

Systemtap

Profiling Tech Talks

Lucian Adrian Grijincu
lucian.grijincu@rosedu.org

ROSEdu

11 decembrie 2009

- 1 Ce e systemtap?
- 2 Instalare (Ubuntu)
- 3 Demo
- 4 Explicații
- 5 User space/kernel space
- 6 Moar demos!

- 1 Ce e systemtap?
- 2 Instalare (Ubuntu)
- 3 Demo
- 4 Explicații
- 5 User space/kernel space
- 6 Moar demos!

Cum lucrăm fără systemtap ?

un program cu bug-uri (câte ați găsit?)

```
1 int factorial(int n)
2 {
3     int ret;
4     do {
5         ret *= n;
6     } while(n--);
7     return 0;
8 }
```

Cum lucrăm fără systemtap ?

băgăm niște printfuri

```
1 int factorial(int n)
2 {
3     int ret;
4     printf("f> enter n=%d ret=%d\n", n, ret);
5     do {
6         ret *= n;
7     } while(n--);
8     printf("f> exit  n=%d ret=%d\n", n, ret);
9     return ret;
10 }
```

Cum lucrăm fără systemtap ?

corectăm și mai băgăm niște printfuri

```
1 int factorial(int n)
2 {
3     int ret = 0;
4     printf("f> enter n=%d ret=%d\n", n, ret);
5     do {
6         ret *= n;
7     } while(n--);
8     printf("f> exit  n=%d ret=%d\n", n, ret);
9     return ret;
10 }
```

Cum lucrăm fără systemtap ?

și tot așa ...

```
1 int factorial(int n)
2 {
3     int ret = 1;
4     printf("f> enter n=%d ret=%d\n", n, ret);
5     do {
6         ret *= n;
7         printf("n=%d ret=%d\n", n, ret);
8     } while(n--);
9     printf("f> exit  n=%d ret=%d\n", n, ret);
10    return ret;
11 }
```

Cum lucrăm fără systemtap ?

la final, curățăm programul

```
1 int factorial(int n)
2 {
3     int ret = 1;
4     do {
5         ret *= n;
6     } while(--n);
7     return ret;
8 }
```

Cum lucrăm fără systemtap ?

- repetitiv
- pentru fiecare `printf` trebuie recompilat proiectul (timpii de compilare pot fi foarte mari)
- poate introduce alte probleme
- nu prea este la îndemâna administratorilor de sistem
- după ce se rezolva problema, sunt șterse, și dacă apare mai târziu o problemă în același loc, sunt rescrise
- ...

Cu ce ajută systemtap ?

- permite *administratorilor* și *programatorilor* să scrie scripturi simple cu care să monitorizeze activitatea internă a kernelului sau a aplicațiilor
- datele pot fi
 - extrase
 - filtrate
 - sumarizate
- totul în siguranță, fără a introduce probleme noi
- există și posibilitatea modificării rezultatelor

Instalare (Ubuntu)

- systemtap are nevoie de:

- suport în kernel

```
1 Kernel hacking --->
2     [*] Kernel debugging
3         [*]   Compile the kernel with debug info
4 Instrumentation Support --->
5     [*] Kprobes (EXPERIMENTAL)
```

- simboluri de debugging
- în Ubuntu, suportul este compilat în kernel
- simbolurile de debugging pot fi descărcate
 - <http://ddebs.ubuntu.com> karmic main
 - `wget http://ddebs.ubuntu.com/pool/main/l/linux/linux-image-debug-2.6.31-16-generic_2.6.31-16.53_i386.ddeb`

Testarea instalării

```
1 $ cat hello.stp
2 probe begin
3 {
4     print("hello rtt")
5     exit()
6 }
7 $ stap hello.stp
8 hello rtt
```

system-wide strace

```
1 #!/usr/bin/env stap
2 probe syscall.open
3 {
4     printf("%s(%d) open (%s)\n",
5           execname(), pid(), argstr)
6 }
7
8 probe timer.ms(4000)
9 {
10    exit()
11 }
```

nr apeluri de sistem/pid

```
1 #! /usr/bin/env stap
2 # tema 1 la PSO :)
3 global syscalls
4 probe begin {
5     print ("Collecting data... Ctrl-C to exit\n")
6 }
7
8 probe syscall.* {
9     syscalls[pid()]++
10 }
11
12 probe end {
13     printf ("%s %s\n", "#SysCalls", "PID")
14     foreach (pid in syscalls-)
15         printf ("%d %d\n", syscalls[pid], pid)
16 }
```

nr apeluri de sistem/proces

```
1 #! /usr/bin/env stap
2 # deja mai mult decât tema la PSO
3 global syscalls
4 probe begin {
5     print ("Collecting data... Ctrl-C to exit\n")
6 }
7
8 probe syscall.* {
9     syscalls[execname()]++
10 }
11
12 probe end {
13     printf ("%10s %-s\n", "#SysCalls", "Procname")
14     foreach (proc in syscalls-)
15         printf ("%10d %-s\n", syscalls[proc], proc)
16 }
```

Cum funcționează systemtap ?

- traduce scripturile în C (CPL)
- compilează din ele un modul de kernel (CPL)
- introduce modulul în kernel (PSO)

Cum funcționează systemtap ?

- la introducerea modulului:
activează evenimentele monitorizate (punctele de colectare a evenimentelor trebuie să fie deja definite)
- la declanșarea unui eveniment monitorizat:
se execută rutinele specificate pentru evenimentul monitorizat
- la oprirea programului stap:
dezactivează evenimentele și se descarcă modulul

Cui îi pasă de kernel, vrem să inspectăm temele scrise în C

- se definesc probele ce se inspectează:
- ```
1 provider myprogram {
2 probe func1();
3 probe func2(int);
4 }
```
- din fișierul cu probele se obțin un fișier antet și un fișier obiect cu tot ce e nevoie pentru a folosi noile probe
- ```
1 probes.h: probes.d
2     dtrace -C -h -s $< -o $@
3 probes.o: probes.d
4     dtrace -C -G -s $< -o $@
```
- nu e nevoie să modificați codul sursă (în afara unui `#include "probes.h"`)

Cui îi pasă de C, vrem să inspectăm temele scrise în Python

- recent s-a introdus suport pentru inspectarea codului Python
- expect it soon in a distro release near you

?

Toate exemplele sunt furate de pe pagina proiectului:
<http://sourceware.org/systemtap/examples/index.html>