Módulo 8

Sistema Operacional

Lição 6

Observabilidade

Versão 1.0 - Mar/2008

Autor
-

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alberto Ivo da Costa Vieira
Alexandre Mori
Alexis da Rocha Silva
Allan Wojcik da Silva
Antonio José Rodrigues Alves Ramos
Angelo de Oliveira
Bruno da Silva Bonfim
Carlos Fernandes Gonçalves
Denis Mitsuo Nakasaki
Felipe Gaúcho
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw
Seire Pareja
Thiago Magela Rodrigues Dias
Vinícius Gadis Ribeiro

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

Original desta por – McDougall e Mauro – Solaris Internals. Sun Microsystems. 2007.

1. Objetivos

A Sun fornece freqüentemente aos desenvolvedores acesso imediato às novas tecnologias, assim como ferramentas para rastreabilidade. Estas permitem que os usuários possam analisar profundamente os detalhes da implementação interna do software, do núcleo e do usuário.

Neste ponto já discutimos os conceitos teóricos do sistema operacional. Nesta lição veremos comandos que torna possível observar como estes conceitos são executados no sistema operacional. Discutiremos alguns comandos básicos do poderoso conjunto de ferramentas **DTrace**.

Ao final desta lição, o estudante será capaz de:

- Trabalhar com o conjunto de ferramentas **DTrace**
- Realizar comandos no Solaris que permitem ver em ação: CPU, processos, memória e E/S

2. DTrace

DTrace é uma poderoso conjunto de ferramentas que foi incluído no **Solaris 10** e permite visualizar o comportamento atual de um sistema durante sua execução. **Dtrace** foi desenvolvido como uma ferramenta para diagnóstico do sistema. Antes do **DTrace**, para tentar ver como determinado programa executava, era necessário inserir diversas instruções na saída de seu código. Essa prática é difícil de ser realizada quando necessitamos conhecer como um programa está em área de memória ou quão freqüentemente o arquivo executável chama uma determinada função do núcleo do sistema, ou mesmo como o programa acessa o disco rígido. Isso pode envolver realmente a edição do código fonte do próprio sistema operacional.

Para evitar esses problemas, **DTrace** usa **probes** (sondas) que estão conectados dentro do Sistema Operacional Solaris. Estes **probes** mantêm um registro de tudo, desde processos em uso de memória a quão frequentemente interrupções são chamadas pelo programa e a dados que são gravados no disco. Existem mais de 30.000 **probes** no Solaris. **Probes** não interferem na execução. Habilitar certos **probes** com **DTrace** nos permite um olhar dentro do núcleo do sistema operacional enquanto este está sendo executado.

2.1. Probes BEGIN e END

Iniciaremos o aprendizado com um comando muito simples. Vamos chamar o **DTrace** com a opção -n:

```
# dtrace -n BEGIN

dtrace: description 'BEGIN' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN
```

O **probe** BEGIN é iniciado quando um processo é iniciado. É necessário pressionar **CTRL+C** para sair do **DTrace**. Agora consideraremos outro **probe**, chamado END, que é ativado sempre que um processo finaliza. Quando é executado, o **probe** END não aparece imediatamente. Pois o processo ainda não acabou.

É necessário parar o **DTrace** para ver a ativação do **probe** END pressionando CTRL+C:

```
# dtrace -n BEGIN -n END

dtrace: description 'BEGIN' matched 1 probe
dtrace: description 'END' matched 1 probe
CPU     ID                    FUNCTION:NAME
  2      1                           :BEGIN
 ^C
  2      2                           :END
```

Até agora, vimos apenas **probes** que são ativados quando **DTrace** começa ou termina, sem nenhuma funcionalidade adicional. Escreveremos um *script* para que o **DTrace** possa executar quando o **probe** BEGIN ou END for iniciado.

2.2. Alô mundo

DTrace aceita um *script* escrito na linguagem **D**, similar à linguagem **C** ou **C++**. A sintaxe básica de um *script* em **D** se parece com:

```
<probe description>
/<predicate>/
{
  <código>
}
```

A sessão será executada quando o **probe** listado na descrição for ativado e o predicado (que é uma expressão condicional) for verdadeiro. Com essa idéia, considere o *script* a seguir para

DTrace, o qual será salvo com o nome de **alo.d**. Podemos ver que a instrução "Alô Mundo" será executada quando o **probe** BEGIN iniciar, e a instrução "Adeus" na ativação do **probe** END.

```
BEGIN
{
        trace("Alô Mundo!");
}

END
{
        trace("Adeus!");
}
```

Executaremos agora o **DTrace** utilizando a opção **-s** para informar o nome do arquivo *script* que iremos rastrear.

```
# dtrace -s alo.d
dtrace: script 'alo.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN   Alô Mundo!
    ^C
  2      2                             :END   Adeus!
```

Note como modificamos o comportamento do **DTrace** para mostrar seu início e fim. Rastrear um programa significava ter que modificá-lo, editando-o para inserir instruções de saída. Com **DTrace**, podemos rastrear um programa sem ter que modificar isso simplesmente utilizando o **probe** correto.

2.3. Organização dos probes

Podemos listar todos os **probes** através da opção **-l**. Existem mais de 30.000 **probes** disponíveis no Solaris.

```
# dtrace -l
  ID    PROVEDOR            MÓDULO                  NOME DA FUNÇÃO
   1      dtrace                                           BEGIN
   2      dtrace                                           END
   3      dtrace                                           ERROR
   4      vminfo          fasttrap         fasttrap_uwrite softlock
   5      vminfo          fasttrap          fasttrap_uread softlock
   6 nfsmapid229          nfsmapid            check_domain daemon-domain
```

Probes são identificados pelos **ID** e pelo seu **nome**, que é composto pelos seguintes valores separados por dois-pontos:

- Provedor
- Módulo
- Função
- Nome

Por exemplo, **probe** de ID 5, pode ser também referido como:

```
vminfo:fasttrap:fasttrap_uread:softlock
```

Provedores são módulos do núcleo que contêm o código dos **probes**. **Módulo** e **Função** indicam o módulo do núcleo, biblioteca do usuário ou nome de função que o **probe** é projetado para rastrear. A última parte e o **Nome** do **probe** é a descrição do que é designado a fazer.

Aqui estão alguns exemplos de **Provedores**:

- **DTrace** – **probes** relacionados ao **DTrace**
- **Lockstat** – **probes** para sincronização em nível de núcleo do sistema
- **Profile** – **probes** que executam todos os intervalos especificados que podem ser usados

para obter uma amostra do sistema em execução

- **Syscall** – **probes** para cada entrada e retorno de cada chamada de sistema no programa
- **VMinfo** – **probes** de memória virtual
- **Proc** – **probes** de processo, *threads* e criação e finalização de LWP
- **Sched** – **probes** de agendamento de processo

Para especificar um **probe** em um *script* D, simplesmente coloque seu nome completo na parte de descrição do **probe**. Caso não seja preeenchida partes da descrição, o código iniciará todos os **probes** que coincidam.

Por exemplo, o *script* a seguir roda o código sempre que uma página for carregada da memória virtual para a memória principal, bem como o código que irá executar em qualquer **probe** provido pelo **syscall**.

```
vminfo:genunix:pageio_setup:pgin
{
  trace("Pagina em que ocorreu");
}
syscall:::
{
  trace("Rodando um probe syscall");
}
```

2.4. Variáveis

Variáveis no **DTrace** não possuem tipos de dados, isso significa que este é determinado durante a primeira atribuição de valor. Por exemplo, i = 0 cria uma variável, do tipo inteiro, denominada **i** com o valor 0, enquanto msg = "Alô" cria uma variável denominada msg do tipo *String* com o valor "Alô".

O dtrace:::BEGIN é comumente usado para inicializar variáveis e são acessíveis somente enquanto o processo estiver rodando.

Para mostrar as variáveis em ação, o *script* **DTrace** a seguir, chamado **countdown.d**, mostra o funcionamento de variáveis, e o **probe** profile:::tick-1sec, que roda a cada segundo.

```
dtrace:::BEGIN
{
  ctr = 10;
}

profile:::tick-1sec
{
  trace(ctr);
  ctr--;
}

dtrace:::END
{
  trace("Obrigado por usar meu programa.");
}
```

A saída de countdown.d é a seguinte:

```
# dtrace -s countdown.d
dtrace: script 'countdown.d' find 3 probes
CPU     ID                    FUNCTION:NAME
  2  41214                       :tick-1sec        10
  2  41214                       :tick-1sec         9
  2  41214                       :tick-1sec         8
  2  41214                       :tick-1sec         7
  2  41214                       :tick-1sec         6
  2  41214                       :tick-1sec         5
  2  41214                       :tick-1sec         4
```

```
  2  41214                        :tick-1sec         3
  2  41214                        :tick-1sec         2
  2  41214                        :tick-1sec         1
  2  41214                        :tick-1sec         0
  2  41214                        :tick-1sec        -1
  2  41214                        :tick-1sec        -2
^C
  2      2                        :END   Obrigado por usar meu programa
```

2.5. Predicados

Predicados agem como uma instrução para um *script*. O código do *script* só é executado juntamente com o **probe** e o predicado é então comparado.

Realizamos a seguinte modificação no *script* **countdown.d**:

```
dtrace:::BEGIN
{
  ctr = 10;
}

profile:::tick-1sec
/ ctr > 0 /
{
  trace(ctr);
  ctr--;
}

profile:::tick-1sec
/ ctr == 0/
{
  trace(ctr);
  exit(0);
}

dtrace:::END
{
  trace("O tempo acabou!");
}
```

Nosso primeiro profile:::tick-1sec executará quando a condição "ctr maior que 0" for verdadeira, enquanto o segundo só irá rodar quando a condição "ctr igual a 0" for verdadeira. A função *exit*() finaliza o **trace**.

dtrace -s countdown.d

```
dtrace: script 'countdown.d' encontrou 4 probes
CPU     ID                    FUNCTION:NAME
  2  41214                        :tick-1sec        10
  2  41214                        :tick-1sec         9
  2  41214                        :tick-1sec         8
  2  41214                        :tick-1sec         7
```

2.6. printf

O comando **printf** mostra o valor de um determinado atributo na tela. Sua sintaxe é semelhante à da linguagem C/C++. Para mostrar uma lista de strings utilizando a função *printf*(), utilizamos uma vírgula para separá-las:

```
printf("O tempo restante é %d segs. O que posso dizer é %s", ctr, msg);
```

O formato *String* é apresentado na tela através da substituição dos atributos denotados pelo símbolo %, seguido de um caractere que identifica o tipo de atributo.

- %d significa que um valor inteiro será exibido naquela posição.

- %s significa que uma *String* será exibida naquela posição.
- %f significa que um número decimal será exibido naquela posição.
- %x exibe números inteiros como caracteres em hexadecimal.
- %% imprime um percentual na posição.

Modificamos o *script* **countdown.d** para exemplificar o uso da função *printf*():

```
dtrace:::BEGIN
{
  ctr = 10;
  msg = "Adeus"
}

profile:::tick-1sec
/ ctr > 0 /
{
  printf("Tempo restante de %d segs\n",ctr);
  ctr--;
}

profile:::tick-1sec
/ ctr == 0/
{
  printf("Tempo restante de %d segs\n", ctr);
  exit(0);
}

dtrace:::END
{
  printf("Tempo acabando! %d segs. Tudo que posso dizer é %s\n", ctr, msg);
}
```

Para melhorar a visualização, podemos utilizar a opção -q para listar as strings.

```
# dtrace -q -s countdown.d
Tempo restante 10 segs
Tempo restante  9 segs
Tempo restante  8 segs
Tempo restante  7 segs
Tempo restante  6 segs
Tempo restante  5 segs
Tempo restante  4 segs
Tempo restante  3 segs
Tempo restante  2 segs
Tempo restante  1 segs
Tempo restante  0 segs
Tempo acabado! 0 segs. Tudo que posso dizer é Adeus
```

2.7. Scripts no DTrace – parte 1

Considere o seguinte *script* salvo com o nome **syscall.d**. Syscall:::entry é o conjunto de **probes** disparados sempre que uma aplicação chama uma função do núcleo do sistema. **Execname**, **PID** e **probefunc** são atributos no **DTrace**. **Execname** é o nome do processo que originou este **probe**. **PID** é a identificação do processo. **Probefunc** é o nome da função do **probe**, indicando qual função do sistema foi chamada.

Adicionaremos um predicado para considerar apenas os processos em **bash**.

```
syscall:::entry
/ execname == “bash” /
{
  printf("%s(%d) chamado %s\n", execname, pid, probefunc);
}
```

Executando o *script*, é possível que não ocorre nenhuma saída inicial. Isto porque o *script* **bash**

atual que está em execução é o **DTrace**. Para se obter uma saída inicie outra janela do **terminal** e execute alguns comandos em **bash**. Como por exemplo:

```
# dtrace -q -s syscall.d
Bash(1035) chamado read
Bash(1035) chamado read
Bash(1035) chamado write
Bash(1035) chamado read
Bash(1091) chamado getpid
Bash(1091) chamado lwp_self
Bash(1091) chamado lwp_sigmask
Bash(1091) chamado getpid
Bash(1091) chamado schedctl
...
```

Para investigar essas chamadas de sistema ao núcleo e descobrir como o sistema funciona, uma boa dica é verificar o código-fonte dessas funções.

2.8. Funções agregadas

Talvez seja necessário saber quais funções foram chamadas, mas como saber quantas vezes cada função foi chamada? Podemos utilizar funções agregadas para analisar os dados.

Para ilustrar um exemplo, modificamos o *script* **syscall.d** para produzir um total de **probefunc** que foram chamadas pelo **bash**.

```
syscal:::entry
/execname == "bash"/
{
  printf("%s(%d) chamado %s\n", execname, pid, probefunc);
  @[probefunc] = count();
}
```

A execução prossegue normal, mas no final do *script* obtemos a seguinte saída, que mostra quantas vezes cada função do sistema foi chamada.

```
^C
  exece                                                             1
  fork1                                                             1
  lwp_self                                                          1
  schedctl                                                          1
  setcontext                                                        1
  stat64                                                            1
  waitsys                                                           1
  getpid                                                            2
  gtime                                                             3
  read                                                              3
  write                                                             4
  setpgrp                                                           6
  ioctl                                                            15
  lwp_sigmask                                                      22
  sigaction                                                        33
```

A sintaxe básica da função agregada é:

```
@name[key] = aggfunc(args)
```

A parte esquerda da expressão cria um array. Um vetor agregado é um conjunto que, em vez de índices, utiliza strings para referenciar seus elementos. Por exemplo, @vendas["seg"]=14; if (@vendas["ter"]> 10), etc.

O **@** indica que estamos definindo uma função agregada. O **name** é um nome qualquer para o agregado e **key** é um atributo cujos valores tornam-se índices do array.

O lado direito da expressão que define uma função agregada, pode ser:

- count() - quantas vezes ocorreu um evento.

- sum(exp) – executa a soma da expressão.
- avg(exp) – executa a média aritmética da expressão.
- min(exp), max(exp) – mínimo e máximo, respectivamente, de uma expressão.
- quantize(exp) – cria um gráfico de uma expressão.

Não é necessário exibir explicitamente uma função agregada, pois o **DTrace** automaticamente assume que, todas as funções agregadas são impressas no final da execução do *script*.

2.9. Scripts no DTrace – parte 2

Em vez de apenas contar quantas vezes uma chamada ao sistema foi executada, também podemos conhecer quanto tempo levou para executar. Para tornar isso simples, consideraremos apenas a função *read*. Nosso próximo *script* **timestamp.d** utiliza dois **probes**, syscall::read:entry e syscall::ready:exit. Também utilizamos o atributo **timestamp** que retorna a hora atual.

Para achar a duração, gravamos a *timestemp* na syscall:::entry e a subtraímos da nova timestamp localizada na syscall:::exit.

```
syscall::read:entry
{
  t = timestamp;
}

syscall::read:return
{
  delay = timestamp - t;
  printf("%s(%d) tempo em %s: %d nsecs\n", execname, pid, probefunc, delay);
  t = 0;
}
```

O problema com este código é que vários processos o chamam e modificam o valor do atributo t, que corresponde ao tempo em que o sistema começou a funcionar.

Para resolver este problema, **DTrace** verifica sua estrutura de atributos. Qualquer atributo inserido no **DTrace** é exclusivo de uma *thread*. Por exemplo, ao declarar um atributo self->t, então este é único para cada *thread*.

A seguir, o *script* **timestamp.d** modificado:

```
syscall::read:entry
{
  self->t = timestamp;
}

syscall::read:return
{
  self->delay = timestamp - self->t;
  printf("%s(%d) tempo no método %s: %d nsecs\n", execname, pid, probefunc,
self->delay);
  self->t = 0;
}
```

O código de saída poderia ser mostrado da maneira:

```
# dtrace -q -s timestamp.d
Xsun(485)   tempo em read: 33261 nsecs
Xsun(485)   tempo em read: 23697 nsecs
Xsun(485)   tempo em read: 25137 nsecs
sshd(1405)         tempo em read: 46509 nsecs
Xsun(485)   tempo em read: 27892 nsecs
Xsun(485)   tempo em read: 21706 nsecs
sshd(1405)  tempo em read: 26893 nsecs
sshd(1405)  tempo em read: 13417 nsecs
sshd(1405)  tempo em read: 21497 nsecs
...
```

Para tornar os dados mais significativos, podemos utilizar a função agregada *quantize*.

```
syscall::read:entry
{
  self->t = timestamp;
}

syscall::read:return
{
  self->delay = timestamp - t;
  @[execname] = quantize[self->delay];
  self->t = 0;
}
```

A saída do código é:

```
# dtrace -q -s timestamp.d
^C
  nfsmapid
           valor  ------------- Distribuição ------------- count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@                   1
           16384 |                                         0
           32768 |@@@@@@@@@@@@@@@@@@@@@@                   1
           65536 |                                         0

  sshd
           valor  ------------- Distribuição ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@@@@@                   1
           32768 |@@@@@@@@@@@@@@@@@@@@@@                   1
           65536 |                                         0

  Xsun
           valor  ------------- Distribuição ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  41
           32768 |@@@@                                     4
           65536 |                                         0
```

2.10. DTrace toolkit

Não iremos só discutir uma visão geral do **DTrace** através da linguagem **D**. Para saber mais sobre o **DTrace**, acesse o manual em:

```
http://docs.sun.com/app/docs/doc/817-6223
```

Para tornar as coisas simples, um *toolkit* do **DTrace** pode ser baixado e instalado. O *toolkit* **DTrace** contém uma grande quantidade de *scripts* prontos, que monitoram diversos **probes**.

O *toolkit* **DTrace** pode ser baixado no endereço:

```
http://www.opensolaris.org/os/community/dtrace/dtracetoolkit/
```

Vejamos como instalar e executar a versão **DTraceToolkit-0.99.tar.gz**. Uma vez realizado o download do arquivo, execute os seguintes comandos:

```
gunzip DtraceToolkit-0.99.tar.gz
tar xvf DTraceToolkit-0.99.tar
cd DTraceToolkit-0.99
./install
```

DTraceToolkit será instalado por padrão no diretório **/opt/DDT**. Analisaremos a seguir o conteúdo deste diretório:

- apps/ - *scripts* específicos de aplicação

- cpu / - *scripts* para análise de CPU
- disk/ - *scripts* para análise de I/O
- docs/ - documentação
- mem/ - scripts para análise de memória
- proc/ - scripts para análise de processos

Casa *script* possui uma página principal associada, embora não seja automaticamente instalado. Para acessar a página principal dos *scripts*, executamos o comando **man** com as seguintes opções:

```
# man -M <DTT Man directory> <command>
```

Por padrão, o diretório é **/opt/DTT/Man**. Por exemplo, para solicitar auxílio sobre o *script* **runocc.d**, que verifica o pool de ocupação da CPU, executamos o seguinte comando:

```
# man -M /opt/DTT/Man runocc.d
```

Atualmente, está em desenvolvimento o monitoramento para programas Java, bem como monitoramento para *JavaScript* e aplicações WEB. Quando estiverem prontos, poderão ser usado para rastrear programas Java sem ter de modificar o código fonte.

3. Informações da CPU

3.1. Comando vmstat

O comando **vmstat** lista informações sobre o comportamento global da CPU desde a sua inicialização. Executando este comando, podemos produzir a seguinte saída:

```
# vmstat
kthr      memory            page            disk          faults      cpu
r b w   swap  free  re  mf pi po fr de sr f0 s0 s2 s6   in   sy   cs us sy id
0 0 0 2678760 1842984 0  1  0  0  0  0  0  0  0  0  0  508   30   46  0  1 99
```

Veremos com mais detalhes estes campos:

- **kthr** – número de *threads* nos seguintes estados:
  - **r** – número de *threads* no núcleo executando *queue*
  - **b** – número de *threads* bloqueadas esperando, por exemplo, I/O, recursos do sistema, entre outros
  - **w** – número de pares de troca **LWP** que estão esperando para finalizar o processo de um recurso
- **memory** – memória utilizada em *kilobytes*
  - **swap** – espaço em área de troca
  - **free** – memória livre
- **page** – informação sobre como a memória está sendo utilizada
- **disk** – informação sobre as operações em disco por segundo
- **faults** – informação sobre os *traps* do sistema
- **cpu** – percentual de uso da CPU
  - **us** – utilizada pelo usuário
  - **sy** – tempo do sistema
  - **id** – tempo livre

Também é possível executar o **vmstat** com a informação do intervalo de saída, que executa o comando após cada intervalo específico, para obter uma visão global dos processos que estão em execução.

```
# vmstat 5
kthr      memory            page            disk          faults      cpu
r b w   swap  free  re  mf pi po fr de sr f0 s0 s2 s6   in   sy   cs us sy id
0 0 0 2678752 1842968 0  1  0  0  0  0  0  0  0  0  0  508   30   46  0  1 99
0 0 0 2657392 1820800 0  5  0  0  0  0  0  0  0  0  0  503   50   49  0  1 99
0 0 0 2657392 1820800 0  0  0  0  0  0  0  0  0  0  0  502   34   44  0  1 99
0 0 0 2657392 1820800 0  0  0  0  0  0  0  0  0  0  0  504   44   50  0  1 99
0 0 0 2657392 1820800 0  0  0  0  0  0  0  0  0  0  0  508   70   67  0  1 99
0 0 0 2657384 1819848 2 24 234 0  0  0  0  0  0 33  0  609  663  153  0  2 98
0 0 0 2656272 1815528 3 27 2557 0 0  0  0  0  0 328 0 1471 4262  722  1  6 93
0 0 0 2649720 1791392 0  0 3103 0 0  0  0  0  0 438 0 1896 4418  890  1  7 92
0 0 0 2644960 1771032 0  0 3289 0 0  0  0  0  0 459 0 1880 4639  960  1  7 92
0 0 0 2639400 1750232 3  0 2773 0 0  0  0  0  0 395 0 1671 6227  844  1 10 89
```

Podemos computar o uso da CPU utilizada, subtraindo o tempo livre (**cpu id**) de 100.

3.2. uptime

Para descobrir há quanto tempo seu computador está ativo, bem como a média de carga na **CPU**, executamos simplesmente o comando **uptime**:

```
# uptime
10:50am  up 3 day(s), 5 min(s),  2 users,  load average: 0.11, 0.04, 0.02
```

Note a coluna que contém a média de carga (*load average*): está em 1, 5 e 15 minutos de média para carregar na **CPU**. Estes números são um reflexo de como muitos processos são executados no computador. Por exemplo, 1.00 é 100% de utilização de CPU em um único processador, mas seria a metade em um computador com dois processadores. Se for obtido mais do que o processador consegue executar, significa que está ocorrendo uma saturação de CPU.

3.3. Scripts no DTrace – Parte 3

No diretório **/opt/DTT/cpu**, podemos executar um *script* em **DTrace** para recuperar as informações sobre a CPU dos computadores:

- cputypes.d – lista a informação sobre cada CPU
- loads.d – mostra a média de carga
- intbycpu.d – mostra o número de interrupções manipuladas por cada CPU
- runocc.d – mostra as execuções que estão em uma *queue*

3.4. Script shellsnoop

A aplicação **shellsnoop** usa o **DTrace** para mostrar o que está sendo exibido em outros terminais. A saída a seguir mostra um usuário alterando sua senha:

```
# ./shellsnoop
  PID  PPID      CMD DIR  TEXT
 1412  1411     bash   W  #
 1412  1411     bash   R  p
 1412  1411     bash   W  p
 1412  1411     bash   R  a
 1412  1411     bash   W  a
 1412  1411     bash   R  s
 1412  1411     bash   W  s
 1412  1411     bash   R  s
 1412  1411     bash   W  s
 1412  1411     bash   R  w
 1412  1411     bash   W  w
 1412  1411     bash   R  d
 1412  1411     bash   W  d
 1412  1411     bash   R
 1412  1411     bash   W

 1734  1412   passwd   W  passwd
 1734  1412   passwd   W  : Changing password for
 1734  1412   passwd   W  mario
 1734  1412   passwd   W
 1734  1412   passwd   W  New Password:
 1734  1412   passwd   W
 1734  1412   passwd   W  Re-enter new Password:
 1734  1412   passwd   W
 1734  1412   passwd   W  passwd: password successfully changed for mario
 1734  1412   passwd   W

 1412  1411     bash   W  #
```

A senha não é exibida, **shellsnoop** mostra unicamente o que aparece na tela do terminal.

4. Processos

4.1. Comando ps

O comando **ps** é um comando padrão para listar informações dos processos.

```
# ps -ef
   UID  PID PPID   C    STIME TTY   TIME CMD
  root    0    0   0   Dec 10 ?     0:12 sched
  root    1    0   0   Dec 10 ?     0:01 /sbin/init
  root    2    0   0   Dec 10 ?     0:00 pageout
  root    3    0   1   Dec 10 ?     6:29 fsflush
  root 1412 1411   0 08:14:12 pts/3 0:01 bash
  root    7    1   0   Dec 10 ?     0:09 /lib/svc/bin/svc.startd
  root    9    1   0   Dec 10 ?     0:24 /lib/svc/bin/svc.configd
  root   95    1   0   Dec 10 ?     0:00 /usr/lib/snmp/snmpdx -y -c /etc/snmp
daemon  220    1   0   Dec 10 ?     0:00 /usr/sbin/rpcbind
  root  214    1   0   Dec 10 ?     0:00 /usr/sbin/cron
  ...
```

A opção **-e** mostra todos os processos, e a opção **-f** lista os processos que contém as colunas completas.

- **UID** – identificação do usuário do processo
- **PID** – identificação do processo
- **PPD** – identificação do processo pai
- **C** – coluna obsoleta. Corresponde à utilização do processo agendado
- **STIME** – tempo de inicialização de um processo
- **TTY** – terminal de controle (? se não for controlado por um terminal)
- **TIME** – tempo que um processo está executando na CPU
- **CMD** – comando usado para inicializar o processo

4.2. Scripts no DTrace – Parte 4

Os *scripts* de processo são encontrados no diretório **/opt/DTT/proc**. Vejamos a seguir alguns dos mais usados para monitorar os processos:

- **sampleproc** – um arquivo executável que usa **DTrace** para a inspeção em muitas CPUs na qual a aplicação é executada
- **writebytes.d** e **readbytes.d** – como os *bytes* são lidos e escritos pelo processo
- **syscallbyproc.d** e **syscallbypid.d** – sistema de chamadas por processo ou por identificação do processo
- **filebyproc.d** – lista de arquivos abertos por um processo
- **crash.d** – relatório sobre aplicações que falharam

5. Memória

5.1. pmap -x

O comando **pmap** associado à opção **-x** mostra uma visão da memória com a identificação do processo. O exemplo a seguir mostra um mapa de um processo na memória com ID **1412**. Para explorar o endereço de memória de um processo em particular, temos que usar o comando **ps** para procurar o próprio **ID** do processo.

```
# pmap -x 1412
1412:   bash
 Address  Kbytes     RSS    Anon  Locked Mode   Mapped File
00010000     648     624       -       - r-x--  bash
000C0000      80      48      16       - rwx--  bash
000D4000     168     168      64       - rwx--    [ heap ]
FF100000     864     856       -       - r-x--  libc.so.1
...
```

5.2. Scripts no DTrace – Parte 5

Os *scripts* de memória são encontrados no diretório **/opt/DTT/mem**. A seguir são mostrados alguns *scripts* mais comumente utilizados para analisar a memória:

- **vmstat.d** – utilizado para escrever em **D**
- **xvmstat** – um arquivo executável (**./xvmstat**) que usa **DTrace** para mostrar mais informações em relação ao **vmstat**, tais como memória RAM livre, memória virtual livre entre outros
- **swapinfo.d** – mostra informações da memória virtual
- **minfbypid.d** – detecta um grande consumidor de memória

6. Disco Rígido

6.1. Scripts no DTrace – Parte 6

Scripts para controle do disco rígido são localizados em **/opt/DTT/disk**. A seguir, são mostrados alguns *scripts* mais utilizados para analisar o disco rígido:

- **iofile.d** – mostra o tempo de espera para entrada e saída
- **diskhits** – uma *queue* sendo executada, através de um filename, verifica a entrada e saída e a média de carga de um arquivo
- **iotop** – um arquivo executável que lista os eventos de entrada e saída do disco por processo
- **iosnoop** – um arquivo executável que monitora eventos de entrada e saída para um determinado *userid*, *processid* ou *filename*

A seguir um exemplo de saída de um **iosnoop** mostrando os arquivos editados por um determinado comando:

```
# ./iosnoop
  UID   PID D    BLOCK   SIZE       COMM PATHNAME
    0     3 W    28720   2560    fsflush <none>
    0  1726 R    71712   8192         vi /export/home/alice/temp.txt
    0  1726 R 12227792   8192         vi /export/home/alice/temp.txt
    0  1726 R 11606832   8192         vi <none>
    0  1726 R 12227936   8192         vi /export/home/alice/temp.txt
    0  1726 W  1081232   8192         vi /var/tmp/ExtBaWxd
    0  1726 W  1372256  90112         vi /var/tmp/ExtBaWxd
    0  1726 W 11627936 958464         vi /var/tmp/ExtBaWxd
    0  1726 R 12228880   8192         vi /export/home/alice/temp.txt
    0  1726 R 12229184   8192         vi /export/home/alice/temp.txt
```

6.2. Chime

Chime é uma interface gráfica para visualizar **DTrace**.

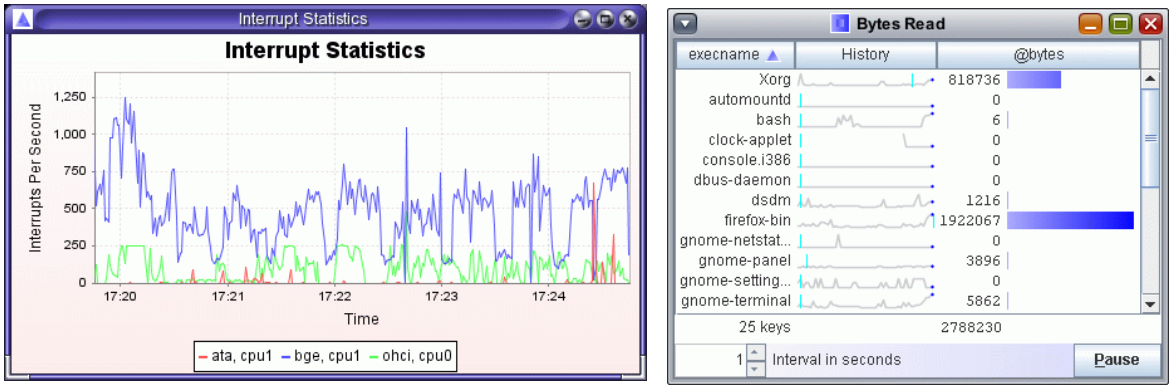

Figura 1: Janelas do Chime

O pacote, bem as instruções para instalação podem ser encontrados no site da comunidade **Open Solaris**. No endereço: http://www.opensolaris.org/os/project/dtrace-chime/.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Instituto Gaudium
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.