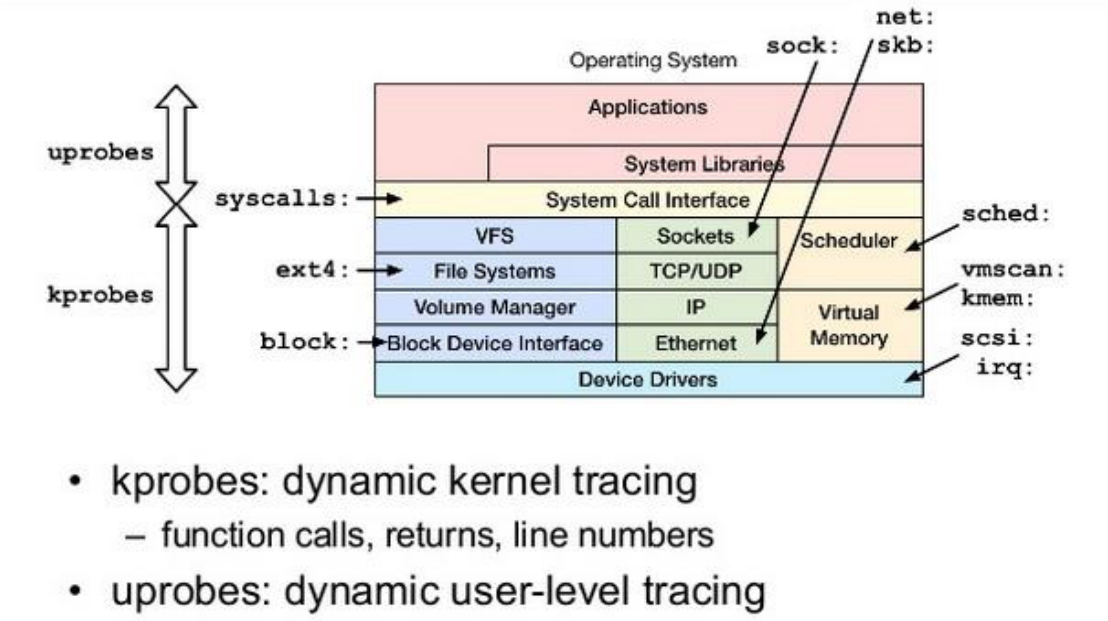
Uprobe 使用指南

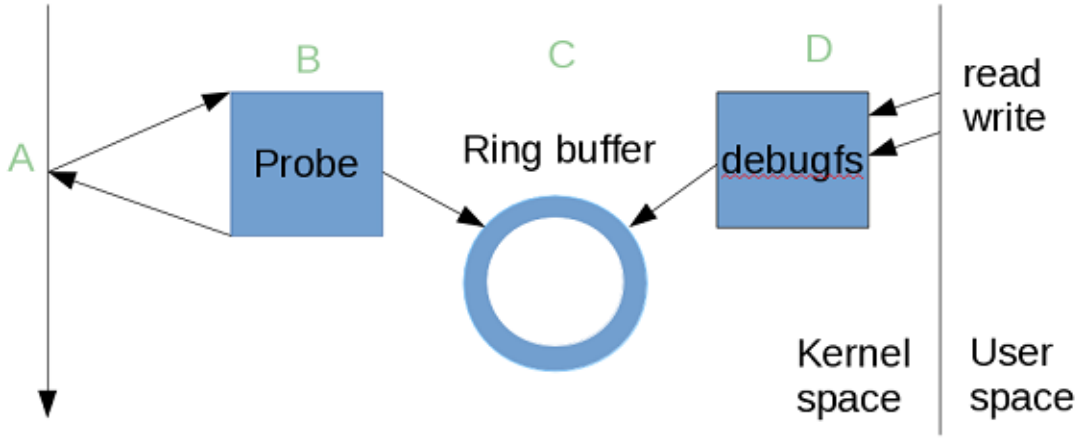
经过长期的发展 **kprobes/uprobes** 机制在事件(events)的基础上分别为内核态和用户态提供了追踪调试的功能，这也构成了 tracepoint 机制的基础，后期的很多工具，比如 **perf_events**, **ftrace** 等都是在其基础上演化而来. 参考由 **Brendan Gregg** 提供的资料来看, **kprobes/uprobes** 在 Linux 动态追踪层面起到了基石的作用, 如下所示:



- kprobes: dynamic kernel tracing
  - function calls, returns, line numbers
- uprobes: dynamic user-level tracing

引用: https://blog.arstercz.com/introduction_to_linux_dynamic_tracing/

Kprobe 和 Uprobe 均可以通过 ftrace 的 /sys/debug/tracing interface (基于 debugfs 的用户空间层面的 API) 执行各种跟踪和分析，虽然 **ftrace** 的内部是复杂的，不过输出的信息却以简单明了为主, 更详细的使用示例可以参考 **ftrace-lwn-365835**, 如下图所示，大致为 **ftrace** 的原理:



快速开始，测试代码准备:

```
#include <stdio.h>
#include <unistd.h>

static void
print_curr_state_one(void)
{
    printf("This is the print current state one function\n");
}


static void
print_curr_state_two(void)
{
    printf("This is the print current state two function\n");
}


int main() {
    while(1) {
        print_curr_state_one();
        sleep(1);
        print_curr_state_two();
    }
}
```

编译并获取可执行文件和反汇编:

csky-linux-gcc test.c -o test

csky-linux-objdump -S test > test.asm

```
➜  linux-next git:(linux-next-ftrace-kprobe-uprobe-simlutate-insn) X ../artifacts/ou
```

```
00008518 <print_curr_state_one>:
    8518:       1422            subi            sp, sp, 8
    851a:       dd0e2000        st.w            r8, (sp, 0)
    851e:       ddee2001        st.w            r15, (sp, 0x4)
    8522:       6e3b            mov             r8, sp
    8524:       1006            lrw             r0, 0x8698      // 853c <print_curr_st
    8526:       eae00007        jsri            0x0     // from address pool at 0x8540
    852a:       6c00            or              r0, r0
    852c:       6fa3            mov             sp, r8
    852e:       d9ee2001        ld.w            r15, (sp, 0x4)
    8532:       d90e2000        ld.w            r8, (sp, 0)
    8536:       1402            addi            sp, sp, 8
    8538:       783c            rts
    853a:       0000            .short  0x0000
    853c:       00008698        .long   0x00008698
    8540:       00000000        .long   0x00000000
00008544 <print_curr_state_two>:
    8544:       1422            subi            sp, sp, 8
    8546:       dd0e2000        st.w            r8, (sp, 0)
    854a:       ddee2001        st.w            r15, (sp, 0x4)
    854e:       6e3b            mov             r8, sp
    8550:       100d            lrw             r0, 0x86c8      // 8584 <main+0x1c>
    8552:       eae0000e        jsri            0x0     // from address pool at 0x8588
    8556:       6c00            or              r0, r0
    8558:       6fa3            mov             sp, r8
    855a:       d9ee2001        ld.w            r15, (sp, 0x4)
    855e:       d90e2000        ld.w            r8, (sp, 0)
    8562:       1402            addi            sp, sp, 8
    8564:       783c            rts
        ...
00008568 <main>:
    8568:       1422            subi            sp, sp, 8
    856a:       dd0e2000        st.w            r8, (sp, 0)
    856e:       ddee2001        st.w            r15, (sp, 0x4)
    8572:       6e3b            mov             r8, sp
    8574:       e3ffffd2        bsr             0x8518  // 8518 <print_curr_state_one>
    8578:       3001            movi            r0, 1
    857a:       eae00006        jsri            0x0     // from address pool at 0x8590
    857e:       e3ffffe3        bsr             0x8544  // 8544 <print_curr_state_two>
    8582:       07f9            br              0x8574  // 8574 <main+0xc>

➜  linux-next git:(linux-next-ftrace-kprobe-uprobe-simlutate-insn) ✗ ../artifacts/out
There are 28 section headers, starting at offset 0x1cd8:
Section Headers:
```

```
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0  0
[ 1] .interp           PROGBITS        00008134 000134 00000d 00   A  0   0  1
[ 2] .note.ABI-tag     NOTE            00008144 000144 000020 00   A  0   0  4
[ 3] .hash             HASH            00008164 000164 00003c 04   A  4   0  4
[ 4] .dynsym           DYNSYM          000081a0 0001a0 0000a0 10   A  5   1  4
[ 5] .dynstr           STRTAB          00008240 000240 000098 00   A  0   0  1
[ 6] .gnu.version      VERSYM          000082d8 0002d8 000014 02   A  4   0  2
[ 7] .gnu.version_r    VERNEED         000082ec 0002ec 000020 00   A  5   1  4
[ 8] .rela.dyn         RELA            0000830c 00030c 00009c 0c   A  4   0  4
[ 9] .init             PROGBITS        000083b0 0003b0 000030 00  AX  0   0 16
[10] .text             PROGBITS        000083e0 0003e0 000282 00  AX  0   0 16
```

根据程序反汇编编插 uprobe 桩

```
echo 'p:enter_current_state_one /root/test:0x518 arg0=%a0 lr=%lr' >> /sys/kernel/debug
echo 'r:exit_current_state_one /root/test:0x518 arg0=%a0' >> /sys/kernel/debug/tracing
echo 'p:enter_current_state_two /root/test:0x544 arg0=%a0 lr=%lr' >> /sys/kernel/debug
echo 'r:exit_current_state_two /root/test:0x544 arg0=%a0' >> /sys/kernel/debug/tracing
echo 1 > /sys/kernel/debug/tracing/events/uprobes/enable
cat /sys/kernel/debug/tracing/trace
```

用户态任意位置都可以设置 uprobe 桩点

除了 ftrace，我们还可以使用 Perf probe -x 动态跟踪点

是一种可观察性功能，可实现以下功能：甚至不需要重新编译就可以插装源代码的任意行。有了程序源代码的副本，跟踪点可以在运行时放置在任何地方，并且每个变量的值都可以转储时间执行通过跟踪点。 这是一项非常强大的技术，用于检测其他人编写的复杂系统或代码。

```
perf probe -x /lib/libc.so.6 memcpy
perf record -e probe_libc:memcpy -aR ls
perf report
```

社区有大量 Uprobe 应用案例，和文档:

http://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html

https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt

https://www.kernel.org/doc/html/latest/trace/uprobetracer.html