Learning DTrace

Part 1: Introduction

Chip Bennett
편역: vangelis(securityproof@gmail.com)

Solaris 10의 가장 혁신적인 기능들 중의 하나는 시스템에는 거의 아무런 영향도 주지 않으면서 운영 중인 시스템의 성능 문제와 다양한 문제들을 동적으로 추적할 수 있도록 해주는 기능이다. 이것은 **DTrace**라고 부르는 것인데, 이것은 Dynamic Tracing를 의미한다. 이 글은 DTrace가 어떻게 작동하는지 여러분이 이해하는데 도움을 주고, 여러분 자신의 시스템 문제들을 해결하기 위해 DTrace를 사용하는데 필요한 기본적인 지식을 제공하기 위해 만들어진 일련의 문서들 중의 첫 번째 것이다.

만약 C와 awk 프로그래밍에 익숙하다면 여러분들은 DTrace를 실행하기 위해 필요한 것의 반은 이미 아는 것이다. DTrace는 D라는 스크립팅 언어를 사용하는데, 이것은 흐름 제어(if, while, for문 등)를 지원하지 않는 것을 제외하고 C와 비슷한 문장구조를 가지고 있다. 시스템 트레이딩의 본질 때문에 DTrace framework는 그 자신만의 흐름을 가지고 있는데, 이것은 awk와 아주 유사한 패턴 일치 문장구조(pattern match syntax)를 사용하는 것에 맞게 되어 있다.

DTrace는 커널이나 사용자 프로그램의 특정 위치에 probe들을 삽입("instrumenting probe"라고도 표현함)하여 그것의 기능을 수행한다. 이 probe들이 특정 위치에 도달했을 때 그 이벤트와 관련된 정보를 제공하는 데이터가 기록된다. 제공된 정보는 probe의 타입에 따라 다양하다. D 언어를 통해 어디에 probe가 삽입되고, 어떻게 그 이벤트가 보고되는지에 대해 DTrace에 지정할 수 있다.

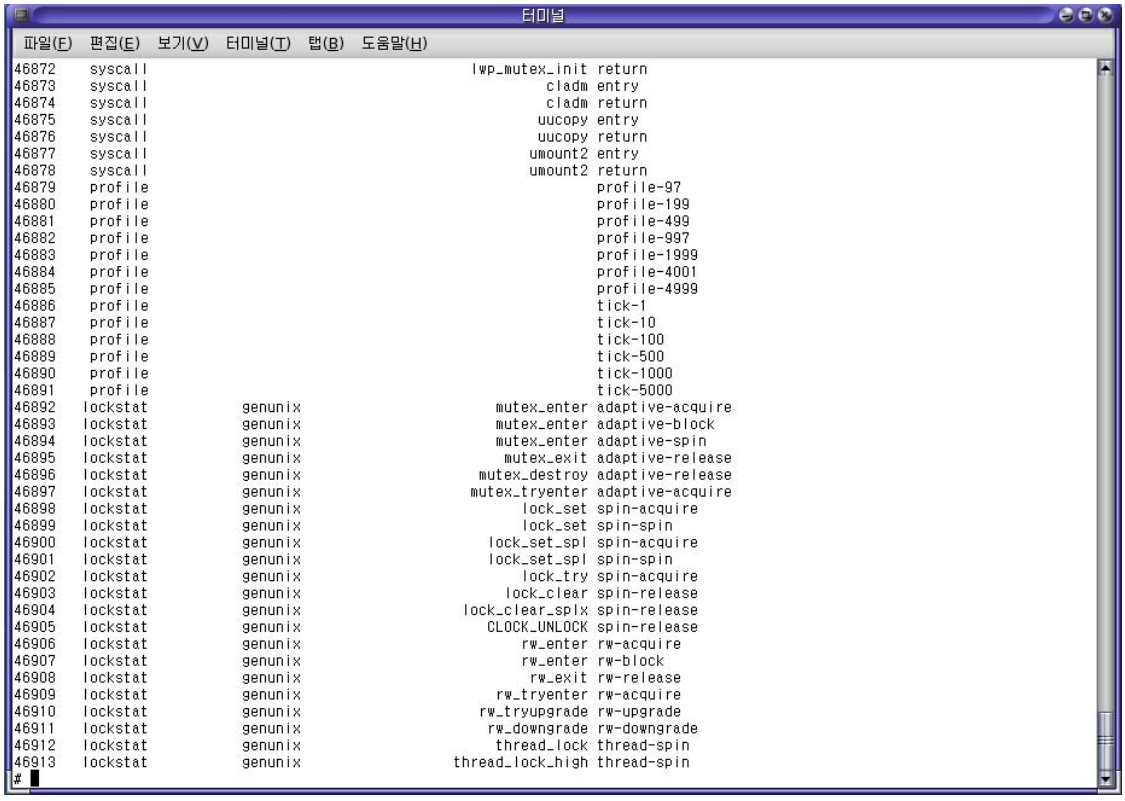
Listing Probes

스크립트를 실제로 작성하기 전에 DTrace를 살펴보는 가장 간단한 방법은 커맨드 라인에서 그것을 직접 사용해보는 것이다. DTrace에서 가장 간단한 명령 옵션은 여러분들의 시스템에 삽입될 수 있는 가능한 모든 probe들의 목록을 확인해보는 것이다.

```
# dtrace -l
   ID   PROVIDER    MODULE          FUNCTION  NAME

    1    dtrace                             BEGIN
    2    dtrace                             END
    3    dtrace                             ERROR
```

```
   4    syscall                    nosys  entry
   5    syscall                    nosys  return
   6    syscall                    rexit  entry
   7    syscall                    rexit  return
   8    syscall                   forkall  entry
   9    syscall                   forkall  return

  -- 중략 --

38329       fbt       zmod             _info  entry
38330       fbt       zmod             _info  return
38331       fbt       zmod         z_strerror  entry
38332       fbt       zmod         z_strerror  return
38333       fbt       zmod       z_uncompress  entry
38334       fbt       zmod       z_uncompress  return
```

각 probe는 4개의 부분(tuple: provider, module, function, name)으로 구성되어 있다. 위의 목록은 probe의 고유 확인 번호와 함께 각 probe의 4개 부분을 보여주고 있다. 마지막 라인을 보면 필자의 시스템은 38,334개의 이용 가능한 probe들이 있다는 것을 알 수 있다. 이것은 시스템마다 다를 수 있다.[1]



[1] 역자의 시스템 Solaris 10은 46,913개의 probe가 있었다(바로 위의 그림 참고).

probe provider는 특정 타입의 삽입('instrumentation')을 실행하는 커널 프로그램이다. 예를 들어, "syscall" provider는 커널 시스템 호출 인터페이스(read, write, fork, 등)에 probe를 제공한다. 이 경우, 각 시스템 호출에 대해 두 개의 가능한 probe가 있는데, 하나는 그 시스템 호출로 들어가기(entry) 위한 것이고, 다른 하나는 그 시스템 호출로부터 나오기(exit) 위한 것이다. 완전히 다른 방식으로 삽입하는 또 다른 하나의 provider는 "profile" provider이다. 이 provider는 지정된 시간 간격으로 이벤트들을 기록하는 probe들을 참작한다. "profile" provider에 대해서는 나중에 더 알아보기로 한다.

Probe tuple의 다른 부분들은 그 probe의 위치를 정의한다:

- probe가 위치한 "module"(만약 그 probe가 프로그램의 위치와 상응한다면)
- 그 module 내의 "function"
- "entry" 또는 "return"과 같이 probe가 기록하는 이벤트를 일반적으로 정의하는 "name"

어떤 probe들은 tuple의 "module" 부분을 가지고 있지 않으며, 어떤 것은 추가로 "function" 부분도 가지고 있지 않다. 하지만, 모든 probe들은 적어도 "provider"와 "name"을 가지고 있다.

우리는 tuple를 콜론으로 경계를 지정함으로써 어떤 probe를 처리하길 원하는지에 대해 추가로 정의할 수 있다. 예를 들어, "lockstat:genunix:mutex_exit:adaptive-release"라는 tuple은 "adaptive-release"라는 이름을 가진 probe를 지칭하는데, 이것은 "genunix" 모듈에 "mutex_exit"라는 함수에 있는 것이고, "lockstat" provider에 의해 제공된다. 이와 같은 probe의 목록을 열거하기 위해 다음과 같이 지정할 수 있다.

```
dtrace -ln lockstat:genunix:mutex_exit:adaptive-release
   ID   PROVIDER      MODULE       FUNCTION  NAME
  470   lockstat     genunix     mutex_exit  adaptive-release
```

-n 플래그는 우리가 이름으로 어떤 tuple을 지정하게 해주는데, 즉, DTrace는 지정된 tuple의 마지막 구성요소가 이름 구성요소라는 것을 추정한다.

우리가 그 다른 구성요소들을 tuple의 마지막 것으로 지정하도록 해주는 다른 세 개의 flag가 있다.

```
# dtrace -lf lockstat:genunix:mutex_exit
   ID   PROVIDER      MODULE       FUNCTION  NAME
  470   lockstat     genunix     mutex_exit  adaptive-release
```

```
# dtrace -lm lockstat:genunix
ID   PROVIDER        MODULE       FUNCTION  NAME
  467   lockstat     genunix    mutex_enter  adaptive-acquire
  468   lockstat     genunix    mutex_enter  adaptive-block
  469   lockstat     genunix    mutex_enter  adaptive-spin
  470   lockstat     genunix     mutex_exit  adaptive-release
  471   lockstat     genunix   mutex_destroy  adaptive-release
...

# dtrace -lP lockstat
   ID   PROVIDER      MODULE       FUNCTION  NAME
  467   lockstat     genunix    mutex_enter  adaptive-acquire
  468   lockstat     genunix    mutex_enter  adaptive-block
  469   lockstat     genunix    mutex_enter  adaptive-spin
  470   lockstat     genunix     mutex_exit  adaptive-release
  471   lockstat     genunix   mutex_destroy  adaptive-release
...
```

우리가 이와 같이 단지 부분적으로 tuple을 지정할 때, 일반적으로 하나 이상의 probe와 일치시킨다. 또한, 모든 lockstat probe를 지정하는 것은 모든 lockstat:genunix probe들을 지정하는 것과 같다는 것을 주목해야 하는데, 왜냐하면 lockstat의 경우 단지 하나의 module이 있기 때문이다.

우리는 또한 placeholder [2] 로써 콜론을 사용하거나 tuple의 하나 또는 그 이상의 부분을 빼버림으로써 하나 이상의 probe를 매칭시킬 수 있다. 예를 들어, 다음은 동등한 probe 지정이며, 같은 세트의 probe들을 매칭시킬 것이다.

```
dtrace -lP lockstat
dtrace -lm lockstat:
dtrace -lf lockstat::
dtrace -ln lockstat:::
```

다음 두 dtrace 명령은 이용 가능한 probe들을 매칭시킬 것이다.

```
dtrace -l
dtrace -ln :::
```

---

[2] <수학> <논리> 플레이스홀더: 식(式) 안의 문자 중, 정해진 집합의 요소 이름을 대입할 수 있는 것

Instrumenting Probes

비록 우리가 흥미를 가진 probe들의 목록을 열거할 수 있는 것은 유용하지만 실제로 그것들 실제로 삽입하는 것이 훨씬 더 유용하다. 우리가 열거해온 같은 probe들을 삽입하기 위해 커맨드 라인으로부터 간단히 **-l** 플래그를 제거하면 된다. 예를 들어, 다음 명령들 모두는 같은 세트의 probe를 삽입하는데, 이 probe의 모든 것은 lockstat에 의해 제공된다.

```
dtrace -P lockstat
dtrace -m lockstat:
dtrace -f lockstat::
dtrace -n lockstat:::
```

probe들이 삽입될 때 기본적인 행위는 각 probe가 시동할 때 그 probe가 시동된 CPU#, 그 probe의 ID#, probe 함수, 그리고 probe의 이름을 쓰는 것이다:

```
# dtrace -P lockstat | head
dtrace: description 'lockstat' matched 22 probes
CPU     ID                    FUNCTION:NAME
  0    470    mutex_exit:adaptive-release
  0    470    mutex_exit:adaptive-release
  0    467   mutex_enter:adaptive-acquire
  0    470    mutex_exit:adaptive-release
  0    467   mutex_enter:adaptive-acquire
  0    470    mutex_exit:adaptive-release
  0    477         lock_try:spin-acquire
  0    478       lock_clear:spin-release
  0    473         lock_set:spin-acquire
```

이 dtrace 세션은 lockstat(22개의 probe)에 의해 제공된 모든 probe들을 삽입한다. 각 probe가 시동될 때, 한 라인의 출력물이 생성된다. 이 지점에서, lockstat provider의 목적은 아주 명확해야 하는데, 커널에서 lock 이벤트(mutex와 spinlock)를 추적하는 방법을 제공하는 것이다. 위의 trace는 그것이 발생할 때마다 각 lock의 목록을 열거한다(만약 우리가 헤더에 dtrace 명령을 pipe하지 않았다면 그 명령은 영원히 실행될 것이며, kill 명령이나 Ctr-C에 의해 인터럽트되어야 할 것이다).

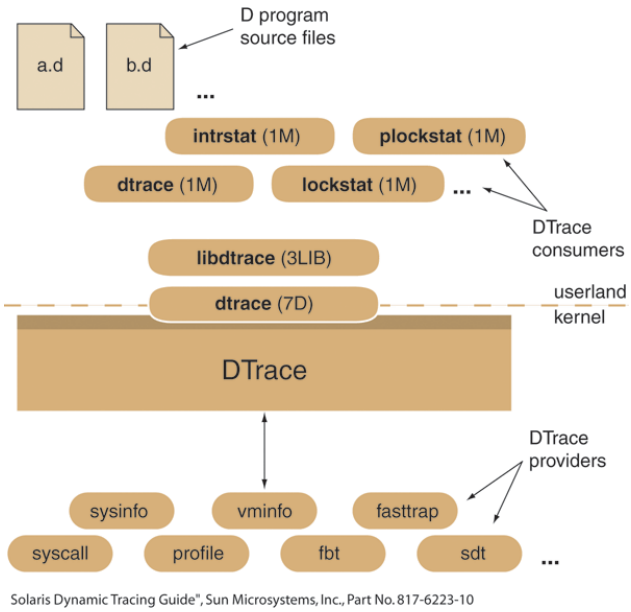
DTRACE 힌트: DTrace 명령과 스크립트를 실행할 때 삽입된 아주 많은 probe들로 인해 시스템에 과부하가 걸리지 않도록 하는 것이 중요하다. 여러분들은 시스템 상의 모든 probe들을 실제로 삽입할 수 있으며, 그것을 처리하겠지만 그것은 다소 CPU 집약적이며, 덧붙여 이벤트 큐들이 차게

되면 많은 드롭된 이벤트과 함께 끝을 낸다.(DTrace는 그것이 이벤트들을 드롭할 때를 알려준다.) 대부분 경우, 만약 여러분들이 수 천 개의 probe를 삽입한다면 아마도 너무 많은 정보를 수집하는 것이 될 것이다. 이에 대한 해결책으로 항상 **-l** 플래그와 함께 dtrace 명령을 내리거나 스크립트를 먼저 실행하는 것이다. 이것은 실제로 삽입하는 것보다는 여러분이 삽입하고 있는 probe들의 목록만을 열거할 것이다. 만약 여러분들이 삽입하려고 했던 probe의 수를 카운트하길 원한다면 DTrace 명령의 리스트 폼의 출력결과를 **wc -l**에 pipe하고, 그 헤더에 하나는 뺀다.

Inside DTrace

이 시점에서, 어떻게 Solaris가 시스템의 나머지에 영향을 주지 않고 이 모든 것을 할 수 있는지 물어볼 수 있다[1]. DTrace 기능은 '생산자'(producer)와 '소비자'(consumer)들의 컬렉션이다. DTrace probe는 그것들이 시동될 때 버퍼(DTrace 소비자당 프로세서 당 하나씩) 안으로 놓이는 이벤트들을 생산한다. DTrace 소비자들(예를 들어, dtrace 명령)은 버퍼로부터 이벤트를 끌어내고 그것들을 처리한다. [그림1]은 DTrace 생산자와 소비자들 사이에 관계를 보여준다[2]. DTrace의 내부에 대한 추가 정보는 Bryan M. Cantrill, Michael W. Shapiro, 그리고 Adam H. Leventhal의 글 "Dynamic Instrumentation of Production Systems" [3] 를 참고해라. 여러분들은 단지 관심을 가진 probe들만 삽입하기 때문에 수집된 데이터의 양은 최소한으로만 유지된다. 덧붙여, DTrace는 probe들로부터 데이터에 기반을 둔 관계적인 표현들인 술어(predicate)들을 이용해 후에 처리되어야(post-process) 하는 이벤트들의 수를 제한한다.

우리는 CPU, probe ID, 함수, 그리고 name을 쓰는 기본 행위에 국한되지는 않는다. 우리는 probe가 시동될 때 그 행위를 맞출(tailor) 수 있으며, 이벤트 데이터의 내용들과 probe들이 시동하는 것에 기반을 둔 다른 행위를 수행할 수 있다. 이 행위는 D 언어를 사용하는데, 필자는 이 시리즈의 part2에서 다룰 것이다.



Solaris Dynamic Tracing Guide", Sun Microsystems, Inc., Part No. 817-6223-10

[3] http://www.sun.com/bigadmin/content/dtrace/dtrace_usenix.pdf

Summary

이 시리즈의 part 1에서 우리는 DTrace를 사용하는 것의 서두만 살펴보았다. 다음 글에서 필자는 D 프로그래밍 언어를 설명하고, 시스템 문제들을 해결하기 위해 어떻게 Dtrace를 사용해야 하는지 보여줄 것이다. DTrace는 이전 툴들이 허용하는 것보다 실행 중인 시스템을 더 깊게 살펴볼 수 있도록 해준다. 이 시리즈를 끝낼 쯤 필자는 DTrace가 실로 Solaris 10의 가장 혁신적인 기능들 중의 하나임을 동의할 것이라고 생각한다.

Endnotes

1. Being a fan of "Star Trek"(R), I enjoy the correlation between "system monitoring impact" and "transporter technology". Werner Heisenberg (1901-1976), renowned physicist, proposed what is referred to as the Heisenberg Uncertainty Principle. In the area of system monitoring, the principle loosely defines how the monitor impacts the system. That is, the more in depth and detailed you monitor and describe what is happening on your system, the more the monitor itself changes and impacts the system.

The fun correlation to "Star Trek" is that the principle is also used to describe the impact of transporter technology on the thing being transported: the more you scan to copy an object, the more you disrupt it (see "Quantum Teleportation": **http://www.research.ibm.com/quantuminfo/teleportation/**). Of course, because the show was based on a certain amount of real science, the engineers in "Star Trek" had at their disposal a device called the "Heisenberg Compensator", which not only added a safety mechanism to the transporter, but also gave the writers something accurate sounding to toss around in the script (i.e., "we had to modify the Heisenberg Compensator", or "the transporter failed because of a problem with the Heisenberg Compensator"). ("Star Trek" is a registered trademark of CBS Studios Inc.)

2. From "Solaris Dynamic Tracing Guide", Sun Microsystems, Inc., Part No. 817-6223-10 -- **http://docs.sun.com/app/docs/doc/817-6223/6mlkidlf1?a=view**

*Chip Bennett (**cbennett@laurustech.com**) is a systems engineer for Laurus Technologies (**http://www.laurustech.com**), a Sun Microsystems partner. He has more than 20 years experience with Unix systems, and holds a B.S. and M.S. in Computer Science. He consults with Laurus and Sun customers on a variety of Solaris related topics, and he co-chairs OS-GLUG, the OpenSolaris Great Lakes User Group (Chicago).*