Grande aliado

O SystemTap ajuda a identificar problemas de kernel em ambientes de produção. Se você precisa verificar por que o sistema está se comportando mal em uma carga de trabalho real, experimente esta poderosa ferramenta.

por Steve Best

O Linux já foi considerado um sistema fraco para varrer e monitorar recursos, mas a chegada dos frameworks *System Tracing* e *Profiling*, ou SystemTap [1], fechou essa lacuna. O SystemTap se desenvolveu e amadureceu muito ao longo dos últimos cinco anos e agora está incluído em diversas distribuições Linux. Uma interface gráfica baseada no Eclipse [2] está disponível separadamente como ferramenta de análise de sistemas e script IDE para desenvolvimento.

O SystemTap permite que administradores de sistemas e desenvolvedores reúnam informações em tempo real de um kernel Linux em execução. A ferramenta de rastreamento do SystemTap simplifica a tarefa de varrer o kernel ao colher dados que podem ser analisados separadamente para otimizar o desempenho ou resolver um problema. Com o SystemTap, você não precisa reconstruir, reinstalar o kernel e reinicializá-lo para obter dados vitais. Esse recurso de evitar a reinicialização é poderosa, pois muitos administradores não gostam de fazê-lo em um sistema em produção.

A ferramenta SystemTap baseia-se em tecnologias e pontos de rastreamento Kprobes, disponíveis no kernel Linux. E com o SystemTap, os usuários ganham mais flexilidade para agregar e visualizar dados de tracepoint (pontos no código que definem recursos presentes). O SystemTap inclui uma linguagem de scripts como o awk [3], e uma interface de linha de comando simples ativa as áreas que precisam de monitoramento. A ferramenta é útil para monitorar a saúde geral de um sistema ou determinar o que este está fazendo em um determinado momento. Às vezes você precisa entender o que o sistema está fazendo em tempo real.

Em vez de implementar uma grande quantidade de outras ferramentas, o SystemTap sozinho é capaz de mostrar onde está o problema de memória, de sistema de arquivos, da rede ou do agendamento de tarefas e adicionar ou ativar probes. Ferramentas como `ps`, `top`, `sar`, `vmstat`, `iostat`, `free`, `uptime`, `mpstat`, `pmap`, `netstat`, `iptraf`, `tcpdump`, `strace` e `lsof` são ótimas, mas não possuem a flexibilidade de um script SystemTap, que você pode mudar em tempo real para obter dados adicionais do kernel. O recurso chave do SystemTap consiste em capturar somente o dado necessário. A coleta de dados começa quando você habilita um ponto de prova (*probe point*) e para quando você o desabilita.

A maioria das ferramentas de monitoramento foca em um pequeno conjunto de estatísticas e formata o retorno dos resultados para um sentido, seja no modo interativo ou no modo de log, mas não em ambos. O SystemTap permite monitorar qualquer conjunto de subsistemas ao adicionar pontos de provas para um ou mais subsistemas chaves do kernel. Pode ser que você precise apenas destacar picos ou outras anormalidades para identificar um problema de desempenho. Mostrarei como é fácil adicionar um ponto de prova e visualizar os dados dentro do kernel.

O SystemTap tem uma interface de linha de comando chamada `stap`. A **figura 1** mostra os passos para transformar o script `.stp` em código C: compile o código C no módulo kernel (arquivo `.ko`), carregue o módulo dentro do kernel, execute o módulo e interrompa-o enquanto os botões [**Ctrl**] + [C] estiverem pressionados, ou o módulo decidirá quando está pronto.

Requisitos para instalação

Para começar, certifique-se de que o sistema no qual você executa o probe do SystemTap tem os pacotes apropriados para isso. Para encontrar localizações dinamicamente a fim de inserir , argumentos das funções provadas e variáveis no escopo do ponto de prova, a ferramenta usa o padrão Dwarf para fazer o debug de informações geradas pelo compilador. Os exemplos do SystemTap mostrados neste artigo são executados sob o Red

Quadro 1: Tapsets

Os tapsets são scripts que encapsulam conhecimento sobre um subsistema de kernel em probes pré-escritos e funções que podem ser usadas com outros scripts. Os tapsets são semelhantes às bibliotecas para programas C. Eles escondem os detalhes subjacentes do kernel enquanto expõem as informações chaves necessárias para visualizar e monitorar um ou mais aspectos do kernel. Os tapsets são normalmente desenvolvidos por especialistas em kernel. Um tapset normalmente expõe um alto nível de informações e transições de estados de um subsistema. Os usuários que precisam de dados de baixo nível normalmente dispensam os tapsets e escrevem scripts personalizados para atacar um problema específico.

Hat Enterprise Linux v5.5, então instalei os seguintes pacotes: SystemTap, kernel debug info e kernel development.

Se você estiver em outra distribuição, terá que verificar se os pacotes estão instalados e instalar todos antes de começar a usar o SystemTap.

Dados contextuais

O SystemTap oferece uma gama de dados contextuais que podem ser formatados com facilidade e geralmente aparecem como chamadas de funções dentro do manipulador. Suporta também uma variedade de eventos pré-formatados e vem com uma biblioteca de scripts. Cada script é chamado de tapset (**quadro 1**). Confira na página `stapfuncs` essas funções e outras definidas na biblioteca tapset. Aqui vão algumas das mais comuns:

- `tid()`: identificador da sequência em execução
- `pid()`: identificador do grupo de tarefas da sequência em execução
- `execname()`: nome da sequência em execução
- `gettimeofday_ns()`: número de nanossegundos desde a época Unix

Usarei essas funções nos scripts de exemplo incluídos no artigo. Os valores obtidos podem ser sequências de caracteres ou números. A função `print()` pode aceitar um argumento sozinho, ou pode-se usar a função no estilo do C como em `printf()`, cujo argumento de formatação pode incluir `%s` para uma sequência de caracteres ou `%d` para um número. A função `printf()` e outras aceitam argumentos separados por vírgulas.

A forma mais simples de realizar probes é a que identifica um evento, o que é feito inserindo estrategicamente um argumento print dentro do kernel. Esse processo é muitas vezes considerado o primeiro passo para resolver um problema de kernel: explorar uma chamada e ver o histórico do que acontece ao pedir para o SystemTap fornecer informações para cada evento. Para expressar esse processo na linguagem de scripts do SystemTap, é necessário especificar onde fazer as provas e o que deve ser retornado como informação.

Listagem 1: Syscall

```
01 global time_open
02 probe syscall.open {
03   time_open[tid()]
➥ = gettimeofday_ns()
04   printf( "syscall_open
➥ %s -pid %d args
➥ (%s)\n",execname(), pid(),
➥ argstr)
05 }
06
07 probe syscall.open.return {
08   if(time_open[tid()]) {
09     printf( "sys_open spend
➥ %d ns to
➥ complete\n",gettimeofday_ns()
➥ - time_open[tid()])
10     delete time_open[tid()]
11   }
12 }
```

Listagem 2: Resultados da sonda

```
01 sys call_open gedit -pid
➥ 3976 args ("/usr/share//mime/
➥ aliases", O_RDONLY|O_LARGEFILE)
02 sys_open spend 9503 ns
➥ to complete
03 sys call_open gedit -pid
➥ 3976 args ("/usr/share//mime/
➥ subclasses",
➥ O_RDONLY|O_LARGEFILE)
04 sys_open spend 8746 ns
➥ to complete
05 sys call_open gedit -pid
➥ 3976 args ("/home/best/.gnome2/
➥ gedit-metadata.xml", O_RDONLY)
06 sys_open spend 8487 ns to
complete
07 sys call_open gedit -pid
➥ 3976 args ("/home/best/
➥ stevebest", O_RDONLY)
```

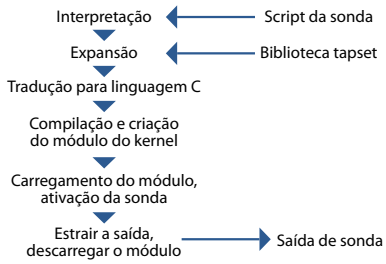

Figura 1: Como o SystemTap funciona.

No próximo passo, mostrarei um exemplo de prova que informa o tempo gasto em uma chamada de sistemas do tipo `open`. Vou precisar de uma sequência da qual tirar valores de tempo, então explicarei os recursos chave do SystemTap no que diz respeito à leitura de valores de uma sequência.

Ler valores de uma sequência

Ler valores de uma sequência é como ler valores de uma variável. Para tanto, inclua a sequência `array_name[index_expression]` como um elemento em uma expressão matemática. Por exemplo:

```
delta = gettimeofday_ns()
➥ - some[tid()]
```

No exemplo acima, a sequência `some[]` foi criada usando uma construção que vincula um valor associado. Você pode usar o sinal de igual para programar um valor associado para os pares únicos indexados, como segue:

```
array_name[index_expression]
➥ = value
```

A declaração da sequência mostra como programar um valor explicitamente associado a uma chave única. Você também pode usar uma função de manipulação como a *index_expression* e *value*. Por exemplo, pode-se usar

sequências para programar um carimbo de tempo da mesma maneira como se usa um valor associado a um nome de processo (que recomenda-se usar como sua chave única), como abaixo:

```
some[tid()] = gettimeofday_ns()
```

O código anterior programa um carimbo de tempo que servirá como um ponto de referência para ser usado para computar o delta. Toda vez que um evento chama uma declaração, o SystemTap retorna o valor `tid()` apropriado. Ao mesmo tempo, o SystemTap usa a função `gettimeofday_ns()` para programar o carimbo de tempo correspondente como um valor associado à chave única definida pela função `tid()`. Esse passo cria uma sequência composta de pares chave contendo linhas de identificadores e carimbos de tempo.

A primeira sonda é mostrada na **listagem 1**. Ela mede o tempo gasto na chamada de sistema open, localizada no código do kernel dentro do arquivo `fs/open.c`. A chamada `open` tem a seguinte sintaxe:

```
SYSCALL_DEFINE3(open, const
➥ char __user *, filename, int,
➥ flags, int, mode)
```

A variante `.function` coloca uma prova perto do início da função em questão para que os parâmetros estejam disponíveis como variáveis de contexto. A variante `.return` coloca uma prova no momento do retorno da função em questão, de forma que o valor esteja disponível como uma variável de contexto `$return`. Os parâmetros também estão disponíveis para a função. Esta pode mudar os valores dos parâmetros no meio do processo, de forma que você pode observar os parâmetros na prova `.function` em vez de fazê-lo na prova `.return`.

Iniciarei a sonda usando o comando `stap` e escreverei o valor de retorno em um arquivo chamado `output` usando a opção `-o`.

```
# stap -o output probe1.stp
```

A sonda e o retorno amostral gerado quando o Gedit abre o arquivo `steve-best`, capturado pelo `sys_open`, levou 9,487 nanossegundos para se completar, como exibido na **listagem 2**.

A **listagem 3** mostra como é fácil adicionar um contador para o número de chamadas `sys_open` e exibir a conta quando a prova for finalizada. Os resultados e o retorno amostral da prova seriam algo como:

```
starting sys_open probe
Press Ctrl-C to stop
ending sys_open probe
sys_open() called 732 times
```

Na **listagem 4**, uma sonda para a chamada de sistema mkdir (`sys_mkdir`) retorna o diretório que está sendo criado e o modo de criação, além de informar se a criação foi bem sucedida (valor de retorno). Nessa sonda, é necessário acessar a sequência de caracteres no ponto da sonda. A sonda usa `user_string` para acessar o nome de caminho e tem a sintaxe:

```
user_string:string (addr:long)
```

Essa função copia uma sequência de caracteres do espaço do usuário em um endereço determinado. O código do kernel para a chamada `sys_mkdir` está localizada no arquivo `fs/namei.c`. O seguinte código mostra a sintaxe dessa chamada:

```
asmlinkage long sys_mkdir(const
➥ char * pathname, int mode)
```

Listagem 3: Contar chamadas sys_open

```
01 global count_for_sys_open
02 probe kernel.function
➥ ("sys_open")
03 {
04   count_for_sys_open++
05 }
06
07 probe begin {
08   printf("starting
➥ sys_open probe \n")
09   printf("Press Ctrl-C
➥ to stop \n")
10 }
11 probe end
12 {
13   printf( "ending sys_open
➥ probe \n")
14   printf( "sys_open()
➥ called %d times \n",
➥ count_for_sys_open)
15 }
16
17 # stap probe2.stp
```

Listagem 4: Chamada de sistema mkdir

```
01 probe syscall.mkdir
02 {
03   printf( "Creating
➥ directory %s mode is %d ",
➥ user_string($pathname), $mode)
04 }
05
06 probe syscall.mkdir.return
07 {
08   if (!$return)
09       printf("Created with
➥ rc %d \n", $return)
10   else
11       printf("Failed with
➥ rc %d \n", $return)
12 }
13
14 # stap probe3.stp
```

Listagem 5: Chamada vfs_write

```
01 global write_totals;
02 probe vfs.write
03 {
04   write_totals"[execname(),
➥ pid()]++"
05 }
06
07 probe end
08 {
09   printf"("*** write
➥ totals ***\n")"
10   foreach ([name,pid]
➥ in write_totals-)
11     printf( "%s (%d):
➥ %d \n", name, pid,
➥ write_totals[name,pid])
12 }
```

Listagem 6: Amostra do resultado

```
01 * write totals *
02 metacity (3582): 327
03 gnome-screensav (3716): 260
04 wnck-applet (3680): 222
05 gnome-panel (3586): 169
06 bonobo-activati (3592): 152
07 gnome-power-man (3647): 151
08 gnome-settings- (3562): 137
09 gnome-terminal (5745): 111
10 notification-da (3720): 102
11 simpress.bin (4020): 85
```

Listagem 7: Chamadas sys_read e sys_write

```
01 global count_sys_read,
➥ count_sys_write, sys_read_
➥ bytes_req, sys_rd_bytes,
➥ sys_write_bytes_req,
➥ sys_wr_bytes
02
03 probe kernel.function
➥ ("sys_read") {
04   count_sys_read++
05   sys_read_bytes_req
➥ += $count
06 }
07
08 probe kernel.function
➥ ("sys_read").return {
09   rnb = $return
10
11   if (rnb == -1)
12     printf(" +++Read not
➥ successful+++ \n")
13   else
14     sys_rd_bytes += rnb
15 }
16
17 probe kernel.function
➥ ("sys_write") {
18   count_sys_write++
19   sys_write_bytes_req
➥ += $count
20 }
21
22 probe kernel.function
➥ ("sys_write").return {
23   wnb = $return
24
25   if (wnb == -1)
26     printf(" ***Write not
➥ successful*** \n")"
27   else
28     sys_wr_bytes += wnb
29 }
30
31 probe begin {
32   printf("Starting probe
➥ sys_read and sys_write \n")
33 }
34
35 probe end {
36   printf("Ending probe
➥ sys_read and sys_write \n")
37   printf( "%d bytes
➥ requested to read \n",
➥ sys_read_bytes_req)
38   printf("Total bytes read
➥ %d \n",sys_rd_bytes)
39   printf("%d calls to
➥ sys_read() \n",count_sys_read)
40   printf( "%d bytes
➥ requested to write \n",
➥ sys_write_bytes_req)
41   printf("Total bytes
➥ written %d \n",sys_wr_bytes)
42   printf("%d calls to
➥ sys_write() \n",count_sys_write)
43 }
44
45 # stap probe5.stp
```

Os resultados da sonda e o retorno padrão da sonda seriam algo como:

```
Creating directory steveb1 mode
➥ is 511 Created with rc 0
```

O próximo script (**listagem 5**) é um exemplo de como investigar um problema envolvendo um sistema que parece realizar um número excessivo de escritas. Ele usa o tapset `vfs` e uma sequência associativa para armazenar o número de registros que um determinado executável desempenha.

A biblioteca tapset está incluída no diretório `vfs.stp`, localizado em `/usr/share/systemtap/tapset`. Para conferir o código fornecido por essa tapset ou outras, veja os arquivos no diretório.

A fonte do kernel para o `vfs_write` está em `fs/read_write.c`. O `vfs_write` chama a sintaxe:

```
ssize_t vfs_write(struct file
➥ *file, const char __user
➥ *buf, size_t count, loff_t *pos)
```

e te dará a lista de executáveis e seus PIDs organizados pelo número de registros vfs realizados enquanto o script estava executando:

```
# stap probe4.stp
```

Os resultados da prova e suas amostras de saída podem ser semelhantes aos exibidos na **listagem 6**.

O próximo exemplo é um pouco mais abrangente, pois cria sondas para as chamadas `sys_read` e `sys_write`, proporcionando o número de bytes lidos e registrados (veja a **listagem 7**). O código do kernel para `sys_read` está em `fs/read-write.c`. A chamada `sys_read` tem a seguinte sintaxe:

```
asmlinkage ssize_t
➥ sys_read(unsigned int fd,
➥ char __user * buf, size_t count)
```

A fonte do kernel para `sys_write` também está em `fs/read_write.c`. A chamada `sys_write` tem a seguinte sintaxe:

```
asmlinkage ssize_t
➥ sys_write(unsigned int fd, const
➥ char __user * buf, size_t count)
```

Os resultados da sonda e de sua saída de amostra podem ser semelhantes aos exibidos na **listagem 8**.

Conclusão

Com o SystemTap você pode ter uma visão completa dos trabalhos internos do sistema operacional. Essa tecnologia te deixa capturar somente os dados que realmente resolvem o problema, diferentemente do rastreamento estático, que geralmente retorna muito mais informações do que o necessário. ■

Listagem 8: Resultados da sonda

```
01 Starting probe sys_read
➥ and sys_write
02 Ending probe sys_read and
➥  sys_write
03 50996284 bytes requested
➥ to read
04 Total bytes read 4193832
05 14751 calls to sys_read()
06 1532394 bytes requested
➥ to write
07 Total bytes to written
➥ 1532390
08 2232 calls to sys_write()
```

Mais informações

[1] SystemTap: `http://sourceware.org/systemtap/wiki`

[2] Interface gráfica para o SystemTap: `http://stapgui.sourceforge.net/`

[3] Referência da linguagem do SystemTap Language Reference: `http://sourceware.org/systemtap/langref/`

Gostou do artigo?

Queremos ouvir sua opinião. Fale conosco em cartas@linuxmagazine.com.br

Este artigo no nosso site: http://lnm.com.br/article/5602