Universal Function Call Tracing

Olaf Dabrunz

odabrunz@fctrace.org

Why Function Call Tracing?

- Quickly provides information about program execution
- Software integration (distributions, ISVs)
- Technical support
- Contributing developers
- Testing (coverage, QA, beta testing)
- Optimization (profiling)
- Software documentation
- Debugging aid: shows actual program behaviour
- Security audit (code analysis, esp. of modularized software)
- · etc...

Example: Fixing Software Bugs

- A customer has data corruption in his database app
- Simple testcases do not reproduce the corruption
- Bug in the application, the database or the kernel?
- Traditional code review takes time: huge number of functions, which are actually used?

3

Fixing Software Bugs with Tracing

- Tracing can show the participating functions, possibly with parameter values
 - Follow execution path (maybe with data) through functions
 - Easily find the used plugins, registered functions etc.
 - Run further tests, maybe follow the code step-by-step while watching the trace

Example: Security Analysis

- Review security problems in an open source app
- Problematic use of userspace data in some function?
- Need to read and follow the code as the data is passed through many functions

Security Analysis with Tracing

- · Run testcase against the code while tracing it
- The trace will show the function calls and parameters
- Often sufficient to follow data across many function calls
- No need to follow data manually
- Could even be used to test for misuse scenarios
- Also could help checking coverage and correctness of code annotations for source code checkers (such as splint)

Example: Technical Support

- Customer has a problem
- Support does not have the hardware or the configuration to reproduce the problem
- Customer provides a kernel stack trace or a crash dump
- A stack trace or crash dump can only show the state when the problem is detected; if the problem was caused earlier on, there is no information about that
- Support and development try to find the root cause by asking the customer to run test cases until the root cause is isolated

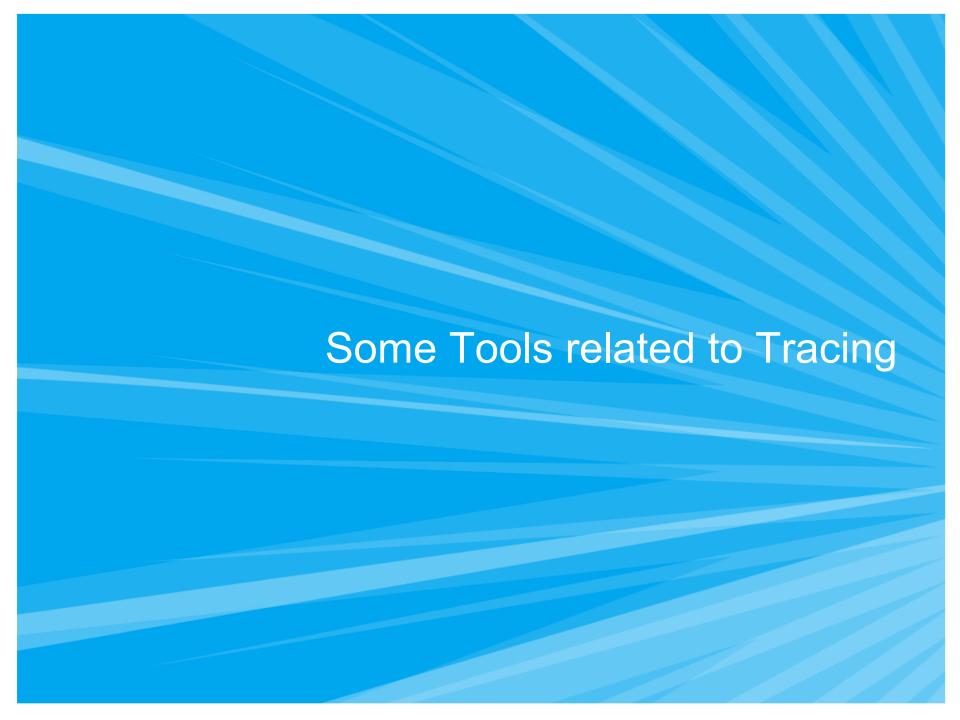
Technical Support uses Tracing

- Support can ask the customer to trace the problematic process(es)
- The function call history (with parameter values) may show where the root cause is
- Even when testcases need to be run, a trace during the testcase can generate more information, so that probably less testcases need to be run

Universal Function Call Tracing

Universal Function Call Tracing

- Always be available, reliably
 - On all hardware platforms
 - With all kernel versions
 - For all programs
- No setup is required (such as compiled-in instrumentation)
- Simply start trace and look at the results (like strace)
- Show all function calls
- One-stop solution: cover as many use cases as possible with a simple mechanism
- Least possible overall slowdown even when multithreading



Some Tools related to Tracing

- · strace, Itrace
- · Perf, Oprofile
- · UST
- Valgrind
- · GDB
- LTTng
- DTrace
- Systemtap
- ftrace
- fctrace

Userspace

Kernelspace

strace and Itrace

strace

- Trace system calls of one or more processes
- Uses specific facility for system call traces ptrace(PTRACE_SYSCALL)
- ptrace() is slow: it requires context switches from the tracer (userspace) to the kernel to the traced process (userspace) and back for every action

Itrace

- Trace library and system calls of one or more processes
- Hooks the shared library linking mechanism
- May miss library function calls when they are called differently
- Cannot trace internal functions

Perf, Oprofile

- Sample execution of kernel functions each time a hardware event fires, e.g. high-resolution timer (TSC)
- Perf can also use tracepoints as event sources
- Gather statistics: how much time spent in which function
- Does not "follow" process execution: not always clear when a function is called
 - Stack analysis helps to find this out, but uses more processing time and fails for tail call optimizations
- May miss called functions, when called and left within sampling period

Valgrind

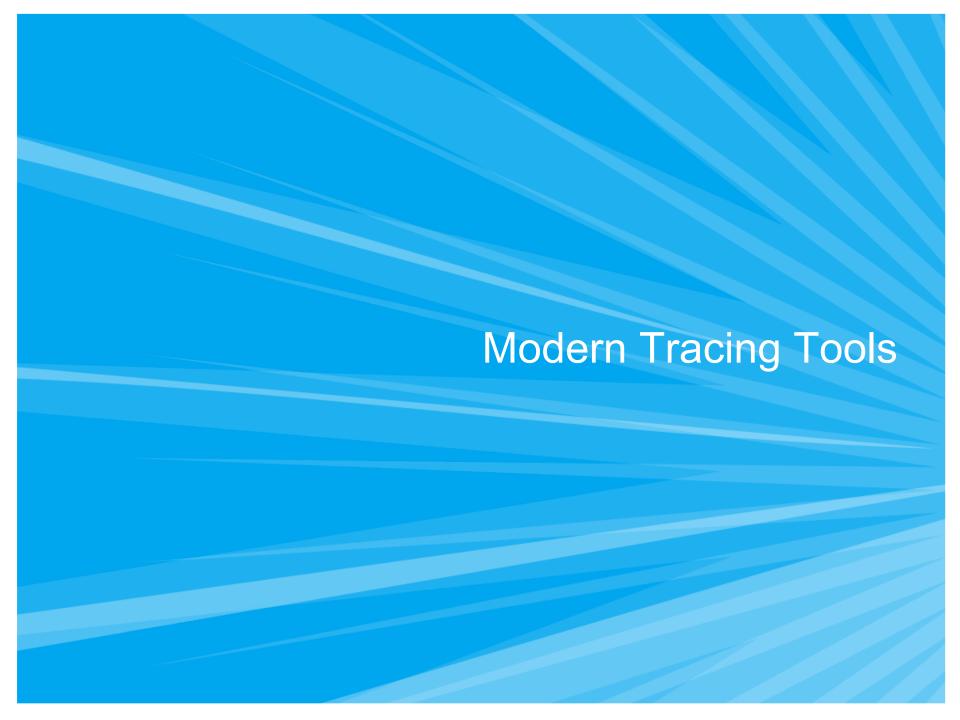
- Userspace simulator executing userspace programs
- Follows variable usage and mis-usage
- Checks library calls and mis-usage, esp. for memory allocations
- No working trace module so far
- Not quick enough for programs in production use

GDB

- Breakpoints
- Watchpoints
- Macros
- No built in function call tracing

Linux Trace Toolkit

- Instrument by patching source code
- Patch inserts calls in several kernel functions
- Cannot be disabled
- · 3% 4% slowdown when LTT is **un**used
- Kernel changes quickly: maintenance of instrumentation patch is work-intensive
- Not targeted at tracing all functions



Modern Tracing Tools

- · strace, Itrace
- · Perf, Oprofile
- · UST
- Valgrind
- · GDB
- · LTTng
- DTrace
- Systemtap
- · ftrace
- · fctrace

Userspace

Kernelspace

Linux Trace Toolkit next generation

- Use intrumentation with "Kernel Markers"
- "Kernel Markers" are special instruction sequences
 - » a load from a direct address, test, and a conditional branch over a call sequence
- Instrumentation is part of kernel code and compiled in
- Can enable and disable instrumentation by changing "Immediate Value" in instruction sequence
- Runtime overhead small when disabled
- Developers are required to instrument their functions with standard kernel markers
- Not targeted at instrumenting all function calls, but to gather information from "points of interest"
- UST does the same for userspace

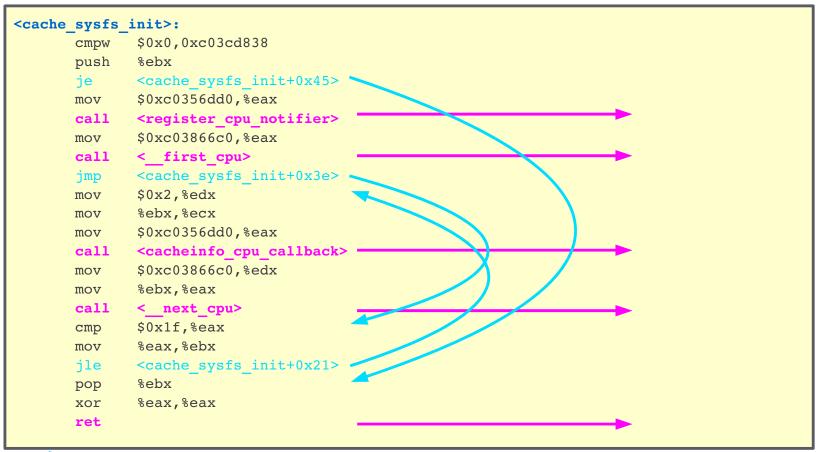
DTrace / Systemtap

- Use intrumentation with breakpoints (on x86: INT3)
- Instrumentation added by overwriting opcode
- Can disable instrumentation by restoring opcode
- Original instruction is copied and single-stepped when breakpoint triggers
- Instrument all functions (limited set of functions possible, but not for complete trace)

Example Function

```
<cache sysfs init>:
      cmpw
            $0x0,0xc03cd838
      push
            %ebx
      je <cache sysfs init+0x45>
      mov $0xc0356dd0, %eax
      call <register cpu notifier>
      mov $0xc03866c0, %eax
      call < first_cpu>
      jmp <cache sysfs init+0x3e>
      mov $0x2, %edx
      mov %ebx, %ecx
      mov $0xc0356dd0, %eax
      call <cacheinfo cpu callback>
      mov $0xc03866c0, %edx
      mov %ebx, %eax
      call < next cpu>
      cmp $0x1f,%eax
      mov %eax, %ebx
      jle
          <cache sysfs init+0x21>
           %ebx
      pop
           %eax,%eax
      xor
      ret
```

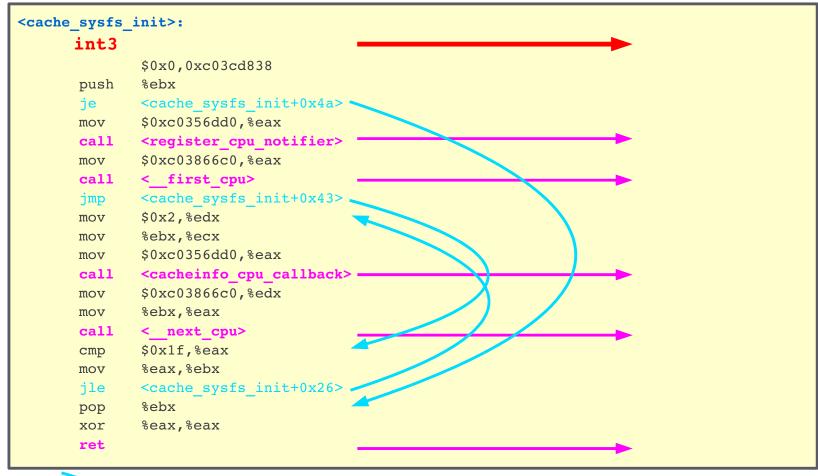
Example Function with Annotations





jumps/branches within the function jumps/branches to other functions

DTrace / Systemtap Instrumentation





jumps/branches within the function jumps/branches to other functions

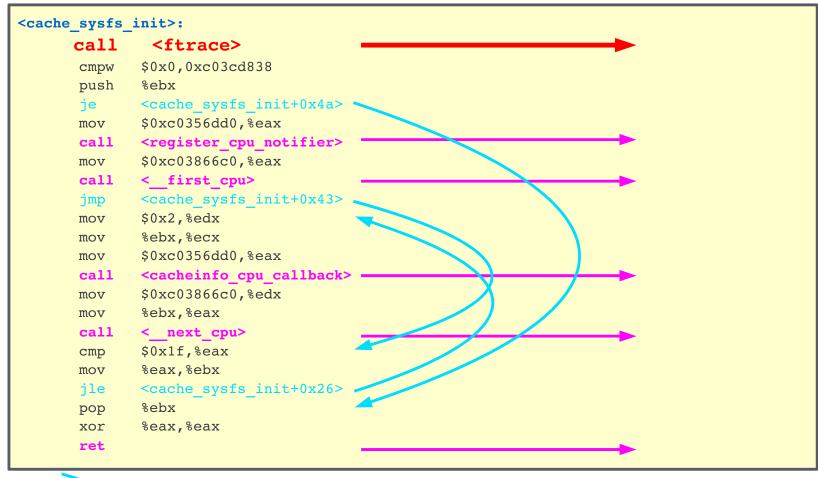
Call Tracing with DTrace / Systemtap

- Complete function call trace slows down system
- When Dtrace was new we tested a system with probes at the beginning of every function and the system slowed down to virtual halt
- Approach unusable for complete call trace
- So should we piece together a call trace?
- Many selective call traces (each with a small footprint)
 need to be run to cover the whole call chain
- Reproducing the same call chain can be an issue, especially when trying to reproduce a bug

ftrace

- Uses profiling instrumentation
- Instrumentation added by compilation with "gcc -pg"
- Can disable instrumentation by overwriting with NOPs
- Instruments all functions (can limit, but not for complete trace)

ftrace Instrumentation





jumps/branches within the function jumps/branches to other functions

ftrace Instrumentation Deactivated

```
<cache sysfs init>:
       nop
       nop
       nop
       nop
       nop
             $0x0,0xc03cd838
      cmpw
      push
             %ebx
      jе
             <cache sysfs init+0x4a>
      mov $0xc0356dd0, %eax
      call <register cpu notifier>
             $0xc03866c0, %eax
      mov
      call
            < first cpu>
             <cache sysfs init+0x43>
       qmj
             $0x2, %edx
      mov
           %ebx,%ecx
      mov
             $0xc0356dd0, %eax
      mov
      call
             <cacheinfo cpu callback>
             $0xc03866c0, %edx
      mov
             %ebx, %eax
      mov
      call
             < next cpu>
             $0x1f,%eax
      cmp
             %eax,%ebx
      mov
```



jumps/branches within the function

jumps/branches to **other** functions

Call Tracing with ftrace

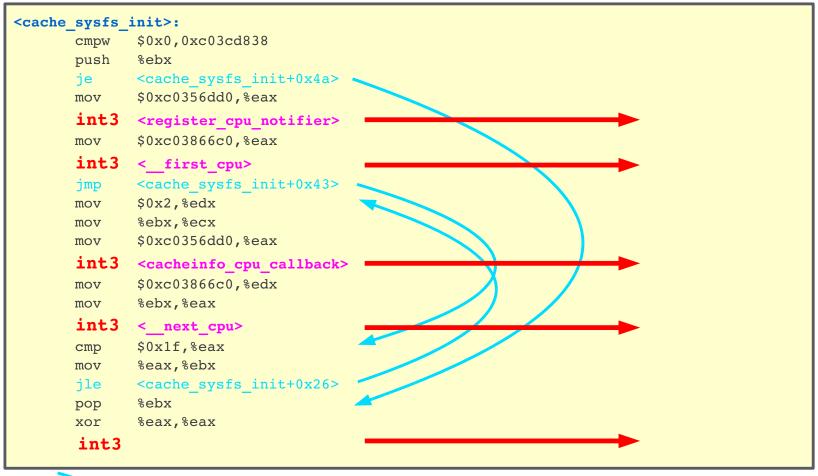
- Complete function call trace causes overall system slowdown
- "Just calling mcount() and having mcount() return has shown a 10% overhead." Steven Rosted
- Actual overhead with real trace code is much higher
- Inline functions are not instrumented

 Piecing together a complete trace from selective call traces has the same issues as for DTrace / Systemtap

fctrace

- Use intrumentation with breakpoints (on x86: INT3)
- Instrumentation added by overwriting opcode
- Can disable instrumentation by restoring opcode
- Original instruction is copied and single-stepped when breakpoint triggers
- Instrumenting a code location is atomic: no expensive synchronization is needed (only light-weight locking for meta-data structures)
- Instrument only the function that the traced process currently executes

fctrace Instrumentation





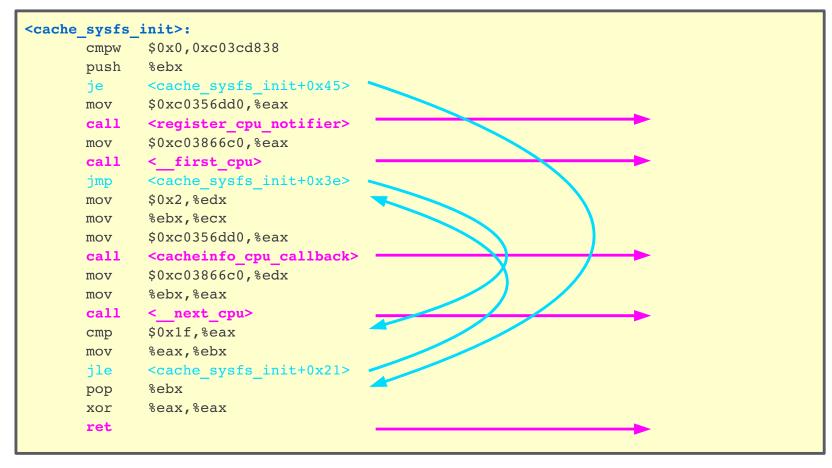
jumps/branches within the function jumps/branches to other functions



fctrace Single Stepping through a Call

```
<cache sysfs init>:
      cmpw
             $0x0,0xc03cd838
      push
             %ebx
      je <cache sysfs init+0x4a>
      mov $0xc0356dd0, %eax
      int3 <register cpu notifier>
             $0xc03866c0, %eax
      mov
      int3 < first cpu>
      jmp <cache sysfs init+0x43>
          $0x2,%edx
      mov
          %ebx,%ecx
      mov
             $0xc0356dd0, %eax
      mov
      int3 <cacheinfo cpu callback>
             $0xc03866c0, %edx
      mov
             %ebx, %eax
      mov
      int3 < next_cpu>
      cmp $0x1f, %eax
      mov %eax, %ebx
          <cache sysfs init+0x26>
      jle
           %ebx
      pop
             %eax, %eax
      xor
      int3
```

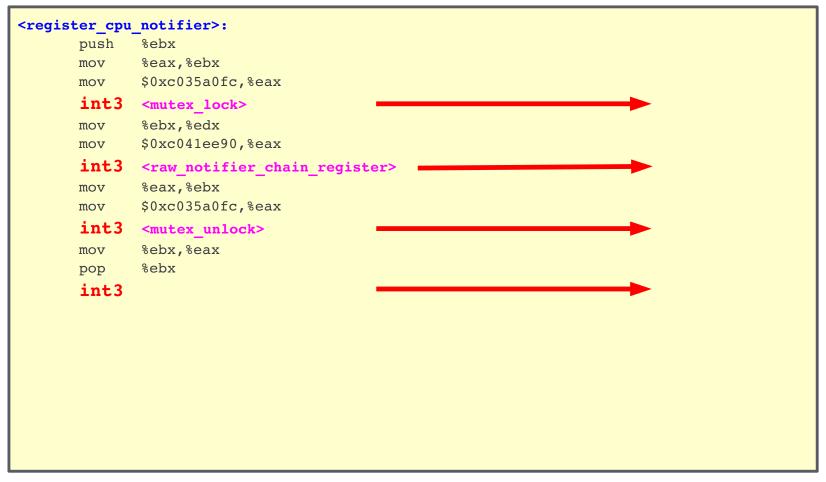
fctrace Leaving a Function





jumps/branches within the function jumps/branches to other functions

fctrace Entering the Next Function



Call Tracing with fctrace

- Complete function call trace does not cause overall system slowdown
 - Other tasks will rarely execute the instrumented function
- The traced task executes the instrumented function:
 it will be slowed down
- No actual speed measurements for the traced task yet
- Speed optimizations for traced task possible
 - Lazy cleanup
 - Hardware support
- Tracing inline functions will be possible

Benefits of fctrace

- Instrumentation does not exist when off
- When on
 - No overall system slowdown
 - Slows down traced tasks only
- No special compilation or setup needed
- Available / portable to all architectures
- Portable to other operating systems
- As easy to use as strace
- Will trace function parameters

fctrace Status

- fctrace prototype exists
- fctrace initially used kprobes
- It worked as long as traced code does not take locks
- Kprobes does not support dynamic changes of probes while the traced code holds spinlocks
- Needed to write a dynamic version of kprobes: vprobes was started
 - pre-allocate memory for all needed probes
 - never schedule() during probe activation or deactivation

vprobes Status (1/2)

- Used kprobes as starting point
- Code has changed a lot
 - New memory management
 - New locking, but needs more work
 - Dropped features that fctrace will replace: e.g. jprobes
- Meanwhile kprobes changed a lot upstream
 - About 200 patches until end of 2009:
 - > Consolidation of 64 bit and 32 bit code
 - > Fixes, features and cleanups in the probe engine
 - Most patches are relevant to vprobes
 - → Needed to find a way to integrate kprobes patches in vprobes, and to develop vprobes alongside kprobes

vprobes Status (2/2)

- It proved too error-prone to integrate kprobes patches:
 - Most patches needed manual merging
 - Several patches needed to be analyzed to understand how they apply to vprobes, e.g. when related to locking or probe lifetime and re-use
 - Vprobes itself was not ready for testing, so integrated patches could not be tested either
 - Too many errors would go unnoticed
 - → A patch management tool is needed to track patches, and to connect upstream patches to vprobes patches so porting problems can later be found
 - I started working on improvements for Git and TopGit, and started writing a patch management frontend for vim that uses Git and TopGit

What's next?

- Develop vprobes alongside kprobes: current patch management tools are still insufficient
- Finish vprobes
 - finalize implementation of sped-up memory access checks
 - interfaces probably final
- Use vprobes in fctrace
 - delegate probe pool handling to vprobes
 - performance optimizations
- Show function call parameters
- Apply vprobes/fctrace mechanism to userspace

The Future: After The Prototype

Reducing Detail through filtering

- Complete call traces contain too much information
- fctrace can filter the traces
- The uninteresting information can incrementally be filtered out

Other tracing mechanisms

- Hardware breakpoints
- Intel Branch Trace mechanism

Hardware Breakpoints

- HW breakpoints are much quicker than modifying code
- But only few HW breakpoints are available
- Up to several tens of call sites need to be instrumented in the kernel – userspace programs may have more
- HW breakpoints are not available on some platforms
 - → Not a universal tracing mechanism

Intel Branch Trace Mechanism

- On Pentium 6: taken branches generate exception
- On Pentium 4: taken branches recorded on a stack
- Promises less overhead than INT3
- Does not know if branch leaves the function (function call) or not (loop, conditional, ...) -- this would require hints in the machine code
- All branches are recorded, CPU is often interrupted
- May perform much worse than INT3, esp. on inner loops
- · Not available on other platforms (PPC, s390, ARM, ...)

More information

- Project homepage http://fctrace.org/
- Author: Olaf Dabrunz <odabrunz@fctrace.org>

More information

- Project homepage http://fctrace.org/
- Author: Olaf Dabrunz <odabrunz@fctrace.org>

Questions?

DProbes

- Userspace package to compile probes
- Compiled probes are loaded into the kernel
- Kprobes infrastructure triggers execution of compiled dprobes

Kernelspace and Userspace

- Separate Memory Regions
- Kernel can access both Userspace and Kernelspace
- Userspace application can directly access only its own Userspace memory
- A process can execute in Userspace or in Kernelspace
- When a process enters or leaves the Kernel, a context switch is necessary

Example Function

c010d93a <cache_sysfs_init>:</cache_sysfs_init>								
c010d93a:	66	83	3d	38	d8 3c	c0	cmpw	\$0x0,0xc03cd838
c010d941:	00							
c010d942:	53						push	%ebx
c010d943:	74	3a					je	c010d97f <cache_sysfs_init+0x45></cache_sysfs_init+0x45>
c010d945:	b8	d0	6d	35	c0		mov	\$0xc0356dd0,%eax
c010d94a:	e8	0e	16	03	00		call	c013ef5d <register_cpu_notifier></register_cpu_notifier>
c010d94f:	b8	c0	66	38	c0		mov	\$0xc03866c0,%eax
c010d954:	e8	7f	fd	0b	00		call	c01cd6d8 <first_cpu></first_cpu>
c010d959:	eb	1d					jmp	c010d978 <cache_sysfs_init+0x3e></cache_sysfs_init+0x3e>
c010d95b:	ba	02	00	00	00		mov	\$0x2, %edx
c010d960:	89	d9					mov	%ebx,%ecx
c010d962:	b8	d0	6d	35	c0		mov	\$0xc0356dd0,%eax
c010d967:	e8	7c	fc	ff	ff		call	c010d5e8 <cacheinfo_cpu_callback></cacheinfo_cpu_callback>
c010d96c:	ba	c0	66	38	c0		mov	\$0xc03866c0, %edx
c010d971:	89	d8					mov	%ebx,%eax
c010d973:	e8	78	fd	0b	00		call	c01cd6f0 <next_cpu></next_cpu>
c010d978:	83	f8	1f				cmp	\$0x1f,%eax
c010d97b:	89	c3					mov	%eax,%ebx
c010d97d:	7e	dc					jle	c010d95b <cache_sysfs_init+0x21></cache_sysfs_init+0x21>
c010d97f:	5b						pop	%ebx
c010d980:	31	c0					xor	%eax, %eax
c010d982:	с3						ret	

Example Function

```
c010d93a <cache sysfs init>:
c010d93a:
               cmpw
                     $0x0,0xc03cd838
c010d942:
               push
                     %ebx
               jе
                     c010d97f <cache sysfs init+0x45>
c010d943:
              mov
                     $0xc0356dd0, %eax
c010d945:
                     c013ef5d <register cpu notifier>
c010d94a:
              call
c010d94f:
               mov
                     $0xc03866c0, %eax
               call
c010d954:
                     c01cd6d8 < first cpu>
c010d959:
                     c010d978 <cache sysfs init+0x3e>
               jmp
c010d95b:
               mov $0x2, %edx
c010d960:
               mov %ebx, %ecx
                     $0xc0356dd0, %eax
c010d962:
               mov
               call
                     c010d5e8 <cacheinfo cpu callback>
c010d967:
c010d96c:
                     $0xc03866c0, %edx
               mov
c010d971:
                     %ebx, %eax
               mov
c010d973:
               call
                     c01cd6f0 < next cpu>
c010d978:
                     $0x1f,%eax
               cmp
                     %eax,%ebx
c010d97b:
               mov
               jle
                     c010d95b <cache sysfs init+0x21>
c010d97d:
c010d97f:
               pop
                     %ebx
c010d980:
                     %eax,%eax
               xor
c010d982:
               ret.
```

Example Function with Annotations

```
c010d93a <cache sysfs init>:
                                                        <-- start of function
c010d93a:
               cmpw
                      $0x0,0xc03cd838
c010d942:
               push %ebx
              je
                                                       <-- branch within the current function
c010d943:
                     c010d97f <cache sysfs init+0x45>
c010d945:
              mov $0xc0356dd0, %eax
               call
                     c013ef5d <register cpu notifier>
                                                       <-- jump/call to other function
c010d94a:
c010d94f:
                      $0xc03866c0, %eax
               mov
               call
                      c01cd6d8 < first cpu> <-- jump/call to other function</pre>
c010d954:
                      c010d978 <cache sysfs init+0x3e> <-- branch within the current function
c010d959:
               dmj
                      $0x2, %edx
c010d95b:
               mov
c010d960:
                   %ebx,%ecx
               mov
c010d962:
                      $0xc0356dd0, %eax
               mov
                      c010d5e8 <cacheinfo cpu callback> <-- jump/call to other function
c010d967:
               call
                      $0xc03866c0, %edx
c010d96c:
               mov
c010d971:
                      %ebx,%eax
               mov
                      c01cd6f0 < next cpu>
                                                      <-- jump/call to other function
c010d973:
               call
c010d978:
                      $0x1f,%eax
               cmp
c010d97b:
               mov
                      %eax,%ebx
                      c010d95b <cache sysfs init+0x21> <-- branch within the current function
c010d97d:
               jle
c010d97f:
                      %ebx
               pop
                      %eax, %eax
c010d980:
               xor
c010d982:
                                                       <-- jump/call to other function
               ret
```