24     printf("    tempo dentro = %d nsec , pri = %d\n", timestamp
25     -tson, args[0]->pr_pri);
26     tsoff = timestamp;
26 }
```

**Listagem 5: Saída de listagem 4**

```

...
off-cpu    tempo dentro = 16649243 nsec , pri = 59
on-cpu    tempo fora = 91681 nsec
change-pri delta pri = -9
off-cpu    tempo dentro = 41670399 nsec , pri = 50
on-cpu    tempo fora = 97571 nsec
off-cpu    tempo dentro = 9318139 nsec , pri = 50
on-cpu    tempo fora = 79198 nsec
off-cpu    tempo dentro = 13291416 nsec , pri = 50
on-cpu    tempo fora = 177805 nsec
off-cpu    tempo dentro = 5495730 nsec , pri = 50
on-cpu    tempo fora = 81745 nsec
change-pri delta pri = -10
off-cpu    tempo dentro = 10681020 nsec , pri = 40
on-cpu    tempo fora = 116827 nsec
off-cpu    tempo dentro = 19035494 nsec , pri = 40
on-cpu    tempo fora = 84719 nsec
off-cpu    tempo dentro = 10120625 nsec , pri = 40
on-cpu    tempo fora = 87395 nsec
off-cpu    tempo dentro = 25661734 nsec , pri = 40
on-cpu    tempo fora = 89414 nsec
change-pri delta pri = -10
off-cpu    tempo dentro = 2488795 nsec , pri = 30
on-cpu    tempo fora = 90233 nsec
off-cpu    tempo dentro = 232736969 nsec , pri = 30
on-cpu    tempo fora = 95045 nsec
off-cpu    tempo dentro = 7080789 nsec , pri = 30
on-cpu    tempo fora = 85251 nsec
off-cpu    tempo dentro = 10344521 nsec , pri = 30
on-cpu    tempo fora = 164332 nsec
off-cpu    tempo dentro = 11108804 nsec , pri = 30
on-cpu    tempo fora = 107210 nsec
change-pri delta pri = -10
off-cpu    tempo dentro = 68544355 nsec , pri = 20
...

```

Se quisermos observar com mais detalhes essa thread que muda o padrão de comportamento do escalonador, temos que usar o script em D da **listagem 4** para verificar os tempos de execução e inatividade desta

thread, e também as mudanças de prioridade aplicadas pelo algoritmo de escalonamento.

Ao executar esse script sobre a nossa thread sedenta por processamento, a saída gerada é semelhante

**Listagem 6: Script de teste de IA**

```

01 #!/usr/sbin/dtrace -s
02
03 #pragma D option quiet
04
05 0sched:::change-pri
06 /args[0]->pr_cname == "IA" && args[1]->pr_pid == $1 && args[0]
->pr_lwpid == 1/
07 {
08     printf("Priority:%d + (%d) = %d\n",args[0]->pr_pri, args[2]
->args[0]->pr_pri, args[2]);
09 }

```

te à **listagem 5**. Nela, é possível ver claramente que o algoritmo de escalonamento age sobre a thread em questão. A **listagem 5** demonstra que o processo permanece muito tempo dentro da CPU (na ordem de dezenas de milissegundos) em relação ao tempo fora dela (na ordem de dezenas ou centenas de microssegundos). Como esperado, o algoritmo de timeshare diminui a prioridade desta thread.

**Interactive (IA)**

Similar ao timeshare, esta classe adiciona um mecanismo que aumenta a prioridade das threads relacionadas à janela ativa do desktop. O algoritmo IA tem o intuito de ser usado em laptops e desktops para aumentar a interatividade e melhorar o tempo de resposta (isto é, reduzir a latência) do usuário.

Para demonstrar este mecanismo, foi utilizado o script D da **listagem 6**, que imprime as alterações de prioridade de um determinado processo.

Repare que, dentre todos os eventos de alteração de prioridades, o script seleciona aqueles que são da classe IA, cujo id do processo é passado como argumento e cujo id da thread é igual a um (para evitar comprometer a análise com o comportamento de threads distintas).

Ao passar como argumento o id de um processo que está aberto no desktop, alterando o foco para este processo e voltando (duas vezes), é gerada uma saída como a da **listagem 7**.

Repare que, ao colocar o foco na janela referente ao processo, o escalonador se encarrega de elevar a prioridade deste até o nível máximo (prioridade 59). Existem dois momentos, de acordo com a saída, em que a thread atinge o limite superior. Estes acontecem no momento em que colocamos a janela em foco. Desta maneira, o algoritmo IA consegue privilegiar os processos

que possuem janelas sendo manuseadas por usuários, gerando uma sensação de respostas instantâneas, ou ao menos com uma latência sensivelmente menor.

## Fixed Priority (FX)

Existem casos em que a preempção, isto é, a retirada forçada de uma thread do processador por outra de maior prioridade, não é desejada. Normalmente não há necessidade de baixa latência, sendo o tempo total de execução o fator crítico.

O FX permite isto porque, como o próprio nome indica, não há mudanças de prioridade nos processos. Não existem ciclos de processamento sendo desperdiçados para gerência de prioridades, manutenção intensiva de filas e trocas de contexto desnecessárias. Desta maneira, os processos são serializados e executados um após o outro, terminando o lote de trabalho em menor tempo.

Ao se executar os scripts anteriores em um ambiente configurado com o algoritmo FX, é possível verificar que sua prioridade não muda, além de poder ver claramente o momento em que cada processo entra no processador, é executado por inteiro e finalmente sai.

## E no Linux?

O Linux, como o OpenSolaris, herda do Unix as principais políticas de escalonamento de tarefas com o objetivo de oferecer máxima confiabilidade na execução de cada thread e processo. Para tanto, os seguintes problemas precisam ser resolvidos: gerenciamento dinâmico de prioridade das tarefas, tipificação e controle de processos em lote, interativos e em tempo real, com a consideração se os mesmos usam intensa alocação de I/O ou de CPU.

O tempo de resposta de cada chamada de sistema e o fluxo de processos assinalados ao sistema operacional

### Listagem 7: Saída da listagem 6

```
Priority:39 + (10) = 49
Priority:49 + (10) = 59
Priority:59 + (0) = 59
Priority:59 + (-20) = 39
Priority:39 + (-10) = 29
Priority:29 + (15) = 44
Priority:49 + (5) = 49
Priority:49 + (10) = 59
Priority:59 + (-10) = 49
Priority:49 + (10) = 59
Priority:59 + (0) = 59
Priority:59 + (-20) = 39
Priority:39 + (-10) = 29
```

deverem ser rápidos o suficiente para evitar o congelamento da execução ou tempos de espera muito altos entre processos.

Por fim, o suporte à implementação SMP é obrigatório para que em máquinas com mais de um núcleo ou processador se possa escalar processos paralelamente, de forma eficaz.

Tanto o OpenSolaris quanto o Linux implementam com sucesso todos esses requisitos, contudo existem algumas diferenças na implementação do sistema do pingüim, especialmente a partir de sua versão 2.6, que primeiramente implementou o escalonador  $O(1)$  e depois, a partir da versão 2.6.23, incluiu o *Completely Fair Scheduler* (CFS, escalonador completamente justo).

## Conclusão

Como dito anteriormente, uma das funções principais de qualquer sistema operacional moderno é a gestão de processos, threads e processadores. O entendimento prático dos algoritmos é essencial para a seleção correta, assim como para o uso didático. Para isso, o Dtrace é uma ferramenta adequada: primeiramente por permitir trabalhar com o kernel funcional e ativo; em segundo lugar pela flexibilidade que os scripts D permitem; em terceiro, pela facilidade e acessibilidade da sintaxe de D, muito semelhante ao popular C. O Dtrace também se mostrou, pelos motivos levantados anteriormente e por ser aberto, adequado para o ambiente acadêmico de ensino didático de sistemas operacionais e de computação em geral. ■

### Sobre o autor

**Marcelo Arbore** ([marbore@br.ibm.com](mailto:marbore@br.ibm.com)) é engenheiro de software da IBM. Formado em engenharia pela Poli-USP, tem experiência com soluções sobre as plataformas Solaris, Linux e AIX.

**José Damico** ([jdamico@br.ibm.com](mailto:jdamico@br.ibm.com)) é engenheiro de software da IBM, onde trabalha com a solução de data warehouse, Smart Analytics. É também especialista em Linux e Open Source, desenvolve e suporta diversos projetos no SourceForge e no GoogleCode.

### Gostou do artigo?

Queremos ouvir sua opinião. Fale conosco em [cartas@linuxmagazine.com.br](mailto:cartas@linuxmagazine.com.br)

Este artigo no nosso site:  
<http://lnm.com.br/article/3012>

