DTrace Lab exercises

DTraceis a comprehensive dynamic tracing framework for the Solaris Operating Environment. It provides a powerful infrastructure to permit administrators, developers, and service personnel to concisely answer arbitrary questions about the behavior of the operating system and user programs. Copyright 2007 Sun Microsystems, Inc.

Prerequisites

- Basic knowledge on the Solaris Operating Environment
- A basic level of understanding of DTrace and understanding of the structure of the D-language would be very useful but not required.

System requirement

- Supported OS: Solaris 10 or newer on X86
- Memory requirement: 1GB recommended
- Disk space requirement: 150M bytes

Basic concepts

- Exercise 1: dtrace command line, listing and turning on probes
- Exercise 2: D Language Actions and Predicates
- Exercise 3: Using Aggregates to collect and process info

Resources

- DTraceToolkit
- Solaris Dynamic Tracing Guide

Exercise 1: dtrace command line example.

Introduction:

The goal of this exercise is to understand some basic concept and turn on and off probes from the command line.

Background Information:

- Solaris 10 introduced a new dynamic tracing framework called DTrace.
- DTrace allows you to dynamically instrument a live running system to collect any arbitrary information from any arbitrary location in the kernel and applications running in a Solaris 10 system.
- DTrace defines trace points called probes which could be turned on and off dynamically.
- There are close to 50000 probes defined in a Solaris 10 system.
- We will explore these probes in this exercise.

Steps to follow:

1. Login
2. Open a terminal window by right clicking any point in the background of the desktop, and select Open Terminal in the pop-up menu.
3. On the command line just type ' **dtrace**'. This should print the different options to the dtrace command.

```
# dtrace
```

4. Now let's list the probes that are available on your system.

```
# dtrace -l
```

You would see the listing of probes. There are 5 columns

- ID - Internal ID of the probe listed.
- Provider - Name of the Provider. Providers are used to classify the probes. This is also the method of instrumentation.
- Module - The name of the Unix module or application library of the probe
- Function - The name of the function in which the probe exists.
- Name - The name of the probe.

Background: What is a probe?
Lets take a step back here and explain the concept of a probe. In DTrace a probe is a point of instrumentation. In other words they are locations in the system that we can be 'probed' to get more details. In Solaris 10 there are many many such locations. The probes can be turned on or off. When the probe is turned off there is no overhead. We can turn these probes on dynamically. And when that location of the system is exercised the probe fires. We can collect interesting information from that probe point when these events happen.

Background: How are probes represented?
The probe is defined using 4 attributes, ***provider, module, function and name.*** *The* ***provider*** *defines a categorization of probes. They also represent a methodology of instrumentation. The* ***module*** *usually represents the name of the kernel module or user land library that has the probe. The* ***function*** *has the name of the function that is being instrumented. The* ***name*** *is the name of the probe. Some providers use the* ***name*** *to define is the point of instrumentation is in the beginning (entry) or end (return) of the function.*

5. To find the number of probes in your system, pipe the above command to **wc**.

```
# dtrace -l | wc -l
  67012
```

The number may vary based on your system type. This is the number of probes that your system is currently aware of.

6. Probes can be listed based on the provider or module or function or name. You can specify the following options with **-l** option.

- **-P** for provider
- **-m** for module
- **-f** for function
- **-n** for name

For example to list the probes in the sysinfo provider type the following command.

```
# dtrace -l -P sysinfo | more
```

To list probes in the ufs module, type the following command.

```
# dtrace -l -m ufs | more
```

To list the probe that have the name **open** type the following

```
# dtrace -l -f open
```

To list the probes with the name **start** type the following

```
# dtrace -l -n start
```

7. So far we just listed the probes we will now turn on and off probes on a live running system.

We can turn on probes using the **-P, -m, -f, -n**. When dtrace probes are turned on, they print out the occurrence of the probe event when they happen.

For example if you need to look at all the **virtual memory** activity in your system you can turn on all vminf

```
# dtrace -P vminfo
```

8. Press **q** to quit from the above run.

Exercise 2: D Language - Actions and Predicates
Introduction:

The goal of this exercise is to build a few D script to enable dynamic instrumentation in the Solaris system.

Background Information:

First the structure of a D script

```
probe-description
/predicate/
{
    action statements
}
```

probe-description *- probe(s) that you want to instrument*
predicate *- filter to limit the execution of action statements*
action *- the statements that collect system state.*

Next we will look at the probe-description. It contains 4 attributes
provider : module : function : name

provider *- name of the provider ex., pid, syscall, io etc...*
module *- name of kernel module or lib name ex. ufs, libc, a.out*
function *- name of the function ex. open, close, strcmp*
name *- name of the probe. ex. entry, return, start*

Next let's see the construct of the probe-description

- *Any one of the attribute can be left out and it will match all possible probes.*
- *Wild card like * and ? can be used in any attribute.*
- *Example*
    - *syscall:::entry - probe describe entry into all syscall*
    - *syscall::open*:entry - probe matches open and open64 syscall*
- *The action section allows us to report or collect some state of the systems. There is some generic info that we can collect or report in the action section. Some probes also provide some specific information for us to collect*
- *The predicate section allows us to limit when action executes.We can limit based on many criteria we will see a few example in this exercise.*
- *In this exercise we will use two action statements printf and trace. printf has the same format as the printf in the C language.*

Steps to follow:

1. Open a terminal window
   - Click on the menu mouse button with the cursor on the desktop background and choose "Open Terminal"
2. Using vi or your favorite editor create a file called syscalls.d

3. We will write a simple D-script to instrument the system call subsytem to print the name of the system calls that are happening in the system. We will also print the name and processid of the application that is making the system call.
   - Use **syscall:::entry** as the probe-description
   - Leave the predicate part empty
   - In the action section add printf statement to print the builtin variables **probefunc**, **pid** and **execname**

Here is how your syscall.d should look like

```
syscall:::entry
{
    printf("%s(%d) called %s\n", execname, pid, probefunc);
}
```

4. Run the program that you just wrote.
   dtrace -qs syscall.d | more
5. Output should look like.

Press **q** to end the script.

6. Now lets modify the code to only print the system calls made by 'java'
   - Copy syscall.d to java_syscall.d
   - Add a predicate to limit for execname == "java"
   - Modify the printf statement to print the process id (pid)

Here is how **java_syscall.d** should look like

```
syscall:::entry
/execname == "java"/
{
      printf("Java(%d) called syscall %s\n", pid, probefunc);
}
```

7. Run the program that you just wrote. You should see the following.

Press **q** to end the script.

Note: If you do not see any output you probably are not running any java process on the system. You can start one by typing java -version on another window.

Optional:

8. Modify syscall.d to only look for write system calls and print the name of the process that makes the write and the amount of bytes it writes.

Hint: probe description. syscall::write:entry No predicates needed for this example "number of bytes" is in arg2. If you need more details on write system call enter **man -s 2 write**

9. Create a D-script that prints out the name and the process id of any process that starts in the system.

Hint: probe description: proc:::exec-success

Exercise 3: Using Aggregates to collect and process info

Introduction:

We just wrote a few examples to illustrate the use of actions and predicates. Now we will see the use of the dtrace aggregate feature to collect and process information.

Background Information:

In the examples above you may have noticed that dtrace commands and scripts can produce volumes of information that needs to be processed. Collecting the information and then processing it in two steps normally uses up more system resources than what is needed.

*dtrace has a first class data structure called **aggregate** that helps you in collecting and processing the data in one step thus dramatically reducing collection cost.*

The syntax of aggregate

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

     @          symbol signifies the aggregate
     name:      optional name of  the aggregate
     key:       optionalindex of the aggregate
     aggfunc  one of the following
          count():       count of the event
          sum(exp):      running total of expression
          avg(exp):      average of expression
          min(exp):      min of the given expression
          max(exp):      max of the expression
          quantize(exp)a graph of distribution of expression
```

In this exercise we will use some of the aggregate functions to collect and process information.

Steps to follow:

1. Open a terminal window

- Click on the menu mouse button with the cursor on the desktop background and choose "Open Terminal"

2. Using vi or your favorite editor create a file called count_signals.d

3. For this exercise we will use the **sysinfo:::pswitch** probe. This probe fires when a the CPU switches from one process to another. We will use this script to count the number of chances a process got to execute.

- Use **sysinfo:::pswitch** as the probe-description
- Leave the predicate part empty
- In the action section add an aggregate to count the number of signals that are sent using the aggregate statement **@[execname]=count()**

Here is how your count_signals.d should look like.

```
sysinfo:::pswitch
{
   @[execname]=count();
}
```

4. Run the script that you just wrote

# dtrace -qs count_switch.d

<press **Ctrl c** after a few seconds>

Note: The dtrace script collects data and wait for you to stop the collection using Cntl C

Also you may notice that you do not explicitly print out the aggregation that you collected. DTrace will automatically print it for you. Again the idea here is that if you collected data you obviously want to print it so DTrace does this for you as an ease of use feature.

5. We will now look at the IO subsystem using DTrace. We will write a script to see the total io and the distribution of io sizes in the system.

- The **io:::start probe** fires when any io activity happens on the system.
- **args[0]->b_bcount** contain total bytes that are read/written for the io activity
- We will use aggfunc **sum(expr)** to keep track of the total io. Use the following aggregation

  @total = sum(args[0]->b_bcount);

- We will use the aggfunc **quantize(expr)** to track the distribution.

  @dist = quantize(args[0]->b_bcount);

- We will print the aggregates explicitly using the printa() command. This has similar syntax as the printf but takes an aggregate as an argument.
- Create a D-script and call it io.d. It should look like

```
io:::start
{
     @total = sum(args[0]->b_bcount);
     @dist = quantize(args[0]->b_bcount);
}    END   {          printa("Total IO completed %@d\n",@total);
                 printf("The IO distribution \n");
                 printa(@dist);
                }
```

6. Now run your code. On another window " **find /**". Press ^c after a 20 secs or so.

Note: The output above may be different from what you get. Here isa quick explanation of the output. This shows the distribution of the size of IO that is done in the system. The output shows that you had 2620 IOs of size between 1K and 2K and so on.

Optional Lab:

11. Calculate the amount of time spent in each system call in the system
- Create a file and call it syscall_time.d
- We will use 2 probe descriptions
  - first probe syscall:::entry
  - second probe syscall:::return
- In the action section of the first probe save timestamp in variable **ts**
  - timestamp is a dtrace builtin that has the number of nanosecond from a point in the past.
- In the action section of the second probe calculate nanoseconds that have passed using the following aggregation

```
@[probefunc]=sum(timestamp - ts)
```