SystemTap Tutorial

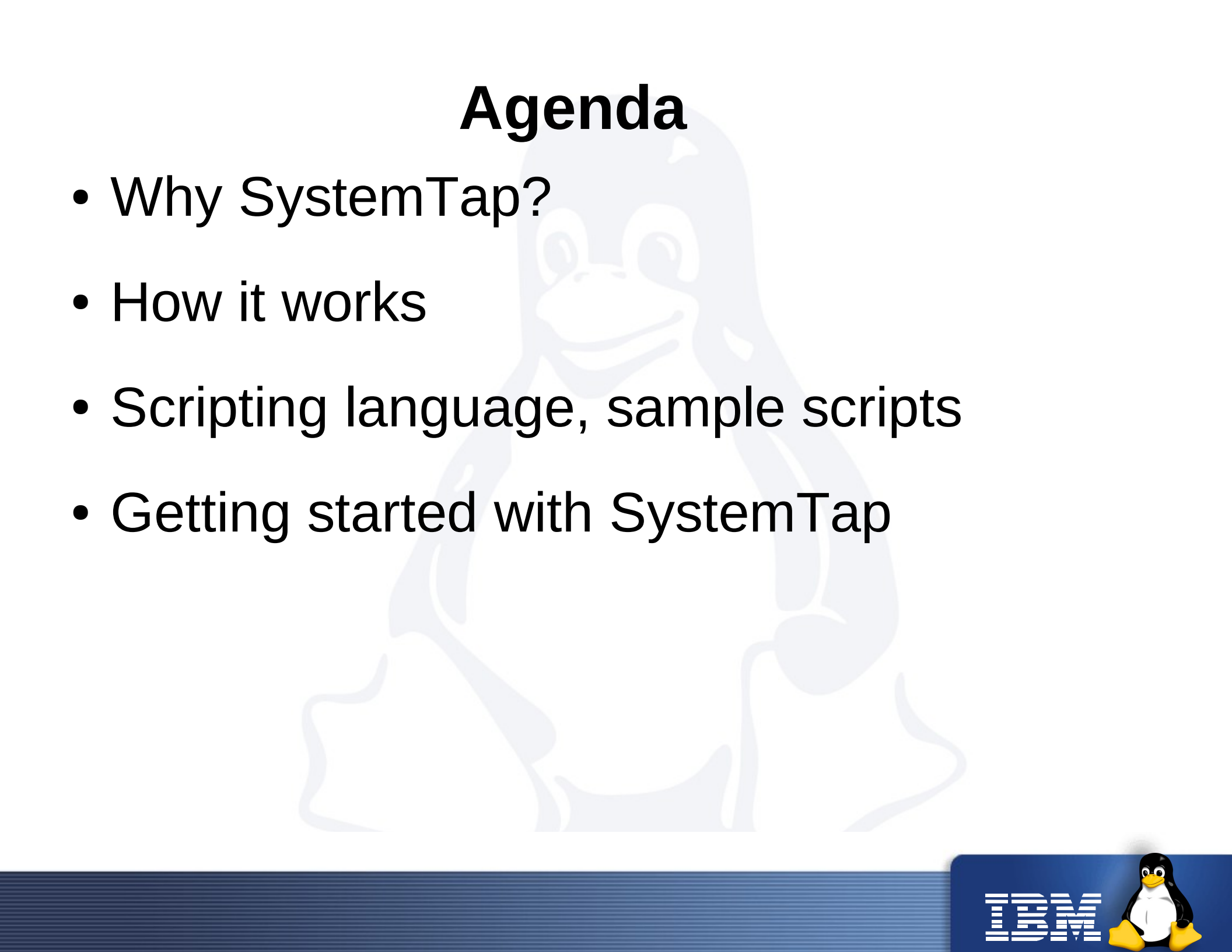
Jim Keniston: jkenisto@us.ibm.com

SystemTap team: systemtap@sources.redhat.com

September 18, 2008

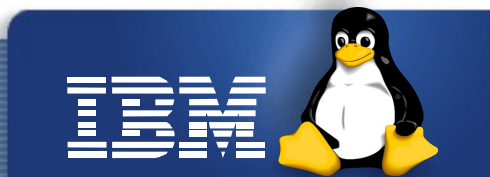
Agenda

- Why SystemTap?
- How it works
- Scripting language, sample scripts
- Getting started with SystemTap



Why SystemTap?

- Script-directed dynamic tracing/probing for Linux
  - Install/remove instrumentation on the fly.
  - No changes to your source code
    - No rebuild/restart needed
    - Keep ad hoc instrumentation separate from source.
- System-wide view
  - Kernel
  - Applications
  - Integrated view of kernel, applications
- Probe wherever you want, however you want.
- Debugging, performance analysis, education



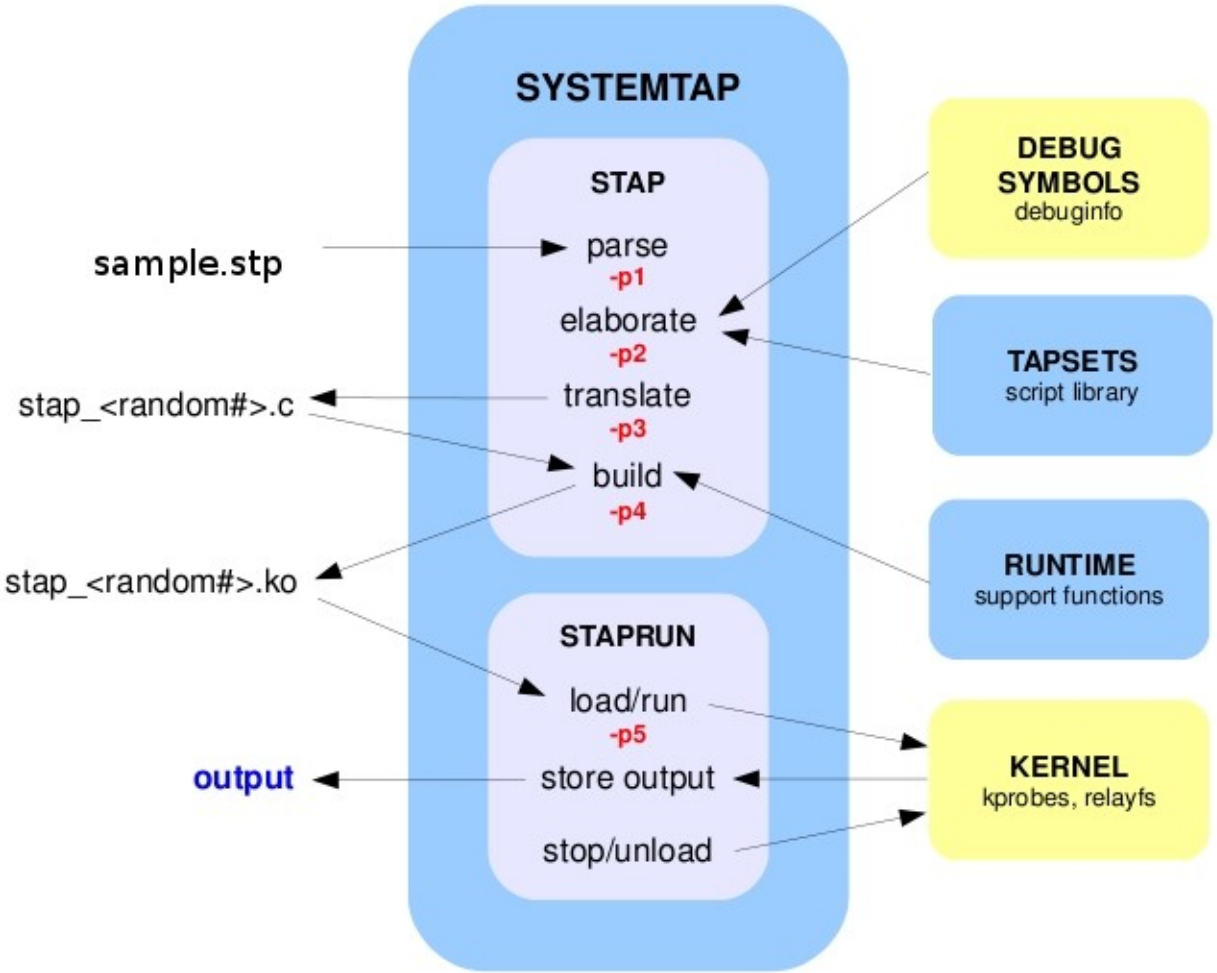
How it works

```
$ cat sync.stp
probe kernel.function("sys_sync") {
   printf("sys_sync called\n");
}
$ stap -v sync.stp
...
Pass 5: starting run.
sys_sync called
sys_sync called
^C
Pass 5: run completed in 10usr/10sys/...
```

How it works -- cont.

- Stap translates the script to a kernel module .c file (passes 1-3).
- Stap compiles and links the module (pass 4).
- Staprun loads the module (pass 5).
- Instrumentation messages (e.g., from printf()) go to stap's stdout.
- When the script terminates (e.g., via CTRL-C), staprun unloads the module.

Even more detail...



Scripting language

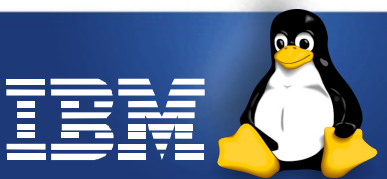
- Awk-like, C-like
- Can dip into C in tapsets or guru (stap -g) mode
- Primary construct: probe *event* { *handler* }

event:

```
kernel.function("sys_sync")
```

handler:

```
{ printf("sys_sync called\n"); }
```



Probe Statement Examples

- `probe kernel.function("funcname")`
- `probe kernel.function("funcname").return`
- `probe module("modname").function("funcname")`
- `probe kernel.statement("*@pathname:247")`
- `probe kernel.mark("markername")`
- `probe process("a.out").function("funcname")`
- `probe process("a.out").thread.begin`
- `probe begin, probe end`
- `probe timer.sec(10)`
- `probe timer.jiffies(100).randomize(20)`

Names of functions, modules, and files can contain wildcard characters.

begin, end, timer probes

```
$ cat hibye.stp
probe begin {printf("Hello, world!\n");}
probe timer.sec(5) {exit();}
probe end {printf("Good-bye, world!\n");}
$ stap hibye.stp
Hello, world!
[5 seconds later]
Good-bye, world!
$
```

Variables

- Script variables
  - Global or local
  - Implicitly typed: type inferred from usage
    - Integers (64-bit signed)
    - Strings
    - Associative arrays (global only)
    - Statistics aggregates (global only)
  - Implicitly initialized to zero/empty
- Target variables
  - Values from the probepoint context
  - $ prefix – e.g., $task->pid
  - Magic variables – e.g., $return, $$parms, $format/ $name for markers
  - kernel_string(), user_string() for safe access to strings

Script and target variables

```
global failures, big_reads
probe kernel.function("sys_open").return {
   if ($return < 0) failures++;
}
probe kernel.function("sys_read") {
   if ($count > 4*1024) big_reads++;
}
```

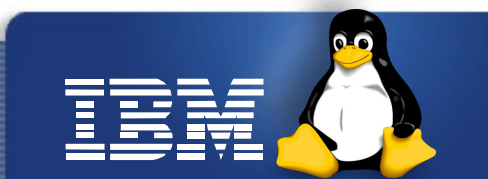
Script variables: failures, big_reads

Target variable: $count – sys_read()'s 3rd arg

Magic context variable: $return

Script statements

- Semicolon separator is optional
- Group compound statements with **{ }**
- Branching
  - **if (COND) STMT [else STMT]**
- Looping:
  - **while (COND) STMT**
  - **for (INIT; COND; ITER) STMT**
  - **foreach (VAR in ARRAY [limit NUM]) STMT**
  - **foreach ([VAR1, VAR2] in ARRAY [limit NUM]) STMT**
  - **break; continue;**
- Other:
  - **return [VAL]; next; delete VAR;**



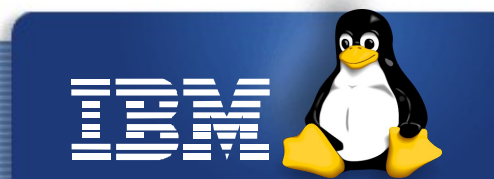
Associative arrays

```
global syscalls
probe kernel.function("sys_*") {
   syscalls[probefunc()]++
}
probe timer.sec(5), end {
   foreach (func in syscalls- limit 10)
      printf("%6d %s\n", syscalls[func],
        func);
   delete syscalls   // Clear the array.
}
```

Associate array: syscalls

Indexed by function name

Print top 10 in descending order every 5 seconds.



Sample output

```
3073 sys_fstat64
3053 sys_gettimeofday
2373 sys_stat64
2030 sys_access
2030 sys_faccessat
1453 sys_brk
1205 sys_mprotect
1019 sys_llseek
 744 sys_munmap
 585 sys_ioctl
 575 sys_lstat64
...
```

Associative arrays - cont.

- Array indexes can be strings, integers, or any combination thereof.

- Even a stack backtrace can be used as an index:

```
backtraces[backtrace()]++
```

- Associative arrays can be used as:
  - sets:

```
if (!(thisfunc in exceptions)) { ... }
```

  - structs (array indexes = struct members)
  - flight-recorder circular buffers:

```
nxt++; nxt %= bufsz; buf[nxt] = message
```

Writing Functions

- Your script can include functions:

```
function FNAME:type(ARG1:type, ARG2:type) {
    /* code to run when FNAME is called */
    return SOMETHING
}
```

- The **:type**s are optional, and may be **:string** or **:long**.

Predefined functions

- Printing
  `printf(), sprintf(), etc.`
- Strings
  `strlen(), substr(), isinstr(), strtol()`
- Timestamps
  `get_cycles(), gettimeofday_s(), gettimeofday_ns()`
- Context
  `cpu(), execname(), tid(), pid(), uid(), backtrace(), print_stack(), print_backtrace(), pp(), probefunc(), probemod()`

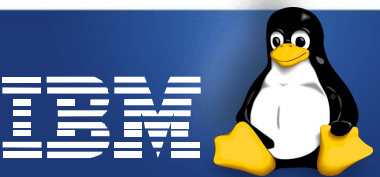
- See **`man stapfuncs`** for details and many more.

Tapsets = probe libraries

- Tapsets
  - provide abstractions of common probepoints,
  - provide values of interest from those probepoints, and
  - define functions... all for use by your script.

- Tapsets are SystemTap scripts.
  - not runnable (probe aliases, not probes)
  - installed in /usr[/local]/share/systemtap/tapset/
  - `stap -Idir` says to look in *dir* as well.

- Typically captures expertise about a particular subsystem or app.



Tapset examples

`syscall.*`

- Probes each system call, provides **`name`** and **`argstr`**

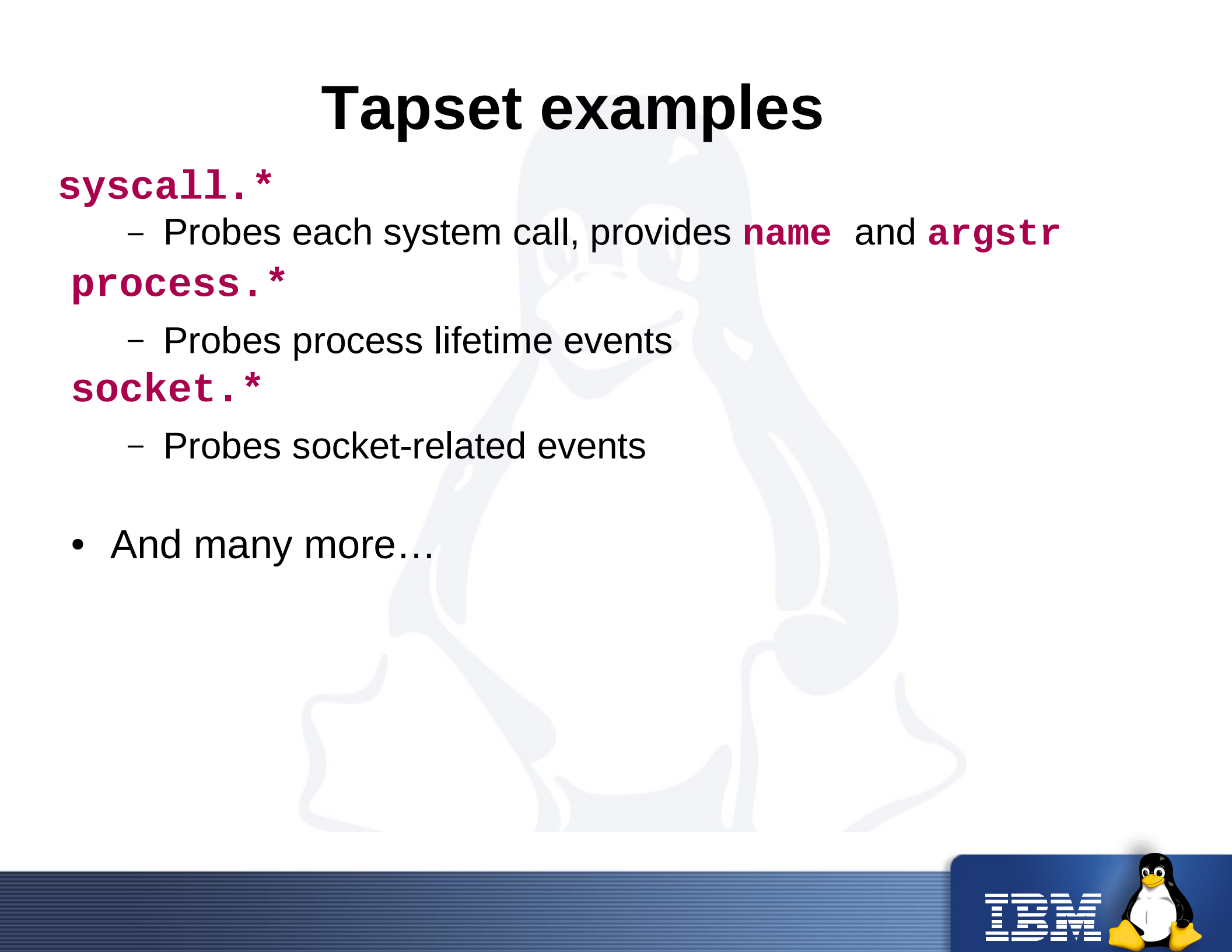
`process.*`

- Probes process lifetime events

`socket.*`

- Probes socket-related events

- And many more…



Example: syscall tapset

without syscall tapset:

```
probe kernel.function("sys_*") {
   syscalls[probefunc()]++
}
```

using syscall tapset:

```
probe syscall.* {
   syscalls[name]++
}
```

Syscall tapset, cont.

For every system call, syscalls.stp provides:

- name: syscall name
- argstr: argument values encoded in a string
- (usually) individual arg values, named according to the man pages
- retstr: return value encoded in a string

```
probe syscall.* {
   printf("%s(%s)\n", name, argstr)
}
probe syscall.*.return {
   printf(“%s returns %s\n”, name, retstr)
}
```

Tapsets define probe aliases

Probe alias in tapset:

```
probe syscall.write= kernel.function("sys_write")
{
        name = "write"; fd = $fd; buf_uaddr = $buf;
        count = $nbytes; argstr = sprintf(...)
}
```

Probe in script:

```
probe syscall.write {
```

automatically inserted from tapset:

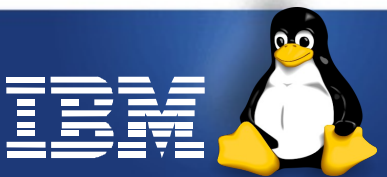
```
        name = "write"; fd = $fd; buf_uaddr = $buf;
        count = $nbytes; argstr = sprintf(...)
    wr_bytes_requested += count
}
```

Tapsets also define functions

```
function execname:string () %{ /* pure */
   strlcpy (THIS->__retvalue, current->comm,
     MAXSTRINGLEN);
%}
function pid:long () %{ /* pure */
   THIS->__retvalue = current->tgid;
%}
```

Embedded C

- Embedded C is copied directly from your script to the module .c file.
- Embedded C is allowed only in tapsets and in scripts compiled with stap -g (guru mode).
- %{ *embedded C* %}
- in functions: THIS->*argname*, THIS->retvalue

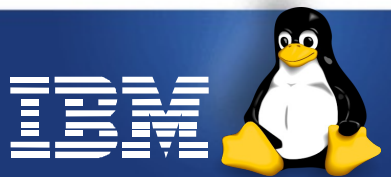


SystemTap safety features

- Stap-generated modules include defensive code...
- Handlers:
  - cannot run for long.
  - cannot allocate memory.
  - cannot perform unsafe accesses/computations.
- Locking to ensure correctness
- Resource limits can be adjusted with “stap -D”
- Stap refuses to probe blacklisted functions.
- Guru mode
  - more flexibility / power
  - less safety
  - change kernel data

Getting started with SystemTap

- You need:
  - debug info (DWARF)
    - Build your kernel/app with -g, or get debuginfo rpm from distro.
    - SystemTap can [again, soon] do some probing without debuginfo.
  - elfutils
  - kprobes and/or markers (both upstream)
- For user-space probing, you need:
  - utrace (not yet upstream)
  - uprobes (trailing utrace)



Installing SystemTap

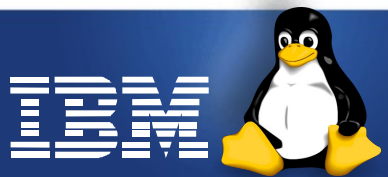
- On Fedora:
  - `yum install systemtap kernel-devel`
  - `yum --enablerepo=\*-debuginfo install kernel-debuginfo`
- Instructions for installing SystemTap on other distros are available on the SystemTap wiki – e.g.,

  http://sourceware.org/systemtap/wiki/SystemtapOnUbuntu

  .../SystemtapOnopenSuse

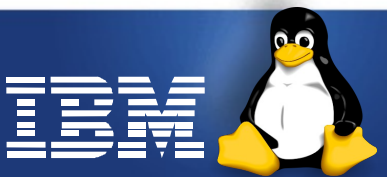
  .../SystemtapOnDebian
- See also

  http://sourceware.org/systemtap/wiki/SystemTapWithSelfBuiltKernel



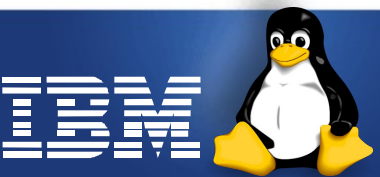
Installing Latest SystemTap

- Download SystemTap from "`git clone http://sources.redhat.com/git/systemtap.git systemtap`"
- Download and extract elfutils from `ftp://sources.redhat.com/pub/systemtap/elfutils/elfutils-<nnn>.tar.gz`
- `./configure –with-elfutils=<elfutils dir>`
- `make all`
- `[make check to run test suite]`
- `make install (as root)`



SystemTap Important Links

- Wiki http://sourceware.org/systemtap/wiki
- Tutorial http://sourceware.org/systemtap/tutorial/
- FAQ http://sourceware.org/systemtap/wiki/SystemTapFAQ
- Language Reference http://sourceware.org/systemtap/langref/
- Using Markers http://sourceware.org/systemtap/wiki/UsingMarkers
- Report Bugs http://sourceware.org/systemtap/wiki/HowToReportBugs
- LPC2008 Slides http://sourceware.org/systemtap/wiki/LPC2008 [soon]



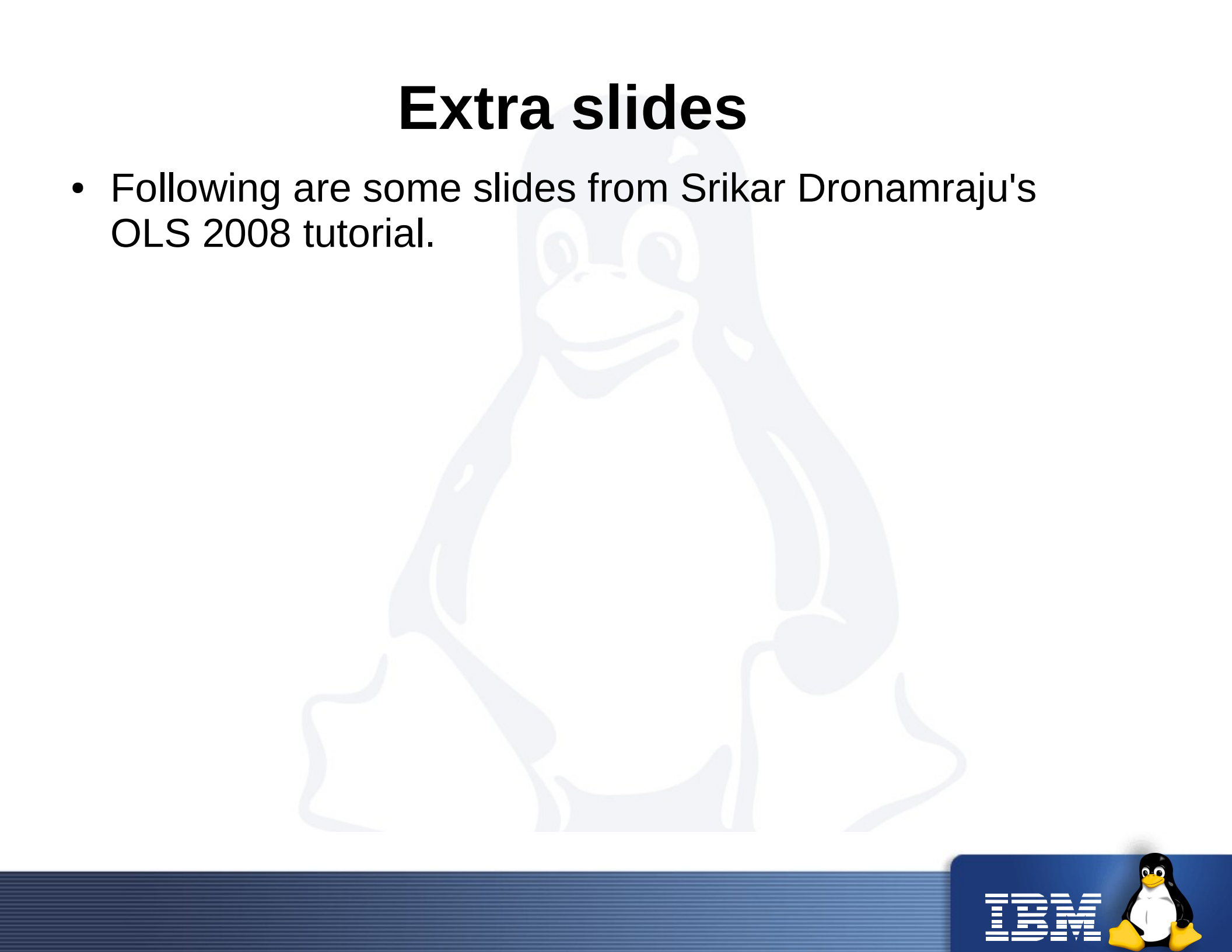
Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM is a registered trademark of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.

Questions?

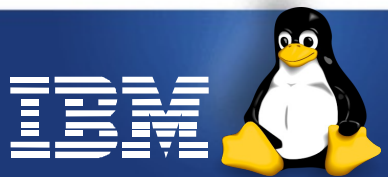
Extra slides

- Following are some slides from Srikar Dronamraju's OLS 2008 tutorial.



Sample Scripts 1

```
#!/usr/bin/env stap
# Simple HelloWorld Script.

probe begin {
        print("Hello World!\n")
        print("waiting for 10 seconds\n")
}

# Exit after 10 seconds
probe timer.ms(10000) { exit () }

probe end {
        print("Iam  Done .... Bye !\n")
}
```

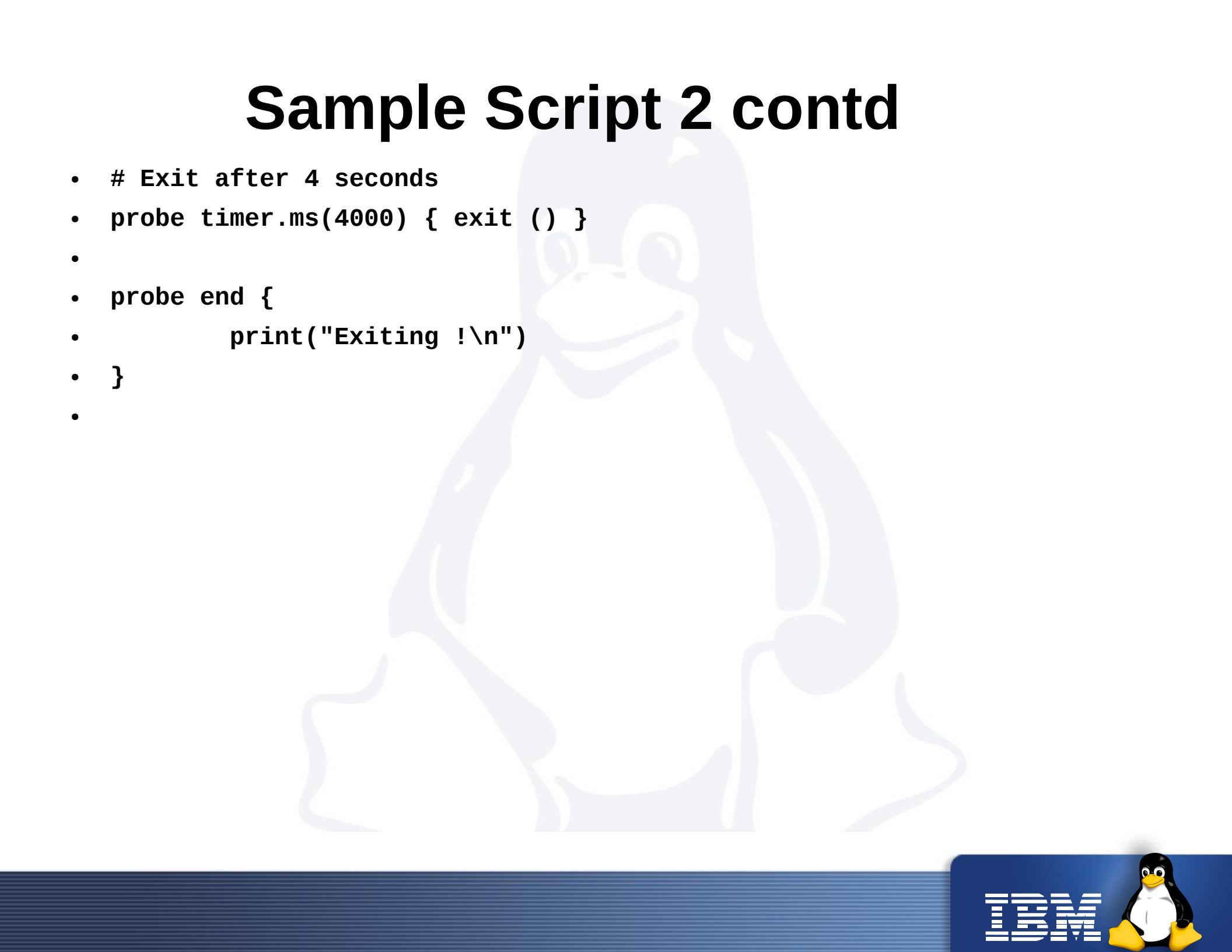


Sample Script 2

```
#! /usr/bin/env stap
# probe sys_open
# Trace open calls.

probe begin {
        print("Logging Information\n")
}


probe kernel.function("sys_open")
{
        filename = user_string($filename)
        printf("%s(%d) open (%s)\n", execname(), pid(), filename)
}
```

Sample Script 2 contd

```
# Exit after 4 seconds
probe timer.ms(4000) { exit () }

probe end {
        print("Exiting !\n")
}
```

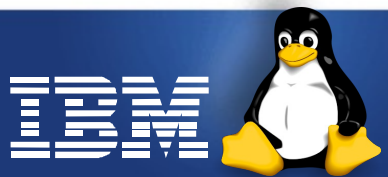


Sample Script 3

```
#! /usr/bin/env stap
# probe sys_open
# Trace open calls.

global entry_opens
probe begin {
        print("Logging Information\n")
}
probe kernel.function("sys_open")
{
        filename = user_string($filename)
        printf("%s(%d) open (%s)\n", execname(), pid(), filename)
        t=gettimeofday_us(); p=pid();
        entry_opens[p] = t;
}
```

Sample Script 3 Contd

```
probe kernel.function("sys_open").return
{
        t=gettimeofday_us(); p=pid();
        printf (" time taken =%d \n", t - entry_opens[p]);
}

# Exit after 4 seconds
probe timer.ms(4000) { exit () }

probe end {
        print("Exiting !\n")
}
```



Sample Script 4

```
#! /usr/bin/env stap
#
# Trace entry and exit of all functions defined in
kernel/sched.c.
#
probe begin {
  printf ("Collecting data... Type Ctrl-C to exit\n")
}
probe kernel.function("*@kernel/sched.c").call {
  printf ("%s -> %s\n", thread_indent(1), probefunc())
}
probe kernel.function("*@kernel/sched.c").return {
  printf ("%s <- %s\n", thread_indent(-1), probefunc())
}
```