TPC3: Desenvolvimento de Programas em DTrace - Engenharia dos sistemas de Computação -

autor: Daniel Malhadas

***Resumo*—O presente documento apresenta um estudo aprofundado sobre a ferramenta DTrace. Para isso foram realizados vários programas onde se tem o intuito de explorar várias das funcionalidades desta ferramenta. Estes programas permitem fazer traçados de chamadas de sistema e assim organizar informação relativa à sua instrumentação, que foi depois cuidadosamente estudada de forma a comprovar a correção dos programas desenvolvidos e assim verificar uma boa aprendizagem da ferramenta em questão. Note-se ainda que todo este estudo foi realizado em ambiente Solaris 11 e, por essa razão, os dados instrumentados serão relativos somente a esse ambiente.**

***Index Terms*—DTrace, Solaris 11, Computação Paralela e Distribuída.**

I. Introdução

A. Contextualização e Motivação

Com este projeto pretendeu-se escrever programas em DTrace que permitissem fazer o traçado de chamadas ao sistema. De seguida vários testes são realizados com o intuito de comprovar a correção dos programas desenvolvidos. Os resultados destes testes foram depois organizados e apresentados ao longo deste documento. A razão do desenvolvimento deste projeto é então o de melhor entender o uso e a utilidade da ferramenta DTrace e, já que os testes são realizados em ambiente Solaris 11, entender também melhor como funcionam as chamadas ao sistema e a própria ferramenta DTrace neste ambiente específico.

B. O que é a ferramenta DTrace

A Oracle Solaris DTrace é uma ferramenta de rastreio/traçado avançada usada para testar troços problemáticos de um programa em tempo real. Para isto, esta ferramenta permite observar questões de performance tanto em pequenas aplicações como do próprio sistema operativo em si de forma dinâmica e segura permitindo a quem a utilize a identificação de problemas que seriam difíceis de detetar com outras ferramentas similares por estarem demasiado disfarçados sobre várias camadas de software.
A ferramenta permite também a instrumentação de várias estatísticas em tempo real relativas ao programa a testar. Estatísticas como: consumo de memória, tempo de CPU despendido, que chamadas de função foram realizadas, etc.[1] De forma a poder traçar o que está a acontecer, a ferramenta DTrace recorre à monitorização de diversos "sinais"/"pontos de interesse"marcados no sistema operativo a que se dá o nome de **probes**. Estes probes são lançados em momentos específicos e, cabe ao utilizador da ferramenta DTrace, saber quais os probes que deve interceptar e o que fazer quando os intercetar de forma a alcançar os resultados que pretende. Segue-se uma tabela com os probes disponíveis em ambiente Solaris 11 e uma breve descrição: [2]

Common DTrace Providers	Description
dtrace	Start, end and error probes
syscall	Entry and return probes for all system calls
fbt	Entry and return probes for all kernel calls
profile	Timer driven probes
proc	Process creation and lifecycle probes
pid	Entry and return probes for all user-level processes
io	Probes for all I/O related events
sdt/usdt	Developer defined probes at arbitrary locations/names within source code for kernel and user-level processes
sched	Probes for scheduling related events
lockstat	Probes for locking behavior within the operating system

II. Programas Desenvolvidos

A. Desenvolvimento do Programa 1

De início quis-se desenvolver um programa que permitisse fazer o traçado das chamadas ao sistema open() (como em Solaris 11 não existem probes para fazer o traçado de open fazemos antes o traçado foi de openat() que é uma chamada de sistema mais geral que engloba também chamadas de sistema open()[2]). entendeu-se que este programa deveria ser capaz de imprimir a seguinte informação por linha:

- **1 -** Nome do ficheiro executável e respetivos: PID do processo, UID do utilizador e GID do grupo;
- **2 -** Caminho absoluto para o ficheiro que for aberto;
- **3 -** A cadeia de caracteres com as "flags"da chamada ao sistema openat(), O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT;
- **4 -** O valor de retorno da chamada de sistema.

O programa desenvolvido encontra-se apresentado de seguida:

```
1 #!/usr/sbin/dtrace -s
2 #pragma D option quiet
3
4 /**
5  * Tracer de chamadas ao sistema openat().
6  * int openat(int fildes, const char *path, int oflag, mode_t mode);
7  **/
8
9 dtrace:::BEGIN {                                                    (1)
10     printf("Tracer de chamadas ao sistema openat().\n");
11     printf("_____________________________________________");
12     printf("_____________________________________________\n");
13     printf("| %s, %s, %s, %s, %s, %s, %s\n",
14            "Executable", "Path", "Flags", "PID", "UID", "GID", "Return");
15     printf("_____________________________________________");
16     printf("_____________________________________________\n");
17 }
18
19 /*Catch openat system call entry*/
20 syscall::openat*:entry {                                            (2)
21     /*Print executable name and absolute path*/
22     printf("| %s, %s", execname, copyinstr(arg1));
23
24     printf("%s", arg2 & O_RDONLY ? ", O_RDONLY" :
25                 (arg2 & O_WRONLY ? ", O_WRONLY" : ", O_RDWR") );
26     printf("%s%s", arg2 & O_APPEND ? "|O_APPEND" : "",
27                    arg2 & O_CREAT  ? "|O_CREAT"  : "");
28 }
29
30 /*Catch apenat systema call return*/
31 syscall::openat*:return {                                           (3)
32     /*Print pid, uid, gid and return value*/
33     printf(", %d, %d, %d, %d\n", pid, uid, gid, arg1);
34     printf("_____________________________________________");
35     printf("_____________________________________________\n");
36 }
```

Note-se como o programa desenvolvido se encontra dividido em três blocos principais. No ponto circundado 1 podemos observar o primeiro bloco que indica o que fazer quando o DTrace se encontra perante o probe **dtrace:::BEGIN**. Este probe é um probe especial lançado aquando do início da execução do DTrace e pode portanto ser usado para inicializar variáveis ou imprimir cabeçalhos. Neste caso este probe está a ser usado para imprimir um pequeno cabeçalho.

No segundo ponto circundado estamos já a fazer o traçado de aberturas de ficheiro em tempo real e vemos já a utilização de outro probe distinto chamado **syscall:::entry** que é lançado no início de uma chamada de sistema. Como damos a informação **openat*** então as únicas chamadas de sistema que o nosso programa irá notar são as da chamada de sistema openat().

Vemos ainda o uso da variável *"builtin"* **execname** que contém o nome do processo a ser executado. Temos também acesso aos argumentos fornecidos ao probe que, para o probe entry, representam os argumentos da chamada de sistema realizada. Neste bloco vemos **arg1** que é um exemplo de um desses argumentos e representa o caminho absoluto para o ficheiro em questão. No entanto, os argumentos do probe são sempre todos inteiros sem sinal (uint64_t) e, de forma a obter-se a string dada em *"user-level"* como argumento para a chamada de sistema num local legível pelo DTrace (kernel), usa-se **copyinstr()**. Desta forma indicamos explicitamente que o número **arg1** deve ser interpretado como uma string. **arg2** é também um argumento do probe que simboliza as *"flags"* dadas como argumento à chamada de sistema. Esta variável permite saber as *"flags"* de permissão que estão envolvidas na abertura do ficheiro em questão.

Por fim vemos no terceiro ponto circundado a utlização do probe **syscall:::return** . Como se fornece a indicação **openat*** então este probe é lançado aquando do fim de chamadas de sistema openat(). Novamente temos acesso aos argumento do probe, mas desta vez **arg1** irá simbolizar o valor de retorno da chamada de sistema realizada. Fazemos ainda a impressão do **PID** do processo, **UID** do utilizador e o **GID** do grupo que são obtidos com variáveis *"builtin"* de nome homónimo. Antes de avançarmos é importante ainda referir que, como o ambiente utilizado é Solaris 11, estes probes lançados para as chamadas de sistema openat() englobam também chamadas de sistema open() e as suas versões de 64-bit openat64() e open64().[2]

B. Testes para Programa 1

Depois de desenvolvido o programa descrito anteriormente este foi testado minuciosamente. Inicialmente observou-se o output depois de correr o seguinte comando: **dtrace -qs programa1.d » output.txt**, com este comando foi possível observar o comportamento do programa com o funcionamento normal do sistema operativo usado. Uma imagem de um excerto do output fornecido pode ser agora observada:

```
Tracer de chamadas ao sistema openat().
| Executable, Path, Flags, PID, UID, GID, Return
| nfsmapid, /etc/resolv.conf, O_RDWR, 10327, 1, 12, 8
| nscd, /system/volatile/repository_door, O_RDWR, 488, 0, 0, 12
| nscd, /system/volatile/repository_door, O_RDWR, 488, 0, 0, 12
| nscd, /system/volatile/repository_door, O_RDWR, 488, 0, 0, 12
| gnome-settings-d, /etc/vfstab, O_RDWR, 20218, 101, 10, 22
| gnome-settings-d, /etc/vfstab, O_RDWR, 7232, 502, 10, 22
| gnome-settings-d, /etc/vfstab, O_RDWR, 6983, 0, 0, 23
| gnome-settings-d, /etc/vfstab, O_RDWR, 6091, 0, 0, 24
| nfsmapid, /etc/resolv.conf, O_RDWR, 10327, 1, 12, 8
| utmpd, /system/volatile/utmpx, O_RDWR|O_CREAT, 217, 1, 12, 5
| utmpd, /system/volatile/utmpx, O_RDWR, 217, 1, 12, 6
| utmpd, /proc/28596/psinfo, O_RDWR, 217, 1, 12, 7
| utmpd, /proc/6343/psinfo, O_RDWR, 217, 1, 12, 7
| utmpd, /proc/28660/psinfo, O_RDWR, 217, 1, 12, 7
```

Ao observarmos este output o programa parece funcionar corretamente, no entanto testou-se ainda com o comando cat de 4 formas distintas. Os comandos utilizados e o output fornecido podem ser agora consultados de forma a provar assim a correção do programa descrito:

cat /etc/inittab > /tmp/test72293

```
Tracer de chamadas ao sistema openat().
| Executable, Path, Flags, PID, UID, GID, Return
| bash, /tmp/test72293, O_WRONLY|O_CREAT, 28795, 40196, 5018, 4
| cat, /var/ld/ld.config, O_RDWR, 28795, 40196, 5018, -1
| cat, /lib/libc.so.1, O_RDWR, 28795, 40196, 5018, 3
| cat, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28795, 40196, 5018, 3
| cat, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28795, 40196, 5018, 3
| cat, /etc/inittab, O_RDWR, 28795, 40196, 5018, 3
```

cat /etc/inittab » /tmp/test72293

```
Tracer de chamadas ao sistema openat().
| Executable, Path, Flags, PID, UID, GID, Return
| bash, /tmp/test72293, O_WRONLY|O_APPEND|O_CREAT, 28821, 40196, 5018, 4
| cat, /var/ld/ld.config, O_RDWR, 28821, 40196, 5018, -1
| cat, /lib/libc.so.1, O_RDWR, 28821, 40196, 5018, 3
| cat, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28821, 40196, 5018, 3
| cat, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28821, 40196, 5018, 3
| cat, /etc/inittab, O_RDWR, 28821, 40196, 5018, 3
```

cat /etc/inittab | tee /tmp/test72293

```
Tracer de chamadas ao sistema openat().
| Executable, Path, Flags, PID, UID, GID, Return
| tee, /var/ld/ld.config, O_RDWR, 28826, 40196, 5018, -1
| tee, /lib/libc.so.1, O_RDWR, 28826, 40196, 5018, 3
| tee, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28826, 40196, 5018, 3
| tee, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28826, 40196, 5018, 3
| tee, /tmp/test72293, O_WRONLY|O_CREAT, 28826, 40196, 5018, 3
| cat, /var/ld/ld.config, O_RDWR, 28825, 40196, 5018, -1
| cat, /lib/libc.so.1, O_RDWR, 28825, 40196, 5018, 3
| cat, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28825, 40196, 5018, 3
| cat, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28825, 40196, 5018, 3
| cat, /etc/inittab, O_RDWR, 28825, 40196, 5018, 3
```

cat /etc/inittab | tee -a /tmp/test72293

```
Tracer de chamadas ao sistema openat().
| Executable, Path, Flags, PID, UID, GID, Return
| cat, /var/ld/ld.config, O_RDWR, 28849, 40196, 5018, -1
| cat, /lib/libc.so.1, O_RDWR, 28849, 40196, 5018, 3
| cat, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28849, 40196, 5018, 3
| cat, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28849, 40196, 5018, 3
| cat, /etc/inittab, O_RDWR, 28849, 40196, 5018, 3
| tee, /var/ld/ld.config, O_RDWR, 28850, 40196, 5018, -1
| tee, /lib/libc.so.1, O_RDWR, 28850, 40196, 5018, 3
| tee, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28850, 40196, 5018, 3
| tee, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28850, 40196, 5018, 3
| tee, /tmp/test72293, O_WRONLY|O_APPEND|O_CREAT, 28850, 40196, 5018, 3
| tee, /usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3, O_RDWR, 28850, 40196, 5018, 3
| tee, /usr/lib/locale/en_US.UTF-8/methods_unicode.so.3, O_RDWR, 28850, 40196, 5018, 3
| tee, /tmp/test72293, O_WRONLY|O_APPEND|O_CREAT, 28850, 40196, 5018, 3
```

Depois de observar estes 4 exemplos podemos perceber que realmente o programa se encontra correto. Nota-se, por exemplo, um tratamento correto da impressão das flags usadas ao notar a diferença entre as flags do primeiro e do segundo exemplo para o executável de nome bash, já que no segundo o uso de » leva a que se adicione a flag **O_APPEND**.

C. Extensão ao Programa 1

Seguidamente teve-se a intenção de modificar o programa criado para que apenas se faça o traceamento de chamadas de sistema openat() relativas a ficheiros com **"/etc"** no caminho sejam detetados. O programa 1 final ficou então como se segue em baixo:

```
1  #!/usr/sbin/dtrace -s
2  #pragma D option quiet
3
4  /**
5   * Tracer de chamadas ao sistema openat().
6   * int openat(int fildes, const char *path, int oflag, mode_t mode);
7   **/
8
9  dtrace:::BEGIN {
10     printf("Tracer de chamadas ao sistema openat().\n");
11     printf("_____________________________________________");
12     printf("_____________________________________________\n");
13     printf("| %s, %s, %s, %s, %s, %s, %s\n",
14             "Executable", "Path", "Flags", "PID", "UID", "GID", "Return");
15     printf("_____________________________________________");
16     printf("_____________________________________________\n");
17 }
18
19 /*Catch openat system call entry*/
20 syscall::openat*:entry {
21
22     self->print_ret = (strstr(copyinstr(arg1), "/etc") != NULL) ? 1 : 0;
23     self->path = copyinstr(arg1);
24     self->flag = arg2;
25 }
26
27 /*Catch openat system call return*/
28 syscall::openat*:return
29 /self->print_ret == 1/ {
30
31     /*Print executable name and absolute path*/
32     printf("| %s, %s", execname, self->path);
33
34     printf("%s", self->flag & O_RDONLY ? ", O_RDONLY" :
35                  (self->flag & O_WRONLY ? ", O_WRONLY" : ", O_RDWR") );
36     printf("%s%s", self->flag & O_APPEND ? "|O_APPEND" : "",
37                    self->flag & O_CREAT  ? "|O_CREAT"  : "");
38
39     /*Print pid, uid, gid and return value*/
40     printf(", %d, %d, %d, %d\n", pid, uid, gid, arg1);
41     printf("_____________________________________________");
42     printf("_____________________________________________\n");
43 }
```

Podemos observar no ponto circundado 1 como se usa uma estrutura condicional de forma a registar numa variável o valor 1 ou o valor 0. Caso seja 1 então, aquando do return da chamada de sistema em questão, isso é detetado pelo predicado no ponto circundado 2, sabemos assim que o caminho absoluto contém **"/etc"** e devemos, portanto, imprimir a sua informação.

D. Testes à extensão do Programa 1

Depois de extendido o programa 1 observou-se o output depois de correr o seguinte comando: **dtrace -qs programa1extendido.d » output.txt**, com este comando foi possível observar o comportamento do programa com o funcionamento normal do sistema operativo usado. Uma imagem de um excerto do output fornecido pode ser agora observada:

```
Tracer de chamadas ao sistema openat().
| Executable, Path, Flags, PID, UID, GID, Return
| nfsmapid, /etc/resolv.conf, O_RDWR, 10327, 1, 12, 8
| gnome-settings-d, /etc/vfstab, O_RDWR, 20218, 101, 10, 22
| gnome-settings-d, /etc/vfstab, O_RDWR, 7232, 502, 10, 22
| gnome-settings-d, /etc/vfstab, O_RDWR, 6091, 0, 0, 24
| gnome-settings-d, /etc/vfstab, O_RDWR, 6983, 0, 0, 23
| nfsmapid, /etc/resolv.conf, O_RDWR, 10327, 1, 12, 8
| sshd, /etc/system.d/crypto:fips-140, O_RDWR, 29180, 0, 0, -1
| sshd, /etc/crypto/pkcs11.conf, O_RDWR, 29180, 0, 0, 4
| sshd, /etc/system.d/crypto:fips-140, O_RDWR, 29180, 0, 0, -1
| sshd, /etc/gss/mech, O_RDWR, 29180, 0, 0, 4
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
| sshd, /etc/krb5/krb5.conf, O_RDWR, 29180, 0, 0, 4
| sshd, /etc/ssh/moduli, O_RDWR, 29180, 0, 0, 4
```

Vemos realmente que só aparecem entradas em que o caminho absoluto contenha **"/etc"** no entanto isto ainda não comprova a validade do programa pois, por uma enorme coincidência, pode não ter havido nenhum caso com um caminho absoluto sem **"/etc"**. De forma a provar a correção do programa fez-se correr novamente o comando **cat /etc/inittab | tee -a /tmp/test72293** já que podemos comparar com o output fornecido pela versão anterior do programa. Se esta extensão estiver correta então, olhando para o output anterior, vemos que deve ser imprimida apenas uma entrada. Olhamos agora o output obtido:

```
Tracer de chamadas ao sistema openat().

| Executable, Path, Flags, PID, UID, GID, Return

| cat, /etc/inittab, O_RDWR, 29187, 40196, 5018, 3
```

Vemos que realmente apenas imprimiu a entrada esperada, fica então assim provada a correção do programa.

E. Desenvolvimento do Programa 2

Para um segundo estudo quis-se desenvolver um programa que, para os processos que estão a correr no sistema, apresenta os valores obtidos em cada iteração das seguintes estatísticas:

1 - Número de tentativas de abrir ficheiros existentes;

2 - Número de tentativas para criar ficheiros;

3 - Número de tentativas bem sucedidas.

Apresenta-se agora o programa desenvolvido:

```
1  #!/usr/sbin/dtrace -s
2  #pragma D option quiet
3
4  /**
5   * Tracer de chamadas ao sistema openat() que conta o numero.
6   * de tentativas e de sucessos.
7   * int openat(int fildes, const char *path, int oflag, mode_t mode);
8   **/
9
10 dtrace:::BEGIN {                                          (1)
11     printf("#Tentativas e #sucessos.\n");
12     printf("_____________________________________________");
13     printf("_____________________________________________\n");
14 }
15
16 /*Catch openat system call entry*/
17 syscall::openat*:entry                                    (2)
18 /(arg2 & O_CREAT) == 0/ {
19     /*Entry that tries to open an existing file*/
20     @try_open[pid]    = count();
21     printf("%s on PID: %d, open try.\n", execname, pid);
22 }
23
24 /*Catch openat system call entry*/
25 syscall::openat*:entry                                    (3)
26 /(arg2 & O_CREAT) == O_CREAT/ {
27     /*Entry that tries to create a file*/
28     @try_create[pid]    = count();
29     printf("%s on PID: %d, create try.\n", execname, pid);
30 }
31
32 /*Catch openat system call return*/
33 syscall::openat*:return                                   (4)
34 /arg1 >= 0/ {
35     /*Count successes*/
36     @success[pid] = count();
37     printf("%s on PID: %d, success.\n", execname, pid);
38 }
39
40 /*print results*/
41 dtrace:::END {                                            (5)
42     /*Print*/
43     printf("_____________________________________________");
44     printf("_____________________________________________\n");
45     printf("| %8s | %5s | %5s | %10s |\n",
46            "PID", "Open", "Create", "Successes");
47     printf("_____________________________________________");
48     printf("_____________________________________________\n");
49     printa("| %8d | %@5d | %@5d | %@10d |\n",
50            @try_open, @try_create, @success);
51     printf("_____________________________________________");
52     printf("_____________________________________________\n");
53
54     /*Clear Counters*/
55     clear(@try_open);                                      (6)
56     clear(@try_create);
```

(7)

Note-se como no ponto circundado 1 temos novamente um bloco que indica o que fazer ao encontrar um probe **dtrace:::BEGIN**. Novamente este bloco é usado apenas para imprimir um cabeçalho.

De seguida observamos que temos dois blocos para o probe **syscall:::entry** relativo à chamada de sistema openat(). No bloco do ponto circundado 2 é usado um predicado que verifica que a chamada de sistema é relativa à abertura de um ficheiro já existente, como se pode ver pela condição **(arg2 & O_CREAT) == 0**. No ponto circundado 3 vemos outro bloco que verifica se a chamada de sistema refere a criação de um ficheiro, como vemos pelo predicado com a condição **(arg2 & O_CREAT) == O_CREAT**. Depois de identificar que tipo de acesso ao ficheiro se está a tentar fazer, é necessário atualizar essa informação nos respetivos contadores que irão acumular o valor de tentativas de cada uma destas duas possibilidades. Estas tentativas de abertura e criação são registadas em duas variáveis distintas: **try_create** para a tentativa de criação e **try_open** para a tentativa de abertura. Vemos esta atualização destes acumuladores no ponto circundado 7. Observa-se que **count()** é usado com o intuito de contar o número de tentativas respetivas a cada variável e que, cada variável, tem atribuído a cada contador o **PID** do processo responsável pela chamada de sistema. Esta agregação de cada PID ao seu contador observa-

se aquando da atualização dos contadores como se evidencia, por exemplo, pelo seguinte troço de código: **@try_create[pid] = count();**.
No ponto circundado 4 temos que, ao encontrar um probe **syscall:::return** de uma chamada de sistema openat(), se verifica com recurso a um predicado com a condição **arg1 >= 0** se a chamada de sistema foi realizada com sucesso ou não (caso tenha devolvido um valor superior a -1 sabemos foi bem sucedida, sendo **arg1** o valor de retorno) e, caso seja um sucesso, usamos novamente **count()** para atualizar um outro acumulador, a variável **success** que se encontra, assim como os outros acumuladores mencionados anteriormente, relacionado com o PID do processo que realizou a chamada de sistema.
Note-se agora que sempre que há a atualização de um contador é feito um print que a descreve indicando também o **PID** e o **nome do executável** correspondentes. Desta forma é possível no fim validar os resultados acumulados obtidos como iremos ver na próxima secção deste documento.
Por fim, no ponto circundado 5, observamos o que fazer ao encontrar um probe do tipo **dtrace:::END**. Aqui será feita a impressão dos contadores que foram sendo atualizados ao longo da execução do programa sendo que depois da impressão, no ponto circundado 6, estes contadores são eliminados com o uso de **clear**.

F. Testes para Programa 2

Depois de desenvolvido o programa descrito este foi testado com o seguinte comando **dtrace -qs programa2a.d**. Segue-se uma imagem do output obtido que comprova a sua correção:

```
#Tentativas e #sucessos.

gnome-settings-d on PID: 20218, open try.
gnome-settings-d on PID: 20218, success.
gnome-settings-d on PID: 7232, open try.
gnome-settings-d on PID: 7232, success.
gnome-settings-d on PID: 6091, open try.
gnome-settings-d on PID: 6091, success.
gnome-settings-d on PID: 6983, open try.
gnome-settings-d on PID: 6983, success.
nfsmapid on PID: 10327, open try.
nfsmapid on PID: 10327, success.
nscd on PID: 488, open try.
nscd on PID: 488, success.
init on PID: 1, open try.
init on PID: 1, success.
init on PID: 1, create try.
init on PID: 1, success.
init on PID: 1, create try.
init on PID: 1, success.
init on PID: 1, open try.
init on PID: 1, success.
nfsmapid on PID: 10327, open try.
nfsmapid on PID: 10327, success.
nscd on PID: 488, open try.
nscd on PID: 488, success.
^C

|      PID |  Open | Create |  Successes |

|     6091 |     1 |      0 |          1 |
|     6983 |     1 |      0 |          1 |
|     7232 |     1 |      0 |          1 |
|    20218 |     1 |      0 |          1 |
|      488 |     2 |      0 |          2 |
|    10327 |     2 |      0 |          2 |
|        1 |     2 |      2 |          4 |
```

É possível ver na imagem impressões intermédias que vão indicando as tentativas e os sucessos que se vão encontrando em tempo real ao longo da execução do programa assim como o seu PID e nome de executável respetivos. Depois de o programa terminar vemos uma tabela com os valores agregados que podem ser confirmados com as impressões anteriores. Se o leitor prestar atenção pode verificar que de facto as impressões estão coerentes, provando assim a correção do programa.

G. Extensão ao Programa 2

Seguidamente teve-se a intenção de modificar o programa 2 para que repetidamente, com um período especificado em segundos passado como argumento da linha de comandos, imprimisse a seguinte informação:

1 - Hora e dia atual em formato legível.
2 - As estatísticas recolhidas por PID e respetivo nome.

Para isto extendeu-se o programa 2 da seguinte forma:

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

/**
 * Tracer de chamadas ao sistema openat() que conta o numero.
 * de tentativas e de sucessos.
 * int openat(int fildes, const char *path, int oflag, mode_t mode);
 **/

dtrace:::BEGIN {
    printf("#Tentativas e #sucessos.\n");
    printf("_____________________________________________");
    printf("_____________________________________________\n");
}

/*Catch openat system call entry*/
syscall::openat*:entry
/(arg2 & O_CREAT) == 0/ {
    /*Entry that tries to open an existing file*/
    @try_open[execname, pid]        = count();
    @try_open_global[execname, pid] = count();
}

/*Catch openat system call entry*/
syscall::openat*:entry
/(arg2 & O_CREAT) == O_CREAT/ {
    /*Entry that tries to create a file*/
    @try_create[execname, pid]        = count();
    @try_create_global[execname, pid] = count();
}

/*Catch openat system call return*/
syscall::openat*:return
/arg1 >= 0/ {
    /*Count successes*/
    @success[execname, pid]        = count();
    @success_global[execname, pid] = count();
}

tick-$1s {                                          (1)
    /*Print*/
    printf("_____________________________________________");
    printf("_____________________________________________\n");
    printf("| %20s | %8s | %5s | %7s | %10s |\n",
           "Executable", "PID", "Open", "Create", "Successes");
    printf("_____________________________________________");
    printf("_____________________________________________\n");
    printa("| %20s | %8d | %@5d | %@7d | %@10d |\n",
           @try_open, @try_create, @success);
    printf("| TIME: %Y\n", walltimestamp);
    printf("_____________________________________________");
    printf("_____________________________________________\n");

    /*Discard last measurements*/
    trunc(@try_create);
    trunc(@try_open);                               (2)
    trunc(@success);
}

/*print results*/
dtrace:::END {
    /*Print*/
    printf("_____________________________________________");
    printf("_____________________________________________\n");
    printf("| %20s | %8s | %5s | %7s | %10s |\n",
           "Executable", "PID", "Open", "Create", "Successes");
    printf("_____________________________________________");
    printf("_____________________________________________\n");
    printa("| %20s | %8d | %@5d | %@7d | %@10d |\n",
           @try_open_global, @try_create_global, @success_global);
    printf("_____________________________________________");
    printf("_____________________________________________\n");

    /*Clear Counters*/
    clear(@try_open);
    clear(@try_create);
    clear(@success);                                (3)
    clear(@try_open_global);
    clear(@try_create_global);
    clear(@success_global);
}
```

Note-se a adição do bloco no ponto circundado 1. Este bloco indica o que fazer de **$1** em **$1** segundos, sendo **$1** o valor em segundos entregue pela linha de comandos, como era pretendido. No interior deste bloco vemos que são imprimidos os valores encontrados desde a última medição à **$1** segundos atrás. É também apresentada a data permitindo assim comprovar a correção desta funcionalidade. Esta data é obtida através de **walltimestamp**. De forma a reiniciar os contadores note-se como, no ponto circundado 2, se recorre ao uso de **trunc()**.Estas variáveis apresentadas reiniciadas de **$1** em **$1** segundos são relativas somente ao tempo desde a última medição, por isso, é necessário ter ainda outras variáveis que guardem os valores globais (como era feito na versão anterior). Ao olhar o código é possível notar que sempre que um acumulador é atualizado para a apresentação de **$1** em **$1** segundos, um acumulador global também o é para assim, quando se encontrar o probe **dtrace:::END** estes acumuladores globais serem apresentados corretamente. Note-se também que, enquanto na versão anterior cada acumulador estava associado ao **PID**, agora encontra-se associado não só ao **PID**, mas também ao **nome do executável**, como podemos ver pelo seguinte troço de código: **@success[execname, pid] = count();**. Desta forma é possível no fim mostar os valores agregados por estas duas variáveis, como era pretendido nos requisitos apresentados anteriormente.
Por fim, no ponto circundado 3, todas as variáveis agregadas a serem eliminadas com o uso de **clear()**.

H. Testes à extensão do Programa 2

Vemos agora o output do programa de forma a comprovar a sua correção, desta vez é usado o comando **dtrace -qs programa2b.d 5**:

```
#Tentativas e #sucessos.

|           Executable |      PID |  Open |  Create |  Successes |
| TIME: 2017 Apr  3 23:46:25

|           Executable |      PID |  Open |  Create |  Successes |
|     gnome-settings-d |     7232 |     1 |       0 |          1 |
|     gnome-settings-d |    20218 |     1 |       0 |          1 |
|              dtrace |    29344 |     2 |       0 |          2 |
|               nscd |      488 |     2 |       0 |          2 |
| TIME: 2017 Apr  3 23:46:30

|           Executable |      PID |  Open |  Create |  Successes |
|     gnome-settings-d |     6091 |     1 |       0 |          1 |
|     gnome-settings-d |     6983 |     1 |       0 |          1 |
| TIME: 2017 Apr  3 23:46:35

|           Executable |      PID |  Open |  Create |  Successes |
|            nfsmapid |    10327 |     1 |       0 |          1 |
| TIME: 2017 Apr  3 23:46:40

^C
|           Executable |      PID |  Open |  Create |  Successes |
|     gnome-settings-d |     6091 |     1 |       0 |          1 |
|     gnome-settings-d |     6983 |     1 |       0 |          1 |
|     gnome-settings-d |     7232 |     1 |       0 |          1 |
|     gnome-settings-d |    20218 |     1 |       0 |          1 |
|            nfsmapid |    10327 |     1 |       0 |          1 |
|              dtrace |    29344 |     2 |       0 |          2 |
|               nscd |      488 |     2 |       0 |          2 |
```

Como o argumento dado ao nosso programa foi o número 5 vemos que a cada 5 segundos são imprimidos os valores obtidos desde a última medição (há 5 segundos atrás). Prova-se a correção destas medições intermédias ao olharmos para o tempo imprimido que realmente mostra cada medição separada por um intervalo de 5 segundos. Vemos ainda o caso de nos últimos 5 segundos não ter ocorrido nenhuma chamada de sistema, que é o caso logo da primeira medição intermédia já que apenas se imprimiu o tempo. Notamos depois que o programa imprime no fim os valores agregados, como fazia na versão anterior. Podemos comprovar a correção destes valores ao olhar para as medições intermédias já que vemos que há

processos que aparecem mais do que uma vez e os seus valores aparecem somados no final como era suposto.

I. Desenvolvimento do Programa 3

Por último houve a intenção de desenvolver ainda um outro programa que permitisse replicar o comportamento do seguinte comando: **strace -c <programa>**. Onde a opção -c permite contabilizar o número de ocorrências de cada chamada ao sistema e o tempo despendido. Segue-se o código desenvolvido:

```
1  #!/usr/sbin/dtrace -s
2  #pragma D option quiet
3
4
5  dtrace:::BEGIN {                                          (1)
6      printf("#Chamadas de sistema.\n");
7      printf("_____________________________________");
8      printf("_____________________________________\n");
9      self->started = 0;
10 }
11
12 /*Catch system call entry*/
13 syscall:::entry
14 /execname == $$1/ {                                      (2)
15     @times_called[probefunc] = count();
16     self->started = timestamp;
17 }
18
19 /*Catch system call return*/
20 syscall:::return
21 /*Ignore also syscalls where entry was not traced*/
22 /execname == $$1 && self->started!=0/ {                   (3)
23     @syscall_time_elapsed[probefunc] = sum((timestamp - self->started));
24     self->started = 0;
25 }
26
27 /*print results*/
28 dtrace:::END {
29     /*Print*/
30     printf("Program: %s\n", $$1);                         (4)
31     printf("_____________________________________");
32     printf("_____________________________________\n");
33     printf("| %15s | %15s | %s\n",
34             "Sys Call", "times Called", "Time spent");
35     printf("_____________________________________");
36     printf("_____________________________________\n");
37     printa("| %15s | %@15d | %@d ns\n",
38             @times_called, @syscall_time_elapsed);
39     printf("_____________________________________");
40     printf("_____________________________________\n");
41
42     /*Clear Counters*/
43     clear(@times_called);                                 (5)
44     clear(@syscall_time_elapsed);
45 }
```

No ponto circundado 1 vemos um cabeçalho a ser impri-mido assim como a atribuição da variável **started** da estrutura **self** com o valor 0. Esta variável servirá para, quando uma chamada ao sistema começar, se poder registar a sua data de início. Torna-se importante defini-la a zero de imediato, pois, se este nosso programa começar depois de o programa argumento ter começado pode acontecer ser detetado o seu fim sem nunca ter sido detetado o seu início. Isto traria problemas pois o tempo que ficaria registado que a chamada ao sistema demorou seria erróneo, assim como o seu número de ocorrências já que este apenas é atualizado quando a chamada ao sistema é iniciada. Ao inicializar a variável **self->started** a 0 podemos verificar o seu valor com um predicado, como vemos no ponto circundado 3, com a condição **self->started!=0** e assim, não registar dados o tempo de fim da chamada ao sistema se um tempo inícial não tiver sido registado. No ponto circundado 3 vemos ainda como o tempo é agregado por chamada ao sistema (**@syscall_time_elapsed[probefunc]**), sendo o nome da chamada identificado pela variável *"builtin"* **probefunc**. Este tempo é atualizado com o uso de **sum()** que recebe como argumento o tempo que deve somar. No final deste bloco a variável **self->started** é novamente inicializada a 0 para assim o processo descrito se poder repetir.

No ponto circundado 2 vemos a variável agregada que irá acumular o número de vezes que cada chamada ao sistema foi invocada com o uso de **count()** e vemos o tempo inicial da chamada de sistema a ser registado em **self->started**.

No ponto circundado 4 observamos os valores agregados a serem imprimidos aquando da terminação do programa. No ponto circundado 5 vemos essas variáveis agregadas a serem por fim eliminadas com o uso de **clear()**.

Note-se ainda que houve a necessidade de, sempre que se refere a variável recebida como argumento na linha de comandos (o nome do programa a observar), este é referenciado da seguinte forma: **$$1**. Ao usarmos dois **$** estamos a indicar explicitamente que o argumento recebido deve ser interpretado como uma string. Temos interesse que isto aconteça para assim esta string argumento poder ser comparada com a string na variável **execname** nos pontos circundados 2 e 3.

J. Testes para Programa 3

Por fim testa-se o terceiro e último programa com o seguinte comando: **dtrace -qs programa3.d ls**. Note-se que ao fornecer **ls** como argumento, apenas serão imprimidas informações relativas a chamadas de sistema invocadas por um programa de nome **ls**. De forma a testar fez-se correr concorrentemente com o comando anterior o seguinte comando: **ls -l**. O output resultante pode agora ser consultado:

```
#Chamadas de sistema.

^C
Program: ls

|        Sys Call |    times Called | Time spent
|           rexit |               1 | 0 ns
|          getpid |               1 | 1735 ns
|       getrlimit |               1 | 2773 ns
|      systeminfo |               1 | 5900 ns
|            read |               1 | 20068 ns
|          munmap |               1 | 25508 ns
|           fcntl |               2 | 4276 ns
|       setcontext |               2 | 6650 ns
|           ioctl |               2 | 26120 ns
|        getdents |               2 | 41593 ns
|          doorfs |               4 | 103220 ns
|         mmapobj |               4 | 124016 ns
|          getuid |               5 | 9511 ns
|         memcntl |               6 | 50632 ns
|     resolvepath |               6 | 72009 ns
|           close |               7 | 25334 ns
|            mmap |               7 | 47817 ns
|             brk |              11 | 44880 ns
|          openat |              11 | 157458 ns
|        pathconf |              22 | 62058 ns
|           write |              22 | 162218 ns
|             acl |              44 | 217631 ns
|         fstatat |              53 | 236580 ns
```

Ao observar o output temos a prova da correção do programa sendo que apenas as chamadas de sistema invocadas no programa em questão se encontram detalhadas com o número de vezes que foram invocadas e o tempo total despendido para cada.

III. Conclusão e trabalho futuro

Dado por concluído o projeto pensa-se ter sido mostrado um bom conhecimento da ferramenta DTrace e do seu uso. Ao longo do desenvolvimento do projeto foi-se tornando cada vez mais fácil a programação dos programas, permitindo assim talvez melhores práticas de programação nos programas finais do que o que se observa nos programas iniciais. Agora, depois de concluídos todos os programas propostos, talvez fosse interessante para trabalho futuro, voltar a implementar cada um sem olhar para esta implementação aqui apresentada. Certamente que, agora com mais conhecimento no assunto, eles ficariam mais sintéticos e mais fáceis de entender por olhos estrangeiros ao seu processo de desenvolvimento.
Em síntese, concluí-se que o projeto foi um sucesso, visto que, como indicado na introdução, o seu objetivo era o de desenvolver competências com o uso da ferramenta DTrace e, como mencionado no parágrafo anterior, foi exatamente isso que aconteceu e encontra-se aqui provado com os programas desenvolvidos e explicados.

IV. Bibliografia

Referências

[1] http://www.oracle.com/technetwork/server-storage/solaris11/documentation/dtrace-cheatsheet-1930738.pdf
[2] http://dtrace.org/blogs/brendan/2011/11/09/solaris-11-dtrace-syscall-provider-changes/