Desenvolvimento de programas DTrace

Diogo Sobral
a82523@alunos.uminho.pt

Maria Dias
a81611@alunos.uminho.pt

May 12, 2020

Abstract

Familiarização com a linguagem DTrace e alguns dos seus providers através da escrita de programas usando a respetiva linguagem. O projeto desdobra-se em quatro programas, que são desenvolvidos e posteriormente testados conforme as suas finalidades.

1 Introdução

Criado pela Sun Microsystems, o DTrace é uma framework de tracing dinâmica, que fornece serviços para avaliar o comportamento do sistema operativo e de programas *user-space*. Permite avaliar, medir a performance e resolver problemas relacionados com o kernel e aplicações no geral. A grande vantagem do DTrace *versus* outras ferramentas semelhantes é exatamente a possibilidade de, para além do habitual *sampling* que estas oferecem, podermos fazer uma instrumentação via traçado dinâmico, conseguindo agregar um grande conjunto de informação acerca dos programas, e ao mesmo tempo introduzindo um overhead mínimo nas medições que são feitas. Executando o comando **dtrace -l | wc -l** no sistema em estudo, cujo ambiente é Solaris 11, descobrimos que temos à nossa disposição 101217 probes distintos. De forma a melhor entender as probes que temos disponíveis para uso e associá-las corretamente às informações que pretendemos recolher, recolhemos uma lista de alguns dos *providers* disponibilizados e um sumário da sua função.

- **cpc:** contadores de performance de CPU;
- **dtrace:** probes relacionadas com o próprio DTrace;
- **fbt:** permite traçado dinâmico ao nivel do kernel;
- **fpuinfo:** permite investigar a simulação de instruções de vírgula flutuante em microprocessadores SPARC;
- **io:** probes relacionadas com o uso do disco;
- **lockstat:** probes que permitem avaliar o comportamento de locks;
- **pid:** permite o traçado dinâmico ao nível do utilizador;
- **plockstat:** reporta estatísitcas de locks ao nível do utilizador;
- **proc:** estudo de eventos ao nível do processo;
- **profile:** probes associadas com interrupções temporais;
- **sched:** estudo do escalonamento do CPU;
- **sdt:** permite criar probes em locais designados pelo programador;
- **syscall:** comportamento das chamadas de sistema;
- **sysinfo:** estatísticas do sistema;
- **vminfo:** estatísticas de memória virtual do kernel;
- entre outros.

Tendo feito um estudo teórico da ferramenta e das facilidades que oferece, passamos então a solucionar alguns exercícios práticos de ambientação com a mesma.

2 Exercício I

O primeiro exercício pede que se faça um traçado à *system call* **open()**, imprimindo as seguintes informações por linha:

- **nome** do executável
- **pid** do processo
- **uid** do utilizador
- **gid** do grupo
- **caminho absoluto** para o ficheiro a abrir
- **flags** usadas para abrir o ficheiro
- **valor de retorno** da chamada de sistema

2.1 Desenvolvimento

Para satisfazer este problema, interessa-nos definir o comportamento para uma *probe* que é ativada sempre que se faz uma chamada de sistema open. O provider que nos interessa para criar esta sonda é o *syscall*, e o nome da função que pretendemos estudar é *openat*, que, no ambiente usado, Solaris 11, tomou o lugar da chamada original open. Como queremos apanhar todas as variantes da função, usamos um asterisco para especificar isso mesmo. A probe que se segue faz o traçado das funções *openat* e *openat64*.

```
syscall::openat*:entry
```

A função em causa, como podemos ver nas páginas man usando o comando *man openat*, recebe vários argumentos, entre os quais o caminho absoluto para o ficheiro que se pretende abrir, que corresponde ao argumento de índice 1 (considerando que começam no índice 0), e as flags usadas para esse efeito, que são o argumento de ordem 2 da função. Usando a sonda apresentada anteriormente, conseguimos ir buscar estes valores com as variáveis *built-in* **arg1** e **arg2** e guardá-los localmente em cada thread que faz disparar a probe.

```
syscall::openat*:entry
{
   self->path = arg1;
   self->flag = arg2;
}
```

Para conseguir imprimir o valor de retorno da chamada, uma vez que este só fica disponível depois de a execução da função terminar, recorremos agora à sonda que é ativada sempre que a *system call* termina.

```
syscall::openat*:return
```

Com a ativação desta sonda passamos a ter acesso a todas as informações pretendidas, e podemos então imprimi-las, linha a linha, por cada processo que a despoleta. Para o nome do executável, pid do processo, uid do utilizador e gid do grupo, basta usar as variáveis built-in do DTrace **execname**, **pid**, **uid** e **gid**, respetivamente.

O valor de retorno da função pode ser retirado diretamente da variável **arg1**. Relativamente ao caminho absoluto do ficheiro que se tentou abrir, relembramos que este valor tinha sido guardado localmente em cada thread na variável **path**. Sendo que se trata de uma string, é necessário usar a função *copyinstr* antes de imprimir o valor, caso contrário produz erro. Outro detalhe importante é que esta função só deve ser aplicada após a chamada de sistema retornar para evitar erros. Isto porque a variável *path*, apesar de poder conter um valor válido, por ainda não ter sido acedida pelo processo em estudo, pode despoletar um erro de "endereço inválido".

Para imprimir a sequência de caracteres que corresponde ao modo de abertura do ficheiro, usamos o operador AND para comparar o valor da flag com as diferentes macros para as permissões de ficheiros. Se o ficheiro não tiver sido acedido com a flag O_RDWR ou O_WRONLY, assumimos que foi apenas aberto em modo de leitura, O_RDONLY.

```
syscall::openat*:return
{
   printf("%-6d %-16s %-6d %-6d %-64s %-8s", pid, execname, uid, gid,
       copyinstr(self->path), (self->path&O_WRONLY) ? "O_WRONLY" :
       (self->path&O_RDWR) ? "O_RDWR" : "O_RDONLY");
   printf("%-8s", self->flag&O_CREAT ? " | O_CREAT" : "");
   printf("%-8s", self->flag&O_APPEND ? " | O_APPEND" : "");
   printf("%d\n",arg1);

}
```

No final, obtemos o programa de DTrace **open.d**, demonstrado em baixo. Note-se que o *pragma* usado permite especificar opções para o comando dtrace; neste caso usamos a opção *quiet* para, ao correr o programa, apenas ser mostrado output relativo aos dados traçados. A primeira linha do script serve para o tornar num executável. O ficheiro deve ter permissões para ser executado e a partir daí podemos corrê-lo com o comando **./open.d**

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

/* Exercicio 1 - Monitorizar chamadas ao sistema open() */

BEGIN
{
   printf("%-6s %-16s %-6s %-6s %-64s %-20s
       %s\n","PID","PROCESS","UID","GID","FILENAME","MODE","RETURN");
}

syscall::openat*:entry
{
   self->path = arg1;
   self->flag = arg2;
}

syscall::openat*:return
{
   printf("%-6d %-16s %-6d %-6d %-64s %-8s", pid, execname, uid, gid,
       copyinstr(self->path), (self->flag&O_WRONLY) ? "O_WRONLY" :
       (self->flag&O_RDWR) ? "O_RDWR" : "O_RDONLY");
   printf("%-8s", self->flag&O_CREAT ? " | O_CREAT" : "");
   printf("%-8s", self->flag&O_APPEND ? " | O_APPEND" : "");
   printf("%d\n",arg1);

}
```

2.2 Output

O programa foi testado para os seguintes inputs:

- cat /etc/inittab > /tmp/test
- cat /etc/inittab >> /tmp/test
- cat /etc/inittab | tee /tmp/test
- cat /etc/inittab | tee -a /tmp/test

produzindo os resultados que se seguem.

2.2.1 cat /etc/inittab > /tmp/test1

```
PID   PROCESS        UID   GID   FILENAME                          MODE                 RETURN
12023 bash           1017  5000  /tmp/test1                        O_RDONLY | O_CREAT   4
12023 cat            1017  5000  /var/ld/64/ld.config              O_RDWR               -1
12023 cat            1017  5000  /lib/64/libc.so.1                 O_RDONLY             3
12023 cat            1017  5000  /usr/lib/locale/UTF-8/UTF-8       O_RDONLY             -1
12023 cat            1017  5000  /etc/inittab                      O_RDONLY             3
11944 sshd           0     0     /dev/dtrace/helper                O_RDONLY             3
1     init           0     0     /etc/inittab                      O_RDONLY             9
1     init           0     0     /system/volatile/init-next.state  O_RDONLY | O_CREAT   9
1     init           0     0     /system/volatile/init-next.state  O_RDONLY | O_CREAT   9
1     init           0     0     /etc/inittab                      O_RDONLY             9
^C
```

2.2.2 /etc/inittab >> /tmp/test10

```
PID   PROCESS        UID   GID   FILENAME                          MODE                          RETURN
13379 bash           1017  5000  /tmp/test10                       O_RDONLY | O_CREAT | O_APPEND 4
13379 cat            1017  5000  /var/ld/64/ld.config              O_RDWR                        -1
13379 cat            1017  5000  /lib/64/libc.so.1                 O_RDONLY                      3
13379 cat            1017  5000  /usr/lib/locale/UTF-8/UTF-8       O_RDONLY                      -1
13379 cat            1017  5000  /etc/inittab                      O_RDONLY                      3
^C
```

2.2.3 cat /etc/inittab | tee /tmp/test0

```
PID   PROCESS        UID   GID   FILENAME                          MODE                 RETURN
12773 tee            1017  5000  /var/ld/64/ld.config              O_RDWR               -1
12773 tee            1017  5000  /lib/64/libc.so.1                 O_RDONLY             3
12773 tee            1017  5000  /usr/lib/locale/UTF-8/UTF-8       O_RDONLY             -1
12773 tee            1017  5000  /tmp/test0                        O_RDONLY | O_CREAT   3
12772 cat            1017  5000  /var/ld/64/ld.config              O_RDWR               -1
12772 cat            1017  5000  /lib/64/libc.so.1                 O_RDONLY             3
12772 cat            1017  5000  /usr/lib/locale/UTF-8/UTF-8       O_RDONLY             -1
12772 cat            1017  5000  /etc/inittab                      O_RDONLY             3
^C
```

2.2.4 cat /etc/inittab | tee -a /tmp/test2

```
PID   PROCESS      UID   GID   FILENAME                          MODE                          RETURN
12838 tee          1017  5000  /var/ld/64/ld.config              O_RDWR                        -1
12838 tee          1017  5000  /lib/64/libc.so.1                 O_RDONLY                      3
12838 tee          1017  5000  /usr/lib/locale/UTF-8/UTF-8       O_RDONLY                      -1
12838 tee          1017  5000  /tmp/test2                        O_WRONLY | O_CREAT | O_APPEND 3
12837 cat          1017  5000  /var/ld/64/ld.config              O_RDWR                        -1
12837 cat          1017  5000  /lib/64/libc.so.1                 O_RDONLY                      3
12837 cat          1017  5000  /usr/lib/locale/UTF-8/UTF-8       O_RDONLY                      -1
12837 cat          1017  5000  /etc/inittab                      O_RDONLY                      3
^C
```

2.3 Exercício Opcional

Como exercício extra, foi sugerido adicionar uma condicionante que fizesse com que apenas ficheiros com "/etc" no caminho fossem detetados. Para isto, bastou acrescentar um predicado à sonda que dispara quando a chamada de sistema retorna, que faz a comparação do caminho absoluto dado para abrir o ficheiro com a string "/etc", por meio da função **strstr**. Esta função retorna 0 se não encontrar a parte comum referida nas strings, portanto apenas queremos que a probe dispare e execute quando o valor de retorno é diferente de zero.

```
/strstr(copyinstr(self->path),"/etc") != 0/
```

Após adicionar este predicado no devido lugar, ficamos com o programa que se segue.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

/* Exercicio opcional - detetar apenas ficheiros com "/etc" no caminho */
```

```
BEGIN
{
   printf("%-6s %-16s %-6s %-6s %-64s %-20s %s\n",
       "PID","PROCESS","UID","GID","FILENAME","MODE","RETURN");
}

syscall::openat*:entry
{
   self->path = arg1;
   self->flag = arg2;
}

syscall::openat*:return
/strstr(copyinstr(self->path),"/etc") != 0/
{
   printf("%-6d %-16s %-6d %-6d %-64s %-8s", pid, execname, uid, gid,
       copyinstr(self->path), (self->flag&O_WRONLY) ? "O_WRONLY" :
       (self->flag&O_RDWR) ? "O_RDWR" : "O_RDONLY");
   printf("%-8s", self->flag&O_CREAT ? " | O_CREAT" : "");
   printf("%-8s", self->flag&O_APPEND ? " | O_APPEND" : "");
   printf("%d\n",arg1);

}
```

2.3.1 Output

O output que se segue mostra o resultado da nossa alteração do programa inicial, sendo que todos os ficheiros detetados contêm agora obrigatoriamente o excerto "/etc" no seu caminho.

```
PID   PROCESS      UID   GID   FILENAME                                  MODE          RETURN
3257  sshd         0     0     /etc/ssh/ssh_host_rsa_key                 O_RDONLY      6
3257  sshd         0     0     /etc/ssh/ssh_host_rsa_key                 O_RDONLY      6
3257  sshd         0     0     /etc/ssh/ssh_host_rsa_key.pub             O_RDONLY      6
3257  sshd         0     0     /etc/ssh/ssh_host_ed25519_key             O_RDONLY      6
3257  sshd         0     0     /etc/ssh/ssh_host_ed25519_key             O_RDONLY      6
3257  sshd         0     0     /etc/ssh/ssh_host_ed25519_key             O_RDONLY      6
3257  sshd         0     0     /etc/ssh/ssh_host_ed25519_key.pub         O_RDONLY      6
3257  sshd         0     0     /etc/system.d/crypto:fips-140             O_RDONLY      -1
3257  sshd         0     0     /etc/ssh/ssh_host_rsa_key                 O_RDONLY      6
3257  sshd         0     0     /etc/gss/mech                             O_RDONLY      4
3257  sshd         0     0     /etc/gss/mech.d/                          O_RDONLY      4
3257  sshd         0     0     /etc/krb5/krb5.conf                       O_RDONLY      4
3257  sshd         0     0     /etc/hesiod.conf                          O_RDONLY      -1
3257  sshd         0     0     /etc/inet/hosts                           O_RDONLY      4
3257  sshd         0     0     /etc/krb5/krb5.conf                       O_RDONLY      4
3257  sshd         0     0     /etc/krb5/krb5.keytab                     O_RDONLY      -1
3257  sshd         0     0     /etc/krb5/krb5.conf                       O_RDONLY      4
3259  sshd         0     0     /etc/gss/mech                             O_RDONLY      7
3259  sshd         0     0     /etc/gss/mech.d/                          O_RDONLY      7
^C
```

3 Exercício II - Alínea A

O objetivo deste exercício é escrever um programa que monitorize os processos que estão a correr no sistema, apresentando, após ser interrompido, as agregações obtidas relativamente às seguintes informações:

- número de tentativas de abrir ficheiros existentes;
- número de tentativas de criar ficheiros;
- número de tentativas bem-sucedidas.

3.1 Desenvolvimento

Uma vez que pretendemos contabilizar tentativas de acesso a ficheiros e número de sucessos, rapidamente identificamos as probes a usar: ***syscall::openat*:entry*** e ***syscall::openat*:return***.

Comecemos por analisar a primeira probe. Aquando da tentativa de aceder a um ficheiro, interessa-nos saber se o modo de abertura contém a flag que identifica a criação de ficheiro ou não. Esta informação passa a constar em forma de predicado na sonda, de forma a permitir distinguir a situação de criação de ficheiro da situação de simples leitura. Assim, na sonda que dispara sempre que se tenta abrir um ficheiro existente, consta o predicado **(arg2&O_CREAT) == 0** e na sonda que dispara aquando da criação de um ficheiro aplicou-se o predicado **(arg2&O_CREAT) == O_CREAT**.

No que toca ao comportamento das sondas em si, é idêntico para os dois casos. Definimos localmente o valor de uma variável **creat** - 0 para tentativas de abrir ficheiros e 1 para tentativas de criar ficheiros. Depois, cria-se em cada caso uma agregação que contabiliza o número de vezes que a sonda dispara. Inicialmente as agregações foram implementadas sem o uso de *keywords*, no entanto, após termos feito uma primeira análise dos resultados, achamos por bem alterar as estatísticas de forma a que dissessem respeito a um só comando, ao invés de serem uma contagem total de tentativas de acesso no sistema, por uma questão de dar mais significância aos resultados. Assim, foi acrescentada a *keyword* **execname** a todas as agregações usadas.

Passando agora para a probe **return**, também aqui devemos fazer a distinção do modo de abertura do ficheiro, através de um predicado que averigua o valor da variável **self->creat**, que foi definida por nós. Para além disso, uma vez que apenas queremos contabilizar o número de tentativas bem sucedidas, em ambos os casos incluímos no predicado a condição que obriga a variável **arg1** - que, como já sabemos, define o valor de retorno da chamada de sistema - seja maior que 0.

Em termos do comportamento das sondas em si, ambas definem apenas uma agregação própria que contabiliza os disparos da probe. Posto isto, ficam definidas todas as sondas necessárias para responder à questão apresentada.

Aquando da terminação do programa, os valores recolhidos pelas agregações são imprimidos no ecrã, tal como podemos ver na próxima secção.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

/* Exercicio 2a - Mostrar estatisticas dos processos a correr por iteracao */

syscall::openat*:entry
/(arg2&O_CREAT) == 0/
{
    self->creat = 0;
    @tryOpen[execname] = count();
}

syscall::openat*:entry
/(arg2&O_CREAT) == O_CREAT/
{
    self->creat = 1;
    @tryCreat[execname] = count();
}

syscall::openat*:return
/arg1 >= 0 && self->creat == 0/
{
   @successOpen[execname] = count();
}

syscall::openat*:return
/arg1 >= 0 && self->creat == 1/
{
   @successCreat[execname] = count();
}

END
{
```

```
   printf("EXECNAME    #OPEN_TRY   #OPEN_SUCCESS    #CREAT_TRY   #CREAT_SUCCESS\n");
   printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - ");
   printa(@tryOpen,@successOpen,@tryCreat,@successCreat);
}
```

3.2 Output

O programa foi posto a correr e, numa nova janela do terminal, executaram-se os comandos ***vim newfile.c*** e ***cat newfile.c***, nesta ordem, de forma a criar e de seguida abrir um ficheiro. O output abaixo demonstra o resultado fornecido pelo programa no momento em que se executaram os comandos anteriormente referidos. Como podemos ver, surgem os nomes de ambos os comandos usados; sendo que para o **vim** surgem estatísticas de abertura e criação de ficheiros, ao passo que o programa **cat** apenas fez tentativas de abertura de ficheiros.

```
^C
EXECNAME           #OPEN_TRY   #OPEN_SUCCESS    #CREAT_TRY   #CREAT_SUCCESS
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  bash             1           1                0            0
  sstored          1           1                0            0
  sshd             2           2                0            0
  cat              4           2                0            0
  init             4           4                4            4
  nscd             8           8                0            0
  vim              111         45               5            5
  vmtoolsd         331         325              1            1
```

4 Exercício II - Alínea B

Nesta alínea, era pedido que se implementasse um programa que, periodicamente, mostrasse as estatísticas referidas no ponto anterior por cada processo, identificando o respetivo pid e nome do comando.

4.1 Desenvolvimento

Em termos dos resultados apresentados, esta pergunta difere da anterior no sentido em que, para além de mostrarmos o nome do comando, passamos também a imprimir o pid do processo. Para além disso, os resultados são apresentados periodicamente e não apenas após o término do programa, e também passamos a imprimir a data atual juntamente com as estatísticas recolhidas.

No que toca a apresentar os resultados periodicamente, o provider *profile* disponibiliza aquilo a que se chama *tick-n* probes, sondas que disparam segundo um intervalo de tempo especificado. Ou seja, para definir uma sonda que dispare a cada 10 segundos bastaria escrever **tick-10sec**. No nosso caso, pretendemos definir um probe que dispare segundo um período de tempo (em segundos), que é passado ao programa como argumento. Para isso, escrevemos **tick-$1sec**, sendo o valor do argumento denotado pela variável macro $1.

Por fim, para apresentar a hora e dia atual com um formato legível, basta imprimir a variável **walltimestamp** de cada vez que dispara a nossa sonda.

Posto isto, ficamos com o programa que se segue.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

/* Exercicio 2b - Mostrar estatisticas dos processos a correr a cada $1 segundos */

syscall::openat*:entry
/(arg2&O_CREAT) == 0/
{
   self->creat = 0;
```

```
    @tryOpen[pid,execname] = count();
}

syscall::openat*:entry
/(arg2&O_CREAT) == O_CREAT/
{
   self->creat = 1;
    @tryCreat[pid,execname] = count();
}

syscall::openat*:return
/arg1 >= 0 && self->creat == 0/
{
   @successOpen[pid,execname] = count();
}

syscall::openat*:return
/arg1 >= 0 && self->creat == 1/
{
   @successCreat[pid,execname] = count();
}

tick-$1sec
{
    printf("%-20Y\n", walltimestamp);
    printf("PID      EXECNAME                #OPEN_TRY   #OPEN_SUCCESS    #CREAT_TRY
        #CREAT_SUCCESS\n");
    printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - ");
    printa(@tryOpen,@successOpen,@tryCreat,@successCreat);
    printf("\n\n");
    clear(@tryOpen);
    clear(@successOpen);
    clear(@tryCreat);
    clear(@successCreat);
}
```

4.2 Output

Se deixarmos o programa simplesmente correr, podemos ver um output como o que se segue surgir periodicamente, de acordo com o intervalo de tempo especificado na linha de comandos. Neste caso, o input dado foi um período de 5 segundos, tendo sido corrido com o comando ***./pstat-tick.d 5***.

```
2020 May 12 21:19:17
PID       EXECNAME        #OPEN_TRY   #OPEN_SUCCESS    #CREAT_TRY   #CREAT_SUCCESS
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    25553 sshd               1             1             0             0
      708 sshd               3             3             0             0
    25589 sshd               3             3             0             0
        1 init               4             4             4             4
     1212 vmtoolsd           6             2             0             0
      905 sstored            11            11            0             0
    25587 sshd               81            73            0             0


2020 May 12 21:19:22
PID       EXECNAME        #OPEN_TRY   #OPEN_SUCCESS    #CREAT_TRY   #CREAT_SUCCESS
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        1 init               0             0             0             0
      708 sshd               0             0             0             0
     1212 vmtoolsd           0             0             0             0
    25553 sshd               0             0             0             0
    25587 sshd               0             0             0             0
    25589 sshd               0             0             0             0
      894 hald-addon-acpi    1             1             0             0
    25583 pstat-tick.d       2             2             0             0
```

```
    905  sstored          24              24              33              33
```

5 Exercicio III

No terceiro e último exercício prático deste TP pretende-se simular o comportamento dos programas **strace -c** ou **truss**. Estes comandos aceitam um programa com argumento e recolhem estatísticas relativas a chamadas de *system calls* feitas pelo mesmo. Pretende-se que. com o uso da ferramenta dtrace, se construa um script que replique este comportamento.

5.1 Desenvolvimento

Todas as informações que precisamos de recolher são fornecidas pelo provider *syscall*. Uma vez que queremos apanhar todas as funções não precisamos de especificar o nome das funções nem dos módulos. Assim, é necessário apenas recolher informação no ponto de entrada e no ponto de saída da função pelo que usaremos as seguintes probes.

```
syscall:::entry
syscall:::return
```

Para evitar que sejam recolhidos dados de outros programas, usamos no predicado de cada probe a seguinte cláusula.

```
/pid == $target/
```

Os programas a simular recolhem estatísticas relativas ao tempo de execução, número de invocações e número de erros. Para contar o número de invocações, recorremos ao uso da funcionalidade *aggregate* com a função *count*. O nome da função que foi apanhada pela probe pode ser acedido na variável *probefunc*. Para além disso, é necessário guardar o valor da variável *timestamp* para calcular o tempo despendido em cada system call. O ponto entry fica então descrito da seguinte maneira.

```
syscall:::entry
/pid == $target/
{
    @count[probefunc] = count();
    self->ts = timestamp;
    sys_calls += 1;
}
```

Tendo já calculado o número de ocorrências, o próximo passo é tratar do tempo de execução. Já que, no ponto de entry, guardamos o timestamp inicial, para calcular o tempo total, basta que no ponto de return se subtraia esse valor ao timestamp final.

```
syscall:::return
/pid == $target && self->ts != 0 && errno == 0/
{
    @time[probefunc] = sum((timestamp - self->ts) / 100000)
    self->ts = 0;
}
```

Finalmente, para recolher o número de erros, verificamos o valor da variável errno. Caso esta seja superior a 0 significa que houve um erro. Adicionamos esta condição ao predicado e, juntamente com um contador, temos o número de erros de cada função.

```
syscall:::return
/pid == $target && self->ts != 0 && errno > 0/
{
    @error[probefunc] = count();
```

```
    @time[probefunc] = sum((timestamp - self->ts) / 100000)
    self->ts = 0;
    n_erros += 1;
}
```

Por fim, imprimimos os valores escolhidos. O script completo pode ser visto em baixo.

```
BEGIN
{
  start = timestamp;
  n_erros = 0;
  sys_calls = 0;
}

syscall:::entry
/pid == $target/
{
  @count[probefunc] = count();
  self->ts = timestamp;
  sys_calls += 1;
}

syscall:::return
/pid == $target && self->ts != 0 && errno == 0/
{
  @time[probefunc] = sum((timestamp - self->ts) / 1000000);
  self->ts = 0;
}

syscall:::return
/pid == $target && self->ts != 0 && errno > 0/
{
  @error[probefunc] = count();
  @time[probefunc] = sum((timestamp - self->ts) / 1000000);
  self->ts = 0;
  n_erros += 1;
}

END
{
  printf("syscall            milliseconds  calls errors\n");
  printa("%-20s %12@d %6@d %8@d\n",@time,@count,@error);
  printf("                   ------------  ----- ------\n");
  printf("sys totals:              %d     %d     %d\n",(timestamp-start) /
      1000000,sys_calls,n_erros);
}
```

5.2 Resultados Obtidos

Resultado após executar dtrace -q -s syscall.d -c ./test.sh onde test.sh faz sleep 3.

```
syscall            milliseconds  calls errors
getrlimit                     0      1       0
lwp_private                   0      1       0
readlinkat                    0      1       0
rexit                         0      1       0
schedctl                      0      1       0
setpgrp                       0      1       0
sigpending                    0      1       0
memcntl                       0      1       1
close                         0      2       0
```

```
getgid                    0       2       0
getuid                    0       2       0
lwp_sigmask               0       2       0
read                      0       2       0
umask                     0       2       0
setcontext                0       3       0
sigaction                 0       3       0
brk                       0       4       0
getpid                    0       4       0
lseek                     0       4       2
fcntl                     0       5       0
mmap                      0       5       0
ioctl                     0       6       1
sysconfig                 0       7       0
openat                    0      11      10
fstatat                   0      33      12
spawn                     1       1       0
waitsys                3004       4       3
                  ------------  -----  ------
sys totals:            3029     110      29
```

6 Conclusão

Este trabalho introduziu-nos à poderosa ferramenta DTrace. Embora um pouco superficial, permitiu-nos não só compreender as potencialidades da ferramenta, mas também trabalhar num um novo paradigma de estudo onde analisamos os problemas de uma perspetiva de fora para dentro ao contrário do habitual. Para trabalho futuro, era interessante recorrer a esta ferramenta para estudar um aplicação já construída ao invés de estudar apenas *system calls* e analisar algumas conclusões que podemos retirar com os dados recolhidos.