Exploração da ferramenta DTrace

- TPC3 -

Carlos Sá - A59905
carlos.sa01@gmail.com

June 12, 2016

Abstract

Este relatório é o resultado de um estudo feito sobre a ferramenta DTrace. Esta ferramenta é uma ferramenta de análise disponível ao nível do sistema operativo que permite fazer o traçado dinâmico de programas e permite obter informações sobre o traçado de chamadas ao sistema que ocorrem ao nível do kernel tais como: informação sobre os processos e de hierarquia de chamadas ao sistema realizadas, valores de retorno de funções etc.

Para este estudo foram construídas scripts em linguagem d com objectivo de fazer o traçado das chamadas a sistema e retirar informações como: PID, valores de retorno, informações sobre os processos a correr no sistema, UID de utilizador, GID do grupo, entre outras. Todo o trabalho será realizado no sistema operativo Solaris 11 e o tratamento estatistico será feito com base nos resultados obtidos nesta plataforma.

O teste das scripts baseia-se na execução de 4 comandos cat diferentes e análise da captura feita com o comando dtrace.

*Numa fase final do trabalho adicionei aínda uma **componente extra** a este trabalho que corresponde à execução de uma script dtrace disponibilizada pelo professor António Pina (**threadsched.d**) para realizar o traçado dinâmico de um programa já com directivas **OpenMP** para vários tipos de escalonamento de carga.*

Contents

1. Introdução

O objectivo deste trabalho consiste em exercitar o comando **dtrace** e resolver uma série de exercícios propostos. Um primeiro exercício que consiste em criar um programa em dtrace e imprimir informação variada sobre o traçado realizado da chamada ao sistema *openat()* que em Solaris 11 é o *openat()* (mais genérico).

Da variada informação relevante a retirar pretende-se retirar informação sobre a identificação do processo (PID), utlizador (UID) e GID (grupo) valor de retorno, caminho para o ficheiro, entre outros. Depois de retirada essa informação pretende-se testar o programa com um conjunto de diferentes hipóteses com recurso ao comando **cat**. Num segundo exercicio pretende-se gerar um conjunto de estatisticas sobre a chamada ao sistema openat: número de tentativas de abrir ficheiros existentes, tentativas de criação de ficheiros, número de tentativas bem sucedidas e repetir a experiência com um dado período para imprimir hora e dia atual, e estatisticas recolhidas por PID com o seu respectivo nome.

2. O comando dtrace

O comando dtrace é um front-end genérico que implementa uma interface simples de chamada da linguagem **D**, capaz de fornecer informação sobre o traçado de aplicações utilizando o kernel e um conjunto de rotinas que permite realizar a formatação e impressão dos dados traçados nessas

mesmas aplicações. Como iremos perceber ao longo deste trabalho, este comando fornece um conjunto de serviços que permite ao programador obter informação variada sobre o programa: valores de retorno, caminhos para ficheiros abertos, tentativas de criação de ficheiros, PID, UID de utilizador etc. A linguagem **D** é a linguagem fundamental da utilização desta ferramenta e as informações obtidas sobre o programa "a traçar" são conseguidas com a utilização e interpretação de scripts em linguagem **D**.

3. Exercício 1 - Traçado da *syscall openat()*

Para este exercício pretendia-se fazer o traçado da *syscall* e imprimir informação variada por linha tal como:

- Nome do executável (execname);
- PID do processo;
- UID do utilizador;
- GID do grupo;
- A path para o ficheiro que for aberto;
- As flags da chamada da syscall openat();
- O respectivo valor de retorno;

Para a resolução deteste primeiro exercício foi necessário criar uma script em linguagem D. A script abaixo já contém o código completo do exercicio 1 (à excepção da alteração necessária para o exercicio opcional que requere a adição de um predicado como irei abordar mais a frente em 5). Abaixo será feita a análise da script por forma a tornar explicita a forma como a informação relativa aos 4 pontos é imprimida.

```
/** Flags for exercise 1 point 3 for Solaris 11
inline int O_WRONLY = 1;
inline int O_RDWR = 2;
inline int O_APPEND = 8;
inline int O_CREAT = 256;
**/

this string s_flags;

syscall::openat*:entry {
    self->path = copyinstr(arg1);
    self->flags = arg2;
}

syscall::openat*:return {

 this->s_flags = strjoin (
        self->flags & O_WRONLY ? "O_WRONLY"
        : self->flags & O_RDWR ? "O_RWR" : "O_RDONLY",
        strjoin ( self->flags & O_APPEND ? "|O_APPEND" : "",
                  self->flags & O_CREAT ? "|O_CREAT" : ""));
    printf("EXECNAME,PID,UID,GID,PATH,FLAGS,RETURN_VALUE\n");
    printf("%s,%d,%d,%d,%s,%s,%d\n",
        execname, pid, uid, gid, self->path, this->s_flags, arg1);
}
```

Para a resolução do 1º ponto do exercicio um, apenas utilizamos precisámos de nos focar no último **printf** dentro do **return**. Como podemos verificar as diferentes variáveis:

- execname (nome do executável);
- pid (correspondente process id do executável);
- uid (identificador de utilizador);
- gid (identificação do grupo);

Imprimem a informação pretendida para este 1º ponto deste exercicio. O caminho do absoluto para o ficheiro que for aberto que está em arg1 e é guardada no campo **path** da estrutura **self** e que também é imprimido. A impressão das flags do openat é feita capturando o valor de *arg*2 no probe entry e tratada no **return** para posterior impressão consoante o ficheiro tenha sido aberto para leitura, escrita etc. Para o tratamento das flags foi necessário utilizar uma variável **s_flags** e a função **strjoin** para fazer join das strings. Para imprimir o valor de retorno basta imprimir o valor da variável *arg1*.

4. Teste do programa com cat

Após a criação do programa, foi necessário fazer um conjunto de testes utilizando o comando cat. Notar que nestes foi utilizada a script acima (aínda sem a adição do predicado que restringiria o traçado à directoria /etc/ - como pedido no exercicio opcional). Todos os testes foram realizados na máquina com Solaris 11. Cada comando **cat** foi executado à vez, depois de correr o comando dtrace com o programa que criei: *syscall_open_ex_um.d*.

```
$ dtrace -qs syscall_open_ex_um.d >> cat_all.csv
```

Nas subsecções seguintes podemos visualizar 4 tabelas. Cada uma delas diz respeito ao resultado imprimido pelo comando dtrace na execução do programa após executar um comando cat. Para realizar estes testes, executo o comando dtrace num terminal e no outro terminal executo um comando cat. O resultado da execução desse comando dtrace é guardado em ficheiro **csv**. O mesmo é feito para os comandos cat restantes e os resultados são acrescentados ao ficheiro **csv**. No final esses dados são divididos e tratados dando origem aos resultados apresentados nas tabelas das subsecções seguintes. Algum output desnecessário foi também retirado para melhor visualização dos resultados para cada um dos testes.

4.1. Teste com **cat /etc/inittab > /tmp/test_a59905**

O primeiro teste foi realizado utilizando o comando:

```
$ cat /etc/inittab > /tmp/test_a59905
```

cat /etc/inittab > /tmp/test_a59905						
EXECNAME	**PID**	**UID**	**GID**	**PATH**	**FLAGS**	**RETURN_VALUE**
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
nfsmapid	1351	1	12	/system/volatile/rpc_door/rpc_100172.1	O_RDONLY	-1
bash	21856	29214	5000	/tmp/test_a59905	O_WRONLY|O_CREAT	4
cat	21856	29214	5000	/var/ld/ld.config	O_RDONLY	-1
cat	21856	29214	5000	/lib/libc.so.1	O_RDONLY	3
cat	21856	29214	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	21856	29214	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
cat	21856	29214	5000	/etc/inittab	O_RDONLY	3

Figure 1: *Resultados do dtrace com o programa* ***syscall_open_ex_um.d***

Note-se que para todos os testes foi verificado especificamente o **UID**, para verificar que os resultados dizem respeito a registos de acções realizadas na máquina provocadas pela minha conta de utilizador (29214). Uma forma de perceber qual era o meu **UID** é executar o comando **id** na bash.

4.2. Teste com **cat /etc/inittab » /tmp/test_a59905**

O segundo teste foi realizado utilizando o comando:

```
$ cat /etc/inittab >> /tmp/test_a59905
```

cat /etc/inittab >> /tmp/test_a59905						
EXECNAME	PID	UID	GID	PATH	FLAGS	RETURN_VALUE
bash	21891	29214	5000	/tmp/test_a59905	O_WRONLY|O_APPEND|O_CREAT	4
cat	21891	29214	5000	/var/ld/ld.config	O_RDONLY	-1
cat	21891	29214	5000	/lib/libc.so.1	O_RDONLY	3
cat	21891	29214	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	21891	29214	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
cat	21891	29214	5000	/etc/inittab	O_RDONLY	3

Figure 2: *Resultados do dtrace com o programa* ***syscall_open_ex_um.d***

4.3. Teste com **cat /etc/inittab | tee /tmp/test_a59905**

O terceiro teste foi realizado utilizando o comando:

```
$ cat /etc/inittab | tee /tmp/test_a59905
```

cat /etc/inittab | tee /tmp/test_a59905						
EXECNAME	PID	UID	GID	PATH	FLAGS	RETURN_VALUE
tee	21896	29214	5000	/var/ld/ld.config	O_RDONLY	-1
tee	21896	29214	5000	/lib/libc.so.1	O_RDONLY	3
tee	21896	29214	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
tee	21896	29214	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
tee	21896	29214	5000	/tmp/test_a59905	O_WRONLY|O_CREAT	3
cat	21895	29214	5000	/var/ld/ld.config	O_RDONLY	-1
cat	21895	29214	5000	/lib/libc.so.1	O_RDONLY	3
cat	21895	29214	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	21895	29214	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
cat	21895	29214	5000	/etc/inittab	O_RDONLY	3

Figure 3: *Resultados do dtrace com o programa* ***syscall_open_ex_um.d***

4.4. Teste com **cat /etc/inittab | tee -a /tmp/test_a59905**

O quarto teste foi realizado utilizando o comando:

```
$ cat /etc/inittab | tee -a /tmp/test_a59905
```

cat /etc/inittab | tee -a /tmp/test_a59905						
EXECNAME	PID	UID	GID	PATH	FLAGS	RETURN_VALUE
tee	21955	29214	5000	/var/ld/ld.config	O_RDONLY	-1
tee	21955	29214	5000	/lib/libc.so.1	O_RDONLY	3
tee	21955	29214	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
tee	21955	29214	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
tee	21955	29214	5000	/tmp/test_a59905	O_WRONLY|O_APPEND|O_CREAT	3
cat	21954	29214	5000	/var/ld/ld.config	O_RDONLY	-1
cat	21954	29214	5000	/lib/libc.so.1	O_RDONLY	3
cat	21954	29214	5000	/usr/lib/locale/en_US.UTF-8/en_US.UTF-8.so.3	O_RDONLY	3
cat	21954	29214	5000	/usr/lib/locale/en_US.UTF-8/methods_unicode.so.3	O_RDONLY	3
cat	21954	29214	5000	/etc/inittab	O_RDONLY	3

Figure 4: *Resultados do dtrace com o programa* ***syscall_open_ex_um.d***

5. Exercicio 1 (opcional) - Detetar apenas os ficheiros em /etc

Para a resolução do exercicio 1 opcional, criei uma cópia do programa em **d** utilizado anteriormente porque praticamente todo o código pode ser aproveitado. A ideia neste exercício opcional é de restringir o domínio de deteção por forma a que apenas os ficheiros na diretoria */etc* sejam detetados. Do resultado da pesquisa realizada percebi que poderia utilizar a função **strstr** para capturar todas as acções cujos **paths** que possuam "/etc". Então, para que seja possível fazer essa restrição basta adicionar um predicado:

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

No programa mostrado em 3 a seguir à parte correspondente ao **return** do *openat()*. Assim, para obter o resultado pretendido para este exercicio basta substituír o corpo de **syscall::openat*:return** pelo apresentado abaixo com predicado necessário à restrição do domínio de detecção:

```
syscall::openat*:return
/strstr(self->path,"/etc") != NULL/
{
    this->s_flags = strjoin (
        self->flags & O_WRONLY ? "O_WRONLY"
        : self->flags & O_RDWR ? "O_RWR" : "O_RDONLY",
        strjoin ( self->flags & O_APPEND ? "|O_APPEND" : "",
                  self->flags & O_CREAT ? "|O_CREAT" : ""));
    printf("EXECNAME,PID,UID,GID,PATH,FLAGS,RETURN_VALUE\n");
    printf("%s,%d,%d,%d,%s,%s,%d\n",
        execname, pid, uid, gid, self->path, this->s_flags, arg1);
}
```

Por forma a verificar o correto funcionamento do programa, igualmente corri o comando dtrace com o programa em **d** já com a adição do predicado. A execução do comando dtrace foi feita de forma análoga às anteriores.

Do lado direito encontram-se os resultados da execução do comando dtrace com os resultados obtidos e guardados no ficheiro **ex_opcional.csv**:

EXECNAME	PID	UID	GID	PATH	FLAGS	RETURN_VALUE
ls	22908	29214	5000	/etc/	O_RDONLY	3
nfsmapid	1351	1	12	/etc/resolv.conf	O_RDONLY	10
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
ls	22909	29214	5000	/etc/log	O_RDONLY	3
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
bash	22895	29214	5000	/etc/	O_RDONLY	4
nfsmapid	1351	1	12	/etc/resolv.conf	O_RDONLY	10
ls	22910	29214	5000	/etc/log/	O_RDONLY	3
bash	22895	29214	5000	/etc/	O_RDONLY	4
nfsmapid	1351	1	12	/etc/resolv.conf	O_RDONLY	10

Figure 5: *Resultados do dtrace com o programa* ***syscall_openat_ex_um_opcional.d***

Podemos verificar que aparentemente o "filtro" está a ser feito corretamente uma vez que só são capturados os ficheiros que pertencem à *path* "/etc". Claro que a tabela ao lado não exprime um resultado muito exaustivo. Apenas foi aqui representado na tabela output relevante à análise do problema. Por forma a capturar algum output relevante, realizei alguns comandos ls sobre a directoria /etc, /etc/log etc, e verifiquei que todos eles eram corretamente capturados pelo programa.

6. Exercicio 2

Nesta secção poderemos encontrar a resolução do exercício 2 completo. Tal significa que apenas um programa em **d** foi criado por forma a responder às duas alineas: **a**) e **b**).

O programa em d que a cada iteração imprime número de tentativas:

```
/* Counting attemps for open create and succeed openat */

syscall::openat*:entry
/(arg2 & O_CREAT) == O_CREAT/
{
    @create[execname, pid] = count();
}

syscall::openat*:entry
/(arg2 & O_CREAT) == 0/
{
    @open[execname, pid] = count();
}

syscall::openat*:return
/arg1 > 0/
{
    @success[execname, pid] = count();
}

tick-$1s {
    printf("%Y\n",walltimestamp);
    printf("%12s,%6s,%6s,%6s,%6s\n", "EXECNAME","PID","CREATE","OPEN","SUCCESS");
    printa("%12s,%6d,%@6d,%@6d,%@d\n", @create, @open, @success);
    trunc(@create);
    trunc(@open);
    trunc(@success);
}
```

Através da análise da script, podemos verificar que uma acção semelhante é realizada sempre que o predicado entre "/" é verificado. Os diferentes probes **entry** cobrem os casos em que se verifica uma tentativa de criação de um ficheiro, abertura de ficheiros existentes e o mesmo para as tentativas bem sucedidas.

A contagem é feita com o recurso à função **count()**, e o resultado é imprimido no final dentro do corpo tick-$1s, como pretendido para a alinea **a)**.

Para a resolução da alinea **b)**, como o objectivo passa por imprimir iterações dado um determinado periodo, foi necessário incluir as impressões dos valores dentro do corpo tick-$1s.

Desta forma consigo fazer com que os resultados de cada iteração sejam imprimidos de acordo com um determinado período. Para tal basta que se execute esta script utilizando o comando dtrace da seguinte forma:

```
$ dtrace -qs syscall_openat_ex_dois.d 4
```

- abrir ficheiros existentes;
- criar ficheiros;
- bem-sucedidas.

Encontra-se abaixo e para o qual passarei a analisar os resultados obtidos.

O último parâmetro passado, corresponde ao número de segundos que queremos entre cada registo. De acordo com o comando acima, significa que queremos imprimir os resultados de cada iteração a cada 4 segundos. Um dos objectivos para a alínea **b)** do exercício 2 passava por, em cada linha, imprimir a data e a hora num formato legível. Assim, depois de pesquisar percebi que existem diferentes formas de o fazer. Existem as funções **gettimeofday()**, **ctime()** e a variável **walltimestamp**. Dos três, escolhi o **walltimestamp** como se verifica no primeiro printf realizado. Por último são também recolhidos o **execname** e o **PID** de cada registo e igualmente imprimido no final. Este trabalho é feito sempre que uma cada um dos predicados é verificado e executada a respectiva a acção dentro do corpo de **entry** e **return**. A acção a realizar tem subjacente a utilização de **agregação** em linguagem **d** com uso de arrays.

A execução do programa faz com que seja imprimido o seguinte output com o resultado pretendido:

2016 Apr 10 14:40:01				
EXECNAME	PID	CREATE	OPEN	SUCCESS
nfsmapid	1351	0	2	0
dtrace	23189	0	2	2
utmpd	259	1	5	6
2016 Apr 10 14:40:05				
EXECNAME	PID	CREATE	OPEN	SUCCESS
nfsmapid	1351	0	2	0
2016 Apr 10 14:40:09				
EXECNAME	PID	CREATE	OPEN	SUCCESS
nfsmapid	1351	0	2	0
2016 Apr 10 14:40:13				
EXECNAME	PID	CREATE	OPEN	SUCCESS
2016 Apr 10 14:40:17				
EXECNAME	PID	CREATE	OPEN	SUCCESS
nfsmapid	1351	0	2	0
2016 Apr 10 14:40:21				
EXECNAME	PID	CREATE	OPEN	SUCCESS
nfsmapid	1351	0	1	1

Figure 6: *Resultados do dtrace com o programa* ***syscall_openat_ex_dois.d***

7. Estudo aplicação ex2_v2.cxx com script threadsched.d

Como parte extra deste trabalho realizarei alguns testes de um programa já com diretivas **OpenMP**. O algoritmo do programa em si resume-se ao código abaixo.

```
std::random_device d;
std::default_random_engine e1(d());

// a distribution that takes randomness and ↩
   produces values in specified range
std::uniform_int_distribution<> dist(1,nr);

 omp_set_num_threads(np);
 T1 = omp_get_wtime();
 #pragma omp parallel for private ( chedule (↩
    runtime)
 for (i=0 ; i < S ; i++) {
    a[i] = 0.;
        for (r = dist(e1) ; r > 0 ; r -= 20) {
            a[i] += r;
        }
 }
T2 = omp_get_wtime();
cout << "fiosExecucao =" << np <
Intervalo=" << nr <<  ": tempo ->
(T2-T1)*1e6 << " usecs\n";
```

Sendo que este código será compilado com o intuito de ser corrido numa máquina *multicore*, interessa estudar os diferentes tipos de escalonamento de carga **OpenMP**. Cada tipo de escalonamento divide o trabalho a realizar pelos cores de forma diferente. Com isso consegue-se perceber qual o escalonamento que confere uma melhor performance ao programa.

Para as execuções relativas aos diferentes tipos de escalonamento será feita uma análise do traçado com uma script dtrace já fornecida (**threadsched.d**)

Sendo que realizarei um estudo de um programa que corre em paralelo na máquina com **Solaris 11** habitual, é pertinente saber o número de cores fisicos e virtuais do processador que esta máquina possui. Para tal recorri ao comando **psrinfo** abaixo:

```
$ psrinfo -pv
```

O comando retorna como resultado a seguinte informação:

```
The physical processor has 8 cores and 16 virtual processors (0-7,16-23)
  The core has 2 virtual processors (0,16)
  The core has 2 virtual processors (1,17)
  The core has 2 virtual processors (2,18)
  The core has 2 virtual processors (3,19)
  The core has 2 virtual processors (4,20)
  The core has 2 virtual processors (5,21)
  The core has 2 virtual processors (6,22)
  The core has 2 virtual processors (7,23)
    x86 (GenuineIntel 306E4 family 6 model 62 step 4 clock 2600 MHz)
      Intel(r) Xeon(r) CPU E5-2650 v2 @ 2.60 GHz

The physical processor has 8 cores and 16 virtual processors (8-15,24-31)
  The core has 2 virtual processors (8,24)
  The core has 2 virtual processors (9,25)
  The core has 2 virtual processors (10,26)
  The core has 2 virtual processors (11,27)
  The core has 2 virtual processors (12,28)
  The core has 2 virtual processors (13,29)
  The core has 2 virtual processors (14,30)
  The core has 2 virtual processors (15,31)
    x86 (GenuineIntel 306E4 family 6 model 62 step 4 clock 2600 MHz)
      Intel(r) Xeon(r) CPU E5-2650 v2 @ 2.60 GHz
```

Assim, a nível de recursos de processamento vamos executar o programa **ex2_v2** numa máquina com 2 processadores **Intel(r) Xeon(r) CPU E5-2650 v2 @ 2.60GHz** cada um deles contendo 8 cores fisicos e um total de 16 cores virtuais. No total, temos então uma máquina com um total de **32 cores**.

7.1. Scheduling OpenMP

Pelo código apresentado no inicio da secção 7, pudemos constatar a existência de uma directiva **OpenMP** imediatamente antes ao inicio do ciclo for. Esta primitiva identifica que o trabalho realizado pelo ciclo ***for*** deve ser feito em paralelo. Tal significa que o trabalho dentro deste ciclo (i.e as iterações do ciclo) devem computadas em paralelo utilizando os cores da máquina. O trabalho a realizar limita-se ao cálculo de um valor que é armazenado na variável **r** que depois é somado na posição **i** do array **a**. Como não existem dependências de dados, esta operação pode ser realizada em paralelo. Existe um total de **S** iterações do ciclo exterior que podem ser divididas e pelas threads.

Quando utilizamos **OpenMP** podemos utilizar diferentes politicas de escalonamento de carga. Podemos dividir uniformemente o número total de iterações do ciclo pelo número de threads existentes e cada thread fica responsável por computar esse número de iterações. Ao número de iterações ou porção de trabalho que cada thread realiza chama-se de **chunksize**.

Claro que dependendo da política de escalonamento de carga escolhida, a computação é influenciada a nível de desempenho. Assim sendo, um estudo pertinente é avalizar a mesma implementação **OpenMP** para os diferentes tipos de escalonamento de carga e perceber o melhor escalonamento de carga para este algoritmo.

Os diferentes escalonamentos de carga do **OpenMP** são:

- **static**: Neste tipo de escalonamento o **chunksize** é obtido dividindo o número iterações do ciclo a realizar em paralelo pelo número de threads disponíveis. Assim sendo o ciclo é dividido em chunks de tamanho aproximadamente iguais. Definindo o tamanho de chunk para 1, cada thread executa uma interação do ciclo de forma alternada.

- **dynamic**: Neste escalonamento as iterações são divididas em blocos e cada thread executa um bloco de iterações de tamanho fixo. Os blocos são organizados numa queue. Quando uma thread termina a execução de um bloco de iterações pega noutro bloco de iterações da queue. Quando se define este tipo de escalonamento, é comum passar como parâmetro o chunksize que corresponde ao tamanho de um bloco de iterações. Contudo, por uma questão de simplificação de estudo e posterior análise, não irei passar como argumento nenhum **chunksize** pelo que será utilizado o chunksize por omissão que é de blocos de 1 iteração. De todos os escalonamentos, este é aquele que introduz maior overhead devido ao tempo que é necessário para fazer a divisão das iterações em blocos e organizálos na queue.

- **guided**: Este escalonamento surge numa tentativa de em certos casos equilibrar melhor a carga de iterações pelas threads. É bastante semelhante ao escalonamento dinámico mas o tamanho dos chunks atribuidos a cada thread varia em runtime consoante o comportamento das threads. Isto faz com que exista um maior equilibrio da carga de trabalho que cada thread realiza. E, simultaneamente diminui-se a assimetria entre threads que realizam trabalho mais rápido do que outras que demoram mais tempo. Neste escalonamento é também possível passar como parâmetro um valor que especifica o tamanho minimo do chunk.

Existe aínda o escalonamento automático (**auto**) que não será utilizado neste trabalho.

7.2. Execução script dtrace com ex2_v2 para diferentes tipos escalonamento

Assim executei o programa ex2_v2 em conjunto com a script **threadsched.d** para os diferentes tipos de escalonamento do **OpenMP**.

A script dtrace é executada em conjunto com o programa **ex2_v2** de acordo com o comando seguinte:

```
1 $ dtrace -s threadsched.d -c "./ex2_v2 $nthreads" >> "DTRACE_"$schedule"_"$nthreads"_"$seq".csv"
```

Note-se que o comando acima se trata de um comando exemplificativo uma vez que para realizar todos os testes foi criada uma script para realizar os testes para todos os tipos de escalonamento para um número variável de threads, recolhendo vários samples e escolhido o melhor tempo de execução. Os resultados são guardados em ficheiros **csv** para posterior tratamento. Como referido no inicio da secção 7, a máquina Solaris utilizada para este estudo possui um total de **32 cores**. Assim, farei o estudo do escalonamento para um número variável de threads desde execução sequencial (1 thread) até 64 threads que é a potência de 2 imediatamente a seguir a 2^5 que corresponde ao número total de threads da máquina, apenas por curiosidade e para investigar o impacto na performance que decorre do facto de termos o programa para mais threads do que as disponíveis pelo hardware.

7.3. Resultados do Scheduling OpenMP

Depois de executados todos os tipos de escalonamento para um número variável de threads cheguei aos resultados ilustrados pelo gráfico da figura 7:

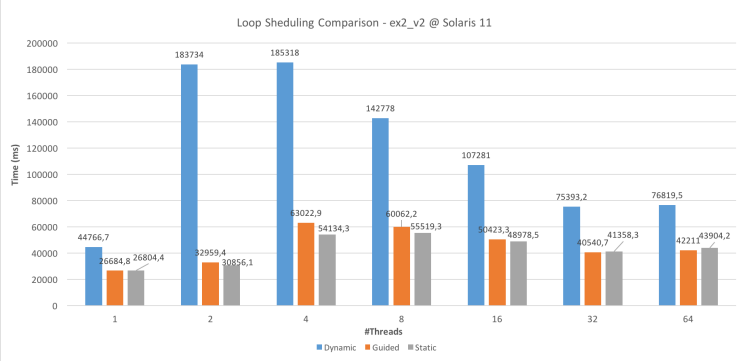

Figure 7: *Resultados obtidos para diferentes tipos de escalonamento de carga **OpenMP***

De todos os escalonamentos existentes, o **dynamic** revela ser o escalonamento que confere pior performance ao programa e deve ser excluído logo à partida. A dimensão do problema e o tipo de algoritmo em análise faz com que o custo do overhead associado a este escalonamento seja demasiado elevado em termos de performance. Aliás, no geral este algoritmo é péssimo para o paralelismo uma vez que a execução sequencial (1 thread) foi aquela que registou um menor tempo de execução, em comparação com os tempos obtidos para os diferentes números de threads utilizadas independentemente do escalonamento em causa.

Dentro do escalonamento **static** e **guided** os resultados são um pouco inconclusivos e não existe uma grande evidência de que um escalonamento seja melhor do que o outro a nível de performance. Quanto muito poderemos dizer que para um número menor de threads o melhor escalonamento é o **static** e para um número de threads superior o **guided** será melhor. Contudo, estas diferenças não são muito evidentes. Fiquei um pouco surpreendido com os resultados obtidos para 64 threads. Quando corremos o programa para 64 threads, estamos a utilizar o dobro das threads da máquina Solaris. Estava já a contar que de facto os tempos fossem piores mas estava a contar que assimetria dos resultados para os três escalonamentos fossem maiores do que as registadas.

Note-se que ao longo deste trabalho foram feitas **10 execuções** para cada número de threads utilizadas em cada tipo de escalonamento. Nos gráfico da figura 7 os tempos exibidos correspondem à seleção do tempo de execução mais curto.

7.4. Análise resultados com script Dtrace **threadsched.d**

Depois de analisar os resultados obtidos pelos diferentes tipos de escalonamento em termos de desempenho e fazer a comparação dos tempos de execução, analisei o resultado do output com a script dtrace.

Como já tinha verificado anteriormente, a máquina solaris é uma máquina dotada com 2 processadores com um total de **32 cores**. Assim sendo, analisei o resultado da script dtrace para a execução de **32 threads**. Depois de analisar os resultados da script dtrace correspondentes aos três tipos de escalonamento verfiquei que não existem diferenças significativas. Isto é, não existe uma grande evidência que exista um escalonamento **OpenMP** que se destaque por ter mais migrações de trabalho de umas threads para as outras, mais sleepings de threads numa dada *condition variable*, número de mudanças de contexto etc.

Assim, utilizarei apenas o exemplo do output de um dos escalonamentos para analisar os resultados obtidos com a script dtrace fornecida.

```
 9:1465569461579 TID  15  from-CPU 17(1) to-cpu 0(1) CPU migration                1
 9:1465569461579 TID  15 CPU   0(1) restarted on the same CPU                      2
79:1465569461649 TID  15 sleeping on  cond var                                    3
79:1465569461649 TID  15 CPU   0(1) preempted                                     4
79:1465569461649 TID  15 CPU   0(1) restarted on the same CPU                      5
79:1465569461649 TID  15 sleeping on  cond var                                    6
79:1465569461649 TID  15 CPU   0(1) preempted                                     7
79:1465569461649 TID  13  from-CPU 16(1) to-cpu 0(1) CPU migration                8
79:1465569461649 TID  13 CPU   0(1) restarted on the same CPU                      9
 9:0          TID  27 CPU   1(1) created                                          10
 9:1465569461579 TID  27 CPU   1(1) restarted on the same CPU                     11
 9:1465569461579 TID  27 sleeping on  cond var                                    12
 9:1465569461579 TID  27 CPU   1(1) preempted                                     13
 9:0          TID  29 CPU   1(1) created                                          14
 9:1465569461579 TID  29 CPU   1(1) restarted on the same CPU                     15
 9:1465569461579 TID  29 sleeping on  cond var                                    16
 9:1465569461579 TID  29 CPU   1(1) preempted                                     17
 9:1465569461579 TID  25  from-CPU 17(1) to-cpu 1(1) CPU migration                18
 9:1465569461579 TID  25 CPU   1(1) restarted on the same CPU                     19
 9:1465569461579 TID  25 sleeping on  cond var                                    20
 9:1465569461579 TID  25 CPU   1(1) preempted                                     21
 9:0          TID  31 CPU   1(1) created                                          22
 9:1465569461579 TID  31 CPU   1(1) restarted on the same CPU                     23
 9:1465569461579 TID  31 sleeping on  cond var                                    24
 9:1465569461579 TID  31 CPU   1(1) preempted                                     25
 9:1465569461579 TID   7  from-CPU 17(1) to-cpu 1(1) CPU migration                26
 9:1465569461579 TID   7 CPU   1(1) restarted on the same CPU                     27
29:1465569461599 TID   7 CPU   1(1) preempted                                     28
29:1465569461599 TID   7 CPU   1(1) restarted on the same CPU                     29
49:1465569461619 TID   7 CPU   1(1) preempted                                     30
49:1465569461619 TID   7 CPU   1(1) restarted on the same CPU                     31
79:1465569461649 TID   7 sleeping on  cond var                                   32
79:1465569461649 TID   7 CPU   1(1) preempted                                     33
79:1465569461649 TID   7 CPU   1(1) restarted on the same CPU                     34
79:1465569461649 TID   7 sleeping on  cond var                                   35
79:1465569461649 TID   7 CPU   1(1) preempted                                     36
79:1465569461649 TID   7 CPU   1(1) restarted on the same CPU                     37
89:1465569461659 TID   1  from-CPU 19(1) to-cpu 1(1) CPU migration               38
89:1465569461659 TID   1 CPU   1(1) restarted on the same CPU                     39
89:1465569461659 TID   1 sleeping on  cond var                                   40
89:1465569461659 TID   1 CPU   1(1) preempted                                     41
89:1465569461659 TID   1 CPU   1(1) restarted on the same CPU                     42
89:1465569461659 TID   1 sleeping on  cond var                                   43
89:1465569461659 TID   1 CPU   1(1) preempted                                     44
89:1465569461659 TID   1 CPU   1(1) restarted on the same CPU                     45
89:1465569461659 TID   1 sleeping on  cond var                                   46
89:1465569461659 TID   1 CPU   1(1) preempted                                     47
89:1465569461659 TID   1 CPU   1(1) restarted on the same CPU                     48
89:1465569461659 TID   1 sleeping on  cond var                                   49
89:1465569461659 TID   1 CPU   1(1) preempted                                     50
89:1465569461659 TID   1 CPU   1(1) restarted on the same CPU                     51
89:1465569461659 TID   1 sleeping on  cond var                                   52
89:1465569461659 TID   1 CPU   1(1) preempted                                     53
89:1465569461659 TID   1 CPU   1(1) restarted on the same CPU                     54
89:1465569461659 TID   1 sleeping on  cond var                                   55
89:1465569461659 TID   1 CPU   1(1) preempted                                     56
 9:1465569461579 TID   1  from-CPU 18(1) to-cpu 2(1) CPU migration                57
 9:1465569461579 TID   1 CPU   2(1) restarted on the same CPU                     58
 9:1465569461579 TID   1 sleeping on  kernel RW lock                              59
 9:1465569461579 TID   1 CPU   2(1) preempted                                     60
 9:1465569461579 TID   1 CPU   2(1) restarted on the same CPU                     61
 9:1465569461579 TID   1 sleeping on  kernel RW lock                              62
 9:1465569461579 TID   1 CPU   2(1) preempted                                     63
 9:1465569461579 TID   1 CPU   2(1) restarted on the same CPU                     64
 9:1465569461579 TID   1 sleeping on  cond var                                   65
 9:1465569461579 TID   1 CPU   2(1) preempted                                     66
 9:1465569461579 TID  25  from-CPU 1(1) to-cpu 2(1) CPU migration                 67
 9:1465569461579 TID  25 CPU   2(1) restarted on the same CPU                     68
39:1465569461609 TID  25 CPU   2(1) preempted                                     69
39:1465569461609 TID  25 CPU   2(1) restarted on the same CPU                     70
79:1465569461649 TID  25 sleeping on  cond var                                   71
79:1465569461649 TID  25 CPU   2(1) preempted                                     72
79:1465569461649 TID  29  from-CPU 3(1) to-cpu 2(1) CPU migration                73
79:1465569461649 TID  29 CPU   2(1) restarted on the same CPU                     74
79:1465569461649 TID  29 sleeping on  cond var                                   75
79:1465569461649 TID  29 CPU   2(1) preempted                                     76
79:1465569461649 TID  25 CPU   2(1) restarted on the same CPU                     77
79:1465569461649 TID  25 sleeping on  cond var                                   78
79:1465569461649 TID  25 CPU   2(1) preempted                                     79
79:1465569461649 TID  29 CPU   2(1) restarted on the same CPU                     80
79:1465569461649 TID  29 sleeping on  cond var                                   81
79:1465569461649 TID  29 CPU   2(1) preempted                                     82
79:1465569461649 TID  25 CPU   2(1) restarted on the same CPU                     83
79:1465569461649 TID  25 sleeping on  cond var                                   84
79:1465569461649 TID  25 CPU   2(1) preempted                                     85
79:1465569461649 TID  25 CPU   2(1) restarted on the same CPU                     86
 9:1465569461579 TID  29  from-CPU 1(1) to-cpu 3(1) CPU migration                 87
 9:1465569461579 TID  29 CPU   3(1) restarted on the same CPU                     88
```

```
79:1465569461649 TID  29 sleeping on  cond var
79:1465569461649 TID  29 CPU   3(1) preempted
79:1465569461649 TID  19  from-CPU 19(1) to-cpu 3(1) CPU migration
79:1465569461649 TID  19 CPU   3(1) restarted on the same CPU
79:1465569461649 TID  19 sleeping on  cond var
79:1465569461649 TID  19 CPU   3(1) preempted
79:1465569461649 TID  19 CPU   3(1) restarted on the same CPU
79:1465569461649 TID  19 sleeping on  cond var
79:1465569461649 TID  19 CPU   3(1) preempted
89:1465569461659 TID  15  from-CPU 0(1) to-cpu 3(1) CPU migration
89:1465569461659 TID  15 CPU   3(1) restarted on the same CPU
 9:1465569461579 TID  28  from-CPU 15(2) to-cpu 4(1) CPU migration
 9:1465569461579 TID  28 CPU   4(1) restarted on the same CPU
79:1465569461649 TID  28 sleeping on  cond var
79:1465569461649 TID  28 CPU   4(1) preempted
79:1465569461649 TID  31  from-CPU 12(1) to-cpu 4(1) CPU migration
79:1465569461649 TID  31 CPU   4(1) restarted on the same CPU
79:1465569461649 TID  31 sleeping on  cond var
79:1465569461649 TID  31 CPU   4(1) preempted
89:1465569461659 TID  31 CPU   4(1) restarted on the same CPU
 9:0         TID   5 CPU   5(1) created
```

O output obtido permite-nos concluir que existe uma série de eventos distintos do escalonador capazes de serem detetados com a script dtrace. Este excerto corresponde à melhor execução com escalonamento **dinâmico** para **32 threads**. Com a script **threadsched.d** podemos detetar quando uma determinada thread é criada (created) sendo que à frente do CPU aparece a identificação da thread. Conseguimos detetar migrações de trabalho de uma thread de um CPU para o outro (CPU migration) e saber exatamente de que CPU e para que CPU existe migração de trabalho. Facilmente conseguimos identificar pontos em que uma determinada thread é parada (i.e colocada a "dormir" - sleeping) numa determinada variável de condição. A thread entra em modo *sleep* até que a variável de condição seja alterada. Isto pode ocorrer se existir uma segunda thread a aceder a uma zona critica do código. Alguns destes eventos introduzem algum **overhead**, como as mudanças de contexto, criação de threads e locks. Contudo, este overhead não é demasiado custoso em termos de performance, o overhead mais significativo é o overhead associado à utilização da API **OpenMP**.

8. Conclusão

Graças à realização do presente trabalho fui capaz de desenvolver algumas competências em linguagem **d** na criação de scripts utilizáveis para realizar traçados utilizando a ferramenta dtrace. Todos os exercícios do TPC foram realizados e testados. Além do traçado por si, este trabalho teve especial importância para explorar os conceitos de **probes**, e **agregação**.

Neste trabalho fez-se um traçado da chamada ao sistema *openat()* mas o trabalho realizado sobre o traçado desta syscall poderia ser facilmente reprodutível para outras.

A nível de trabalho futuro poderia ser explorado o traçado de outras *syscall* e realizada a mesma análise de resultados.

Numa fase final do trabalho aínda adicionei um trabalho extra na secção 7 onde explorei uma aplicação com diretivas **OpenMP** e estudei a performance do programa para três tipos de escalonamento de carga distintos: dinâmico, guided e estático.

Com uma script **dtrace** estudei o traçado dinâmico do mesmo programa com alguns eventos registados do escalonador que envolve criação de threads, migrações e mudanças de contexto, desafetação forçada de threads entre outros.

[1]

References

[1] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2013.