Locating System Problems Using Dynamic Instrumentation

Vara Prasad, Jim Keniston

IBM

William Cohen, Frank Ch. Eigler, Martin Hunt

Red Hat, Inc.

Brad Chen

Intel

Abstract

It is often difficult to diagnose complex problems without multiple rebuilds and reboots. Even in a simple setup, the problem can touch various layers of the application and operating system. Diagnosis is even more difficult in complex, multi-tiered systems. As Linux is deployed in these environments, it is becoming more important to have facilities to locate and identify such problems.

Using the kprobes infrastructure, SystemTap is being developed to dynamically instrument the kernel and user applications. SystemTap instrumentation incurs low overhead when enabled, and zero overhead when disabled. SystemTap provides facilities to define instrumentation points in a high-level language, and to aggregate and analyze the instrumentation data. Details of the SystemTap architecture and implementation are presented, along with examples of solving problems in the production environments.

Agenda

- Introduction
- Kprobes
- Safety
- Architecture
- Details of the system
- Examples
- Status of the project
- Conclusions

Problem Definition

- Kernel Developer: I wish I could add a debug statement easily without going through the compile/build cycle.
- Technical Support: How can I get this additional data that is already available in the kernel easily and safely?
- Application Developer: How can I improve the performance of my application on Linux?
- System Admin: Occasionally jobs take significantly longer than usual to complete, or do not complete. Why?

Current Tools

- Examples: ps, netstat, vmstat, iostat, sar, strace, top, ltrace, oprofile, /proc, LTT, etc.
- Drawbacks:
  - Narrow focus, hence not suitable for system scope
  - Not flexible and configurable
  - Many different tools and data sources but no easy way to integrate the information
  - Overhead even when not in use

Motivation

- Ease of use: Provide an easy mechanism for dynamic instrumentation
- Re-use: An instrumentation library for common tasks
- Empower end users: A scripting language to get an insight into the system
- Infrastructure: A platform for debugging and analysis
- Top to bottom: Provide a tool that helps to solve problems from application layer to the h/w interface

SystemTap

- A tool to take a deeper look into a running system:
  - Provides insight into system operation
  - Assists in identifying causes of performance problems
  - Simplifies building instrumentation
- Started January 2005
- Open Source project
- Active contributions from Red Hat, Intel, IBM and other individual developers

Kprobes

- Kprobes is the foundation for SystemTap
- Probe Point: An address in the kernel for instrumentation
- Probe Handler: An instrumentation routine
- Replace the instruction at the probe points with a breakpoint intstruction
- When the breakpoint is hit, excecute the probe handler
- Execute the original instruction
- Continue at the next instruction following the breakpoint

Kprobes Enhancements

- Jumper probes (jprobes):
  - Useful in tracing function call entries
  - Give access to function call arguments
- Return probes (Kretprobes):
  - Facilitates tracing function return calls
  - Fire when function returns
- Multiple probe handlers at a probe point:
  - Allows different types of instrumentations
- Reentrant probes
- Scalability enhancements

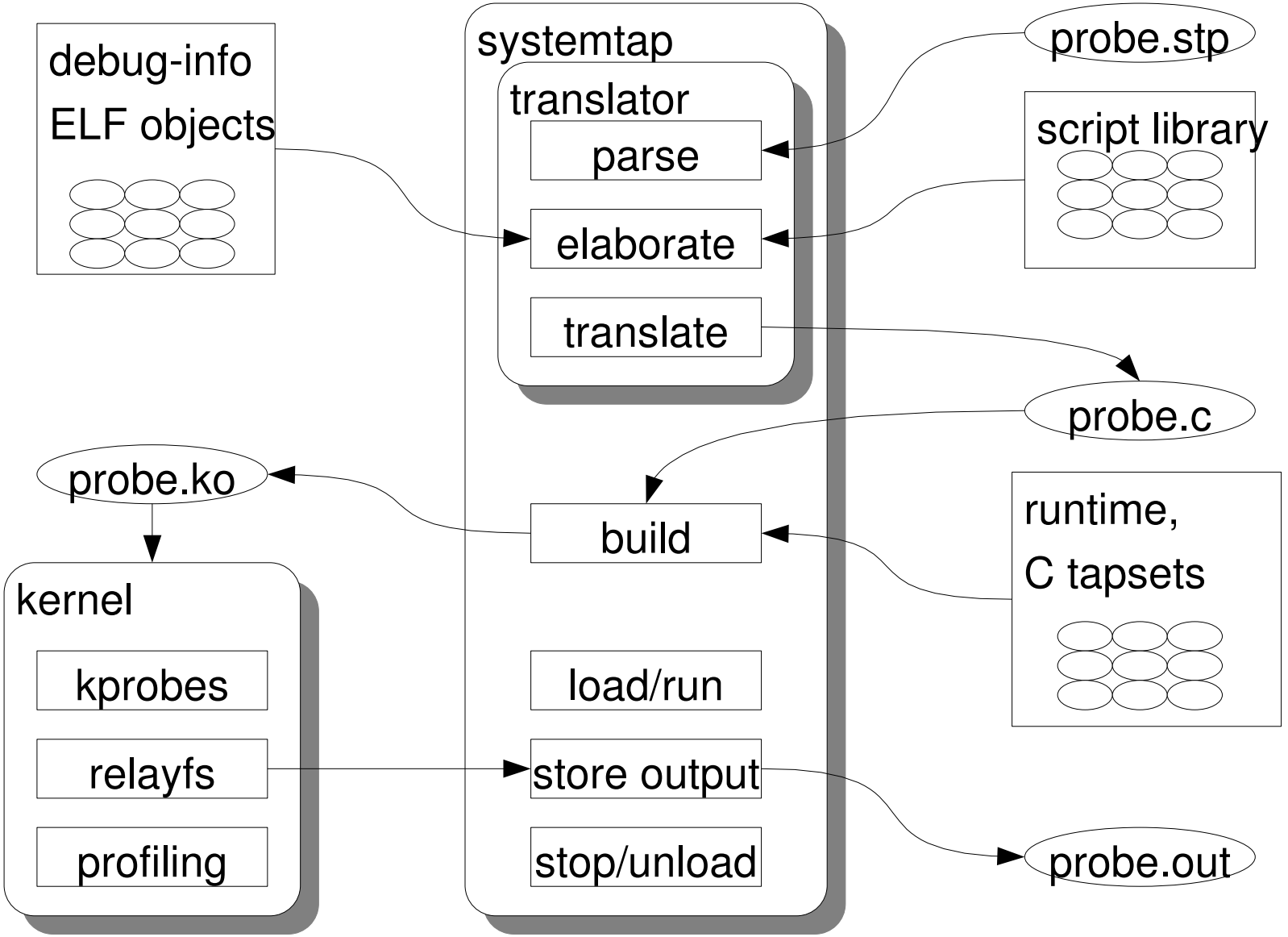
Kprobes Limitations

- No checking that probe point is at instruction boundary
- Kprobes-based code is hard to maintain and port due to hard coding of addresses
- No library of probes for common tasks
- No convenient access to local variables
- Requires significant kernel knowledge

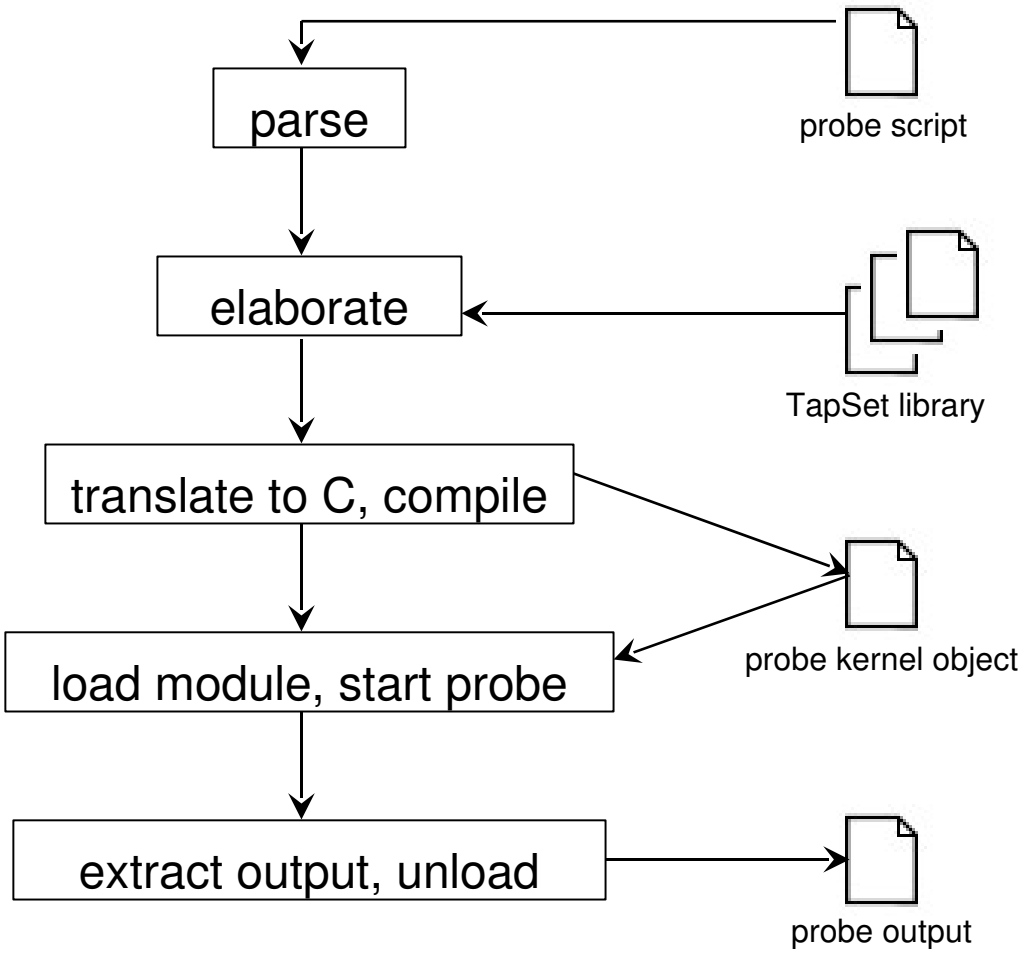
SystemTap Safety Goals

- For use in production environment – Crash proof
- Leverage existing tool chain
- Leverage existing kernel code
- Safe mode: Restricted functionality for production environments
- Guru mode: Full feature set for development environment
- Static analyzer:
  - Protection against translator bugs and user errors
  - Detects illegal instructions and external references

SystemTap Overall Diagram



Instrumentation Generation



redhat

Probe Scripting Language

- awk-like scripting language
- Simplified C-like syntax
- Limited number of types:
  - 64-bit numbers, strings, associative arrays, statistics
- Auxiliary functions
- Structured control statements, e.g. `if-then-else` and `while loops`
- Safety features:
  - No dynamic memory allocation
  - No assembly or arbitrary C code
  - Types and type conversions limited
  - Limited pointer operations

redhat

Elaboration Phase

- Takes a user probe script and:
  - Preprocesses macros
  - Includes required script libraries
  - Resolves references to symbols in code and instrumentation support libraries
- Uses DWARF2/3 debugging information to find:
  - Function entry location
  - Line number information
  - Global and local variables types and locations

redhat

Translation Phase

- Occurs after elaboration
- Each script subroutine expanded to block of C
- Generates calls into runtime library where needed
- Produces code to insert and remove instrumentation
- Generates .c file to be compiled into a kernel module

redhat

Runtime Library

- Provides:
  - Associative arrays (maps)
  - Per-cpu data types such as strings and counters
  - Statistics
  - Stack trace, register dump, symbol lookup
  - Safe copy from userspace
  - Output formatting
  - Assists with kernel-to-user-space transport
- Could be used by C programmers to simplify writing kprobes- and jprobes-based instrumentation

redhat

Build and Load Phase

- The “.c” file generated in the translation phase is built into a probe module in the build phase
- Module is loaded into the kernel
- Compiler and runtime safety checks:
  - Infinite loops and recursion
  - Invalid variable access
  - Division by zero
  - Restricted access to kernel memory
  - Array bound checks
  - Version compatibility checks

Data Handling Phase

- Data Collection – *Kernel Space*
  - During kernel execution, probes get activated
  - Data collection stops when module gets unloaded
- Data Preprocessing – *Kernel Space*
  - Simple Aggregation functions
- Data Transfer – *Kernel to User Space*
  - Relayfs: Efficient and low overhead mechanism for data transfer with per cpu buffering.
  - Netlink for control channel
  - Data generated by the probes is tagged for post processing
- Data Presentation – *User Space*
  - Data is ordered and merged
  - Merged data is formatted and presented to user
  - Optionally raw data can be streamed for other performance analysis tools

Example End User Script

```
global avg(reads);


probe kernel.syscall("read") {
  reads[$pname] <<< byte_count;
}


probe end {
  print(reads,
    "reads by process \"%1s\": %C.  Total bytes=%S.  Average: %A"));
}
```

How did we get the **byte_count** variable in the above user script?

TapSets

- A TapSet defines:
  - Probe Points: a set of instrumentation points for a particular subsystem
  - Data values that are available at each probe point.
- Written by developers knowledgeable in the given area
- Tested and packaged with SystemTap
- TapSets are written using probe scripting language and C.
- Data values are exported by writing Handler Statement Blocks (HSB)
- HSB's are used in generating the Kprobe handler code when needed

TapSet: Probe Point

- Probe points can be defined for kernel code locations in various forms
  - `kernel.function("sys_read").return`
  - `kernel.function("context_switch")`
- Additional probe point definitions can be defined for:
  - asynchronous events
    - `perfcounter("tlbmiss").count(4000)`
  - watchpoints

TapSet: Data Values

- Data values can be exported by writing HSB in SystemTap scripting language.

```
export kernel.syscall("read") =
kernel.function("sys_read")
{
  file_descriptor = $fd;
  byte_count = $count;
  filename = get_filename_from_fd(fd);
}
```

- `file_descriptor`, `byte_count`, `filename` are data values that can be used by an end user script
- `kernel.syscall("read")` is defined an alias for the `sys_read()` probe point
- `$fd` and `$count` are arguments of the `sys_read()` function

TapSet: Data Values (Contd.)

```
export kernel.syscall("read") =
kernel.function("sys_read")
{
  file_descriptor = $fd;
  byte_count = $count;
  filename = get_filename_from_fd(fd);
}
```

- Handler Statement Block can also make calls to “C” functions e.g. `get_filename_from_fd()`
- Used for locking, or when calls to several other kernel functions are needed

redhat

User Script: File Operations Statistics

```
global opens;
global avg(reads);
global avg(writes);

probe kernel.syscall("open") {
  open[$pname] += 1;
}

probe kernel.syscall("read") {
  reads[$pname] <<< byte_count;
}

probe kernel.syscall("write") {
  write[$pname] <<< byte_count;
}

probe end {
  print(opens, "%d opens by process \"%1s\"");
  print(reads,
    "reads by process \"%1s\": %C.  Total bytes=%S.  Average: %A"));
  print(writes,
    "writes by process \"%1s\": %C.  Total bytes=%S.  Average: %A");
}
```

redhat

Probe Output: File Operations Statistics

```
$ ./stp iotask.stp
Press Control-C to stop.
...
3459 opens by process "soffice.bin"
...
reads by process "sshd": 1887.  Total bytes=30916608.  Average:
16384
...
reads by process "ooffice": 7.  Total bytes=6799.  Average: 971
reads by process "soffice": 34.  Total bytes=90983.  Average: 2675
...
reads by process "soffice.bin": 4693.  Total bytes=7454020.
Average: 1588
...
writes by process "sshd": 1879.  Total bytes=1002914.  Average: 533
...
writes by process "soffice.bin": 24451.  Total bytes=518726.
Average: 21
...
```

redhat

User Script: smp_call_function

```
global traces

probe kernel.function("smp_call_function")
{
  traces[$pid, $pname, stack()] += 1;
}

probe end {
  print(traces);
}
```

redhat

Probe Output: smp_call_function

```
root# stp scf.stp
Press Control-C to stop.
All kprobes removed
traces[4010, hald, trace for 4010 (hald)
0xffffffff8011a551 : smp_call_function+0x1/0x70
0xffffffff80182c0c : invalidate_bdev+0x1c/0x40
0xffffffff8019bc48 : __invalidate_device+0x58/0x70
0xffffffff80188f89 : check_disk_change+0x39/0xa0
0xffffffff80133c90 : default_wake_function+0x0/0x10
0xffffffff802abeef : cdrom_open+0xa0f/0xa60
0xffffffff80133c90 : default_wake_function+0x0/0x10
0xffffffff80132650 : finish_task_switch+0x40/0x90
0xffffffff80346bb9 : thread_return+0x54/0x8b
0xffffffff801419cd : __mod_timer+0x13d/0x150
] = 18
...
```

Contributions

- Kprobe (x86, x86_64, ppc64) kernel support – IBM
- Kprobe (ia64) – Intel
- Relayfs – IBM
- Safety – Intel
- Performance monitoring TapSet - Intel
- Translator – Red Hat
- Runtime – Red Hat
- Testing - All

SystemTap Status

- Working to incorporate into RHEL4-U2 Fall 2005
- Kprobes: patches in mainline kernel (ia32, ia64, ppc64, x86_64)
- Relayfs: patches in -mm tree
- Translator/Runtime: 80% complete
- Libdw, libelf: available as part of elfutils
- Tapsets: looking at specific kernel areas: system calls, time keeping functions, virtual filesystem layer, etc.

Future Work

- User-space probes
- Non-root use of the tools
- Instrumentation of various subsystems by experts
- Visualization tools
- Continuous tracing or flight recording
- Automate monitoring of the system to detect potential performance degradation

Further Information

- Website: http://sources.redhat.com/systemtap
- Mailing list: systemtap@sources.redhat.com

Legally Speaking

- This work represents the view of the authors and does not necessarily represent the view of their employers.
- IBM is a registered trademark of International Business Machines Corporation in the United States and/or Other Countries. Red Hat is a registered trade mark of Red Hat Inc. Intel is a registered trade mark of Intel Corporation
- Other company, product, and service names may be trademarks or service marks of others.

Backup slides

Basic Kprobes Mechanism

Disassembly of do_fork()->kernel/fork.c

<do_fork>:

```
push   %ebp
push   %edi
mov    %eax,%edi
push   %esi
xor    %esi,%esi
push   %ebx
sub    $0x24,%esp
mov    %ecx,%ebx
mov    %edx,0x10(%esp)
call   1bf2 <do_fork+0x12>
mov    %eax,%ebp
test   %ebp,%ebp
mov    $0xfffffff5,%eax
js     1cc2 <do_fork+0xe2>
mov    $0xffff000,%eax
and    %esp,%eax
```

Basic Kprobes Mechanism

Disassembly of do_fork()->kernel/fork.c

<do_fork>:

```
push   %ebp
push   %edi
mov    %eax,%edi
push   %esi
xor    %esi,%esi
push   %ebx
sub    $0x24,%esp
mov    %ecx,%ebx
mov    %edx,0x10(%esp)
call   1bf2 <do_fork+0x12>
mov    %eax,%ebp
test   %ebp,%ebp
mov    $0xffffffff5,%eax
js     1cc2 <do_fork+0xe2>
mov    $0xfffff000,%eax
and    %esp,%eax
```

Copy the original instruction

```
Push %ebp
```

Basic Kprobes Mechanism

Disassembly of do_fork()->kernel/fork.c

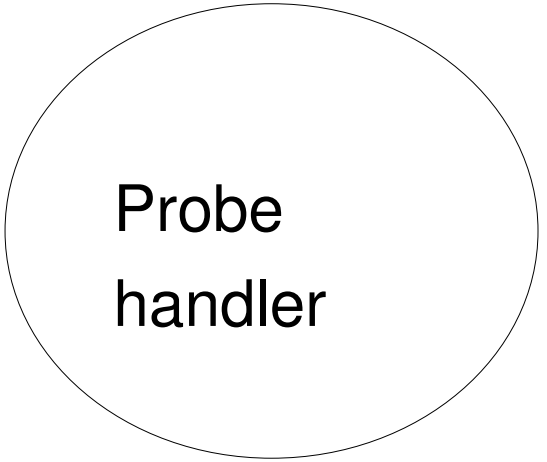
```
<do_fork>:
```

Int3/breakpoint

```
push   %edi
mov    %eax,%edi
push   %esi
xor    %esi,%esi
push   %ebx
sub    $0x24,%esp
mov    %ecx,%ebx
mov    %edx,0x10(%esp)
call   1bf2 <do_fork+0x12>
mov    %eax,%ebp
test   %ebp,%ebp
mov    $0xffffffff5,%eax
js     1cc2 <do_fork+0xe2>
mov    $0xfffff000,%eax
and    %esp,%eax
```

Copy of original instruction

```
Push  %ebp
```



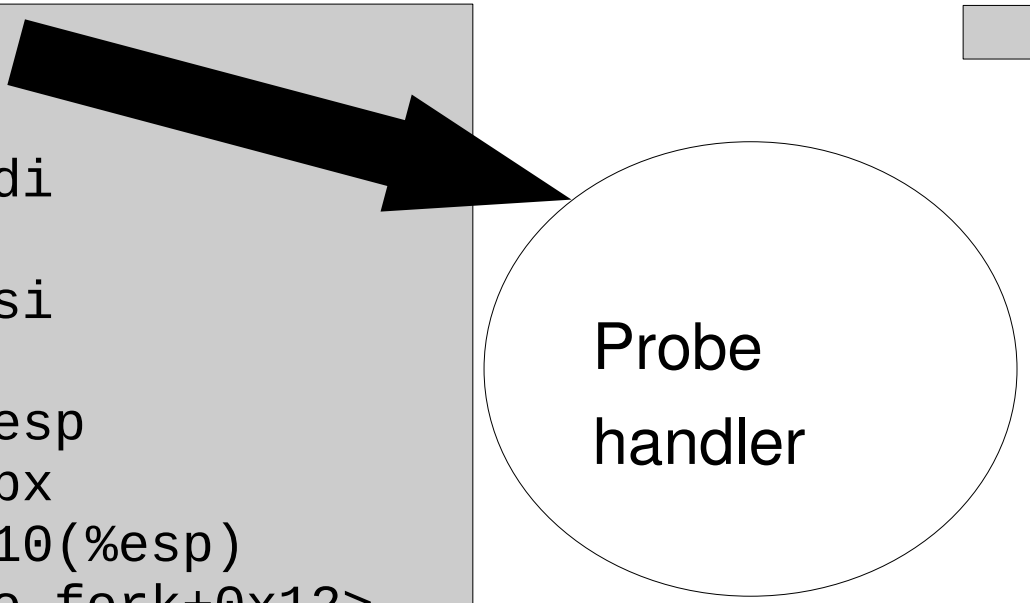
Basic Kprobes Mechanism

Disassembly of do_fork()->kernel/fork.c

```
<do_fork>:
Int3/breakpoint
push   %edi
mov    %eax,%edi
push   %esi
xor    %esi,%esi
push   %ebx
sub    $0x24,%esp
mov    %ecx,%ebx
mov    %edx,0x10(%esp)
call   1bf2 <do_fork+0x12>
mov    %eax,%ebp
test   %ebp,%ebp
mov    $0xffffffff5,%eax
js     1cc2 <do_fork+0xe2>
mov    $0xfffff000,%eax
and    %esp,%eax
```

Copy of original instruction

```
Push  %ebp
```



Basic Kprobes Mechanism

Disassembly of do_fork()->kernel/fork.c

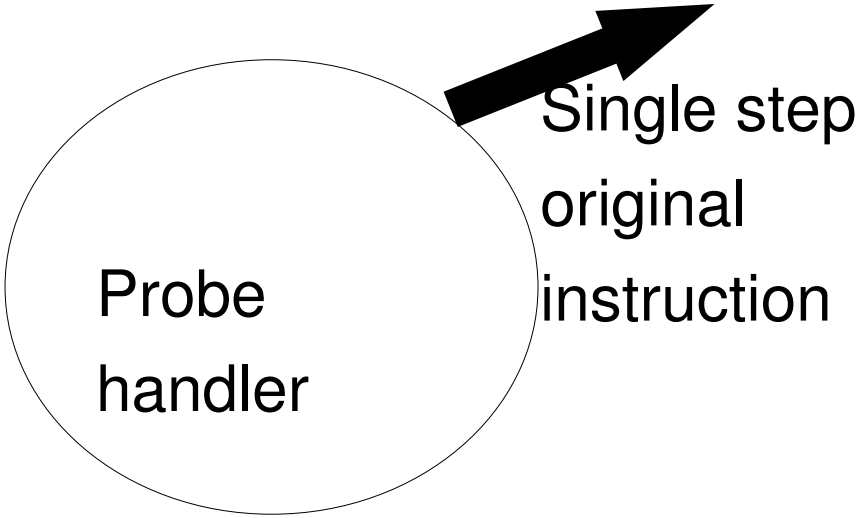
<do_fork>:

Int3/breakpoint

```
push   %edi
mov    %eax,%edi
push   %esi
xor    %esi,%esi
push   %ebx
sub    $0x24,%esp
mov    %ecx,%ebx
mov    %edx,0x10(%esp)
call   1bf2 <do_fork+0x12>
mov    %eax,%ebp
test   %ebp,%ebp
mov    $0xffffffff5,%eax
js     1cc2 <do_fork+0xe2>
mov    $0xfffff000,%eax
and    %esp,%eax
```

Copy of original instruction

```
Push  %ebp
```



Basic Kprobes Mechanism

Disassembly of do_fork()->kernel/fork.c

<do_fork>:

Int3/breakpoint

```
push   %edi
mov    %eax,%edi
push   %esi
xor    %esi,%esi
push   %ebx
sub    $0x24,%esp
mov    %ecx,%ebx
mov    %edx,0x10(%esp)
call   1b <do_fork+0x12>
mov    %eax,%ebp
test   %ebp,%ebp
mov    $0xffffffff5,%eax
js     1cc2 <do_fork+0xe2>
mov    $0xffff000,%eax
and    %esp,%eax
```

Copy of original instruction

Push %ebp

Probe handler

Continue executing next instruction