Debugging Tools for OpenAFS Linux Cache Manager

Yadavendra Yadav
Todd DeSantis

IBM

Agenda

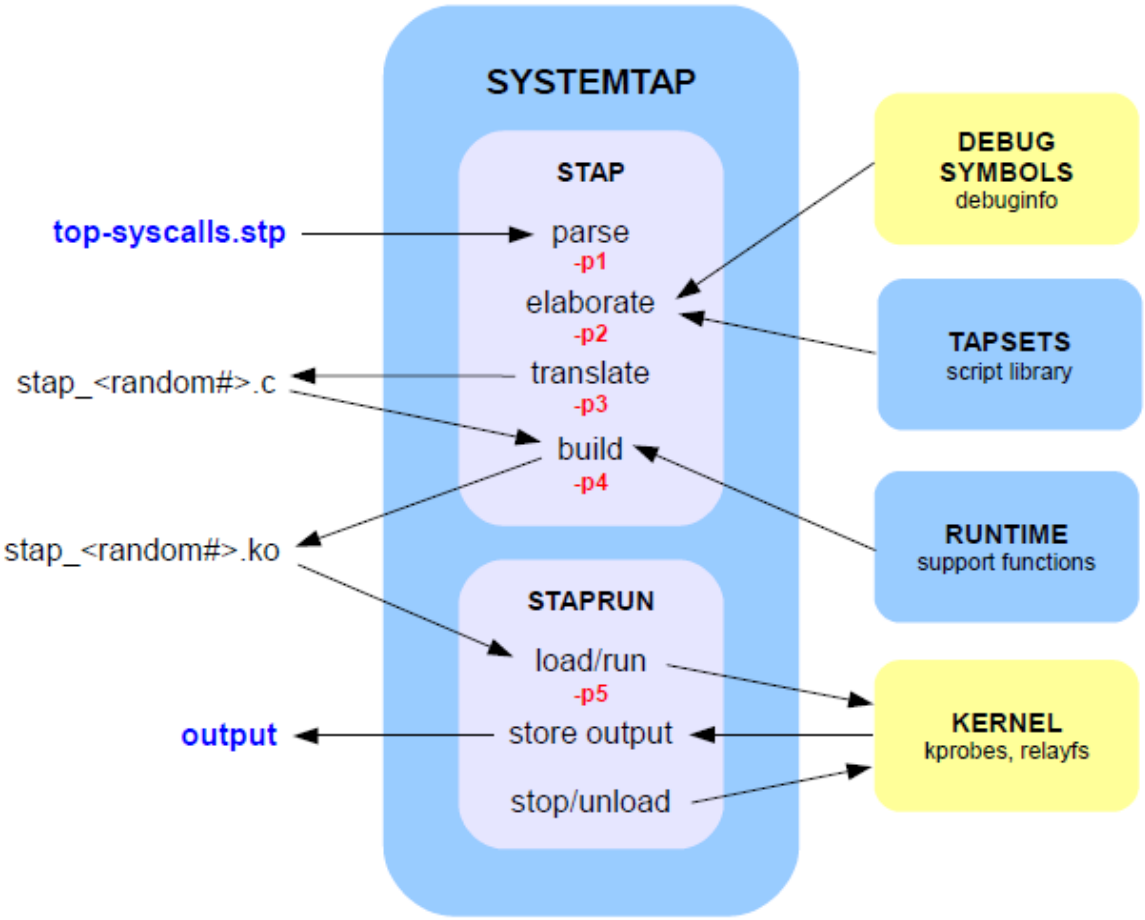
- SystemTap
- Ftrace
- Perf probe
- OpenAFS Crash Plugin

SystemTap

- SystemTap provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system.
- It is based on kprobes / kretprobe.
- Eliminates the tedious and disruptive process of instrumentation, recompile, install, and reboot sequence that may be otherwise required to collect data.
- Provides a simple command line interface and scripting language for writing instrumentation for a live running kernel.

SystemTap Architecture

SystemTap Processing Steps:



Advantages of SystemTap

- No module writing required. Create and insert probes quickly and easily using a simple scripting language.
- No kprobes knowledge required
- No kernel addresses required. Automates gathering of symbol information
- Provides pre-written probes for common kernel areas
- Growing set of pre-written scripts
- Powerful and simple to use

Case Study-1

Problem Statement: Getting -450 (RX Marshal Error) while doing FetchStatus Calls. While we get -450 error, there were tokens expiry messages in syslog files.

Initial Analysis: Since there were token expiry messages, we wanted to inject a fault in routines which return RXKADEXPIRED error.

Wrote below script which returns RXKADEXPIRED error from rxkad_PreparePacket. With this we were able to simulate the issue while executing status calls.

Script

```
probe module("openafs").function("rxkad_PreparePacket").return
{
   printf("Going to return RXKADEXPIRED\n")
   $return = 19270409

}
```

Case Study-2

Problem Statement : We were getting RX_PROTOCOL_ERROR while doing FetchStatus calls.

Initial Analysis:

Step 1: First we needed to find from which place in RX Layer RX_PROTOCOL_ERROR is returned. For that we wrote systemTap script to put probes at all points where RX_PROTOCOL_ERROR was returned.

Script

```
probe module("openafs").statement("*@*/rx.c:3108") {
printf("Hit probe at 3108 : %s####CallNum[%d] Iter[%d]\n",$$vars, callNum, iter)
}

probe module("openafs").statement("*@*/rx.c:3516") {
        printf("Looks we have hit RX_PROTOCOL_ERROR [Process Data %s %d %s] [Probe Data %s:%s]\n",execname(), pid(), pp(), probefunc(),$$vars)
}
…
```

Case Study-2 (cont…)

Step 2: After running systemTap script we came to know the place from where RX_PROTOCOL_ERROR was returned.

Step 3: With our debug data we came to know that while sending data packet call number is 0.

Step 4: So we simulated the problem by making call Number as 0 in our systemTap script. This also served as our Unit test case.

Script:

```
probe module("openafs").statement("*@*/rx.c:3108") {
    header = &@cast($np, "rx_packet", "*openafs*")->header

    callNum = @cast(header, "rx_header", "*openafs*")->callNumber
    if (iter == 1) {
          type = @cast(header, "rx_header", "*openafs*")->type
    if (type == 1)
            @cast(header, "rx_header", "*openafs*")->callNumber = 0    }
```

Performance Measurement

SystemTap can used to gather performance statistics

- Script

```
global entry, timings
function collect_entry()
{ entry[probefunc(),tid()] = gettimeofday_us()
}

function collect_exit()
{ timings[probefunc()] <<< (gettimeofday_us() - entry[probefunc(),tid()])
}

probe module("openafs").function("afs_GetDCache").call
{
      collect_entry()
}

probe module("openafs").function("afs_GetDCache").return
{
      collect_exit()
}

probe end
{
      printf("function count min_us avg_us max_us\n")
      foreach (i in timings)
      {
            printf("%-25s %7d %8d %8d %8d\n", i,
                  @count(timings[i]),@min(timings[i]),
                  @avg(timings[i]), @max(timings[i]))
      }
}
```

- Output

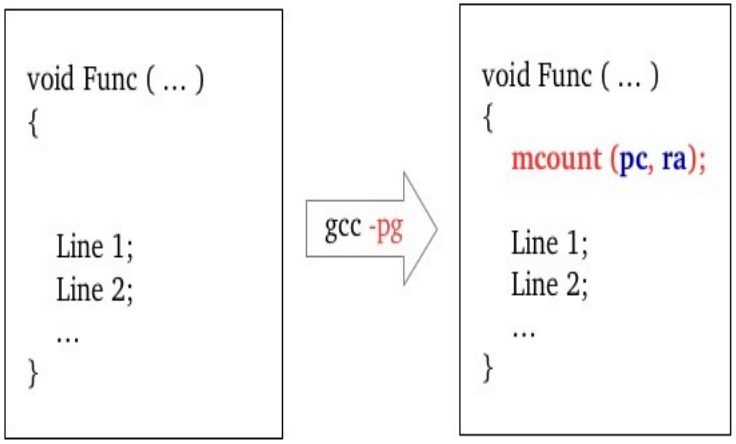
function	count	min_us	avg_us	max_us
afs_GetDCache	271	0	200	52396

Ftrace

- Ftrace is an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel.
- It can be used for debugging or analyzing latencies and performance issues
- There are multiple options with ftrace like trace function, function graph etc
- Use gprof hooks. Add mcount() call at entry of each function call
- Require kernel to be compiled with –pg option
- During compilation mcount call sites are recorded
- Convert mcount() call to NOP at boot time.

Ftrace

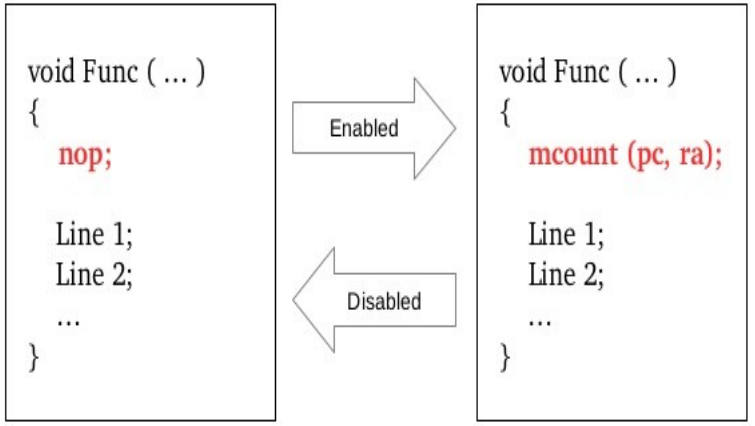
Function Tracer



Re-use gprof mechanism, then re-implement mcount()

02/09/2016 15

Dynamic Function Tracer



02/09/2016 17

Setting up Ftrace

- Currently the API to interface with Ftrace is located in the Debugfs file system. Typically, that is mounted at /sys/kernel/debug.
- When Ftrace is configured, it will create its own directory called tracing within the Debugfs file system.
- For the purpose of debugging, the kernel configuration parameters that should be enabled are:

  CONFIG_FUNCTION_TRACER CONFIG_FUNCTION_GRAPH_TRACER
  CONFIG_STACK_TRACER CONFIG_DYNAMIC_FTRACE

Running Ftrace

- After mounting tracefs you will have access to the control and output files of ftrace. Here is a list of some of the key files:
  - current_tracer
  - available_tracers
  - tracing_on
  - trace
  - set_ftrace_filter
  - set_ftrace_notrace
  - set_ftrace_pid
  - enabled_functions
  - Trace

- Starting & stopping Ftrace:

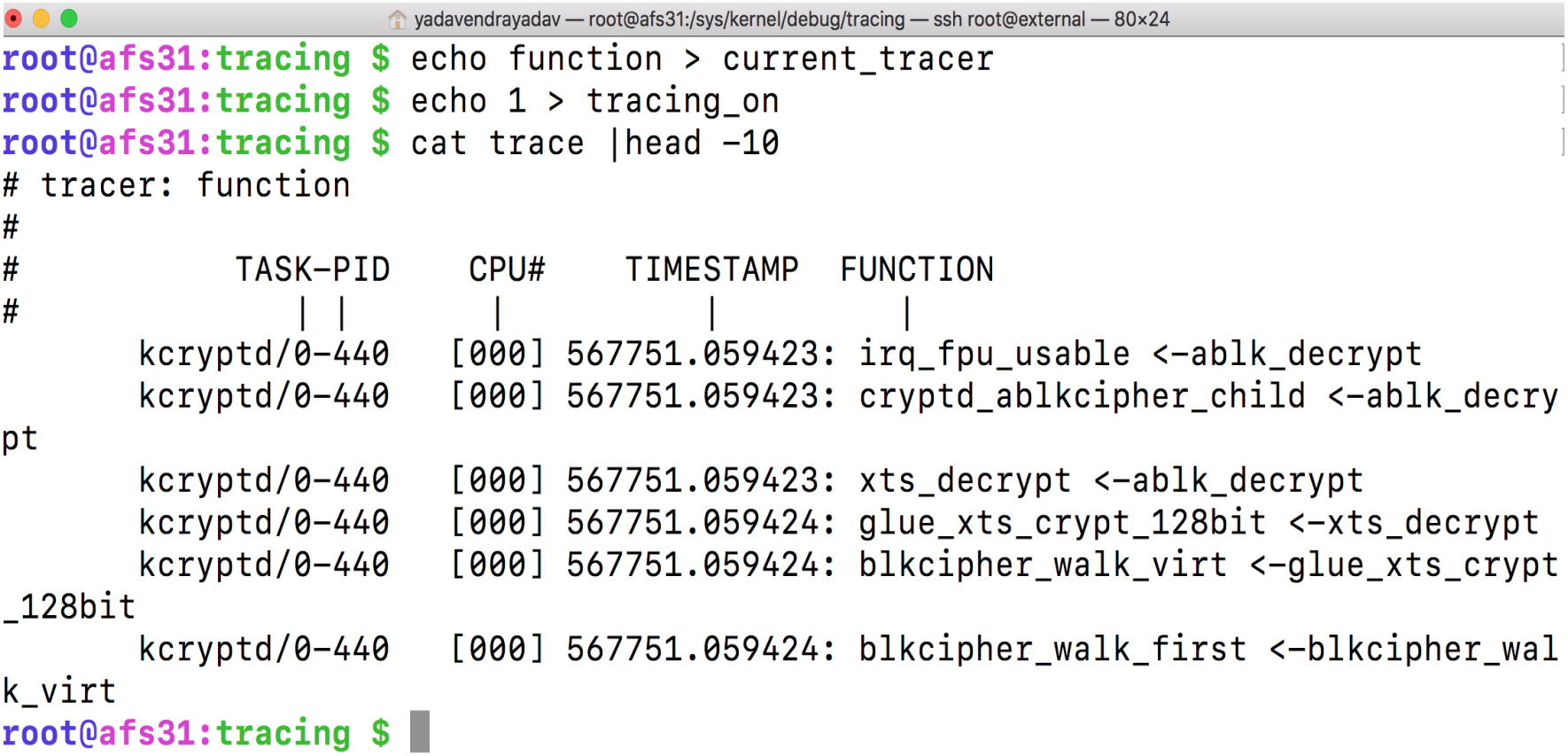
Start :

[tracing]# echo 1 > tracing_on

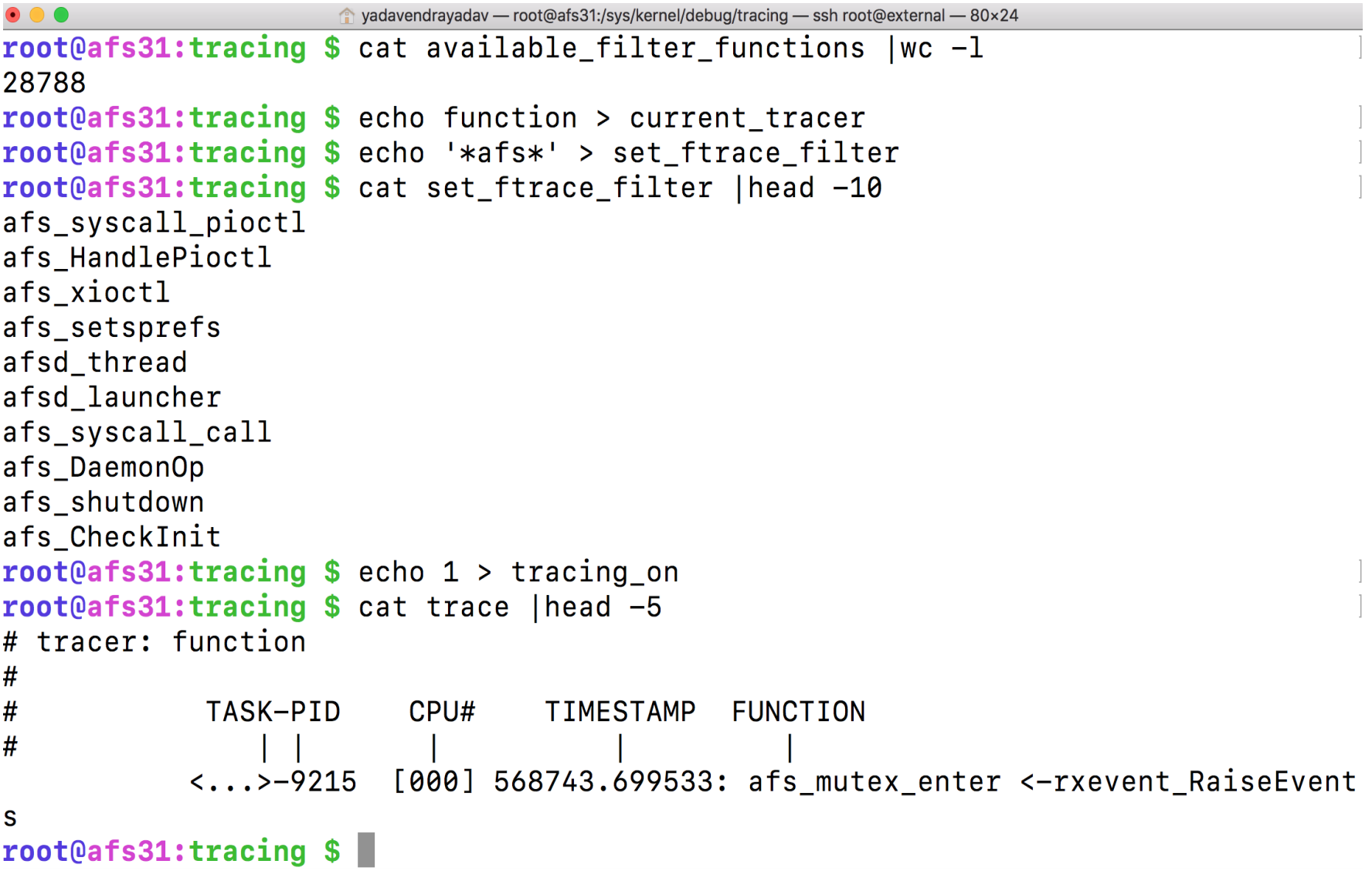
Stop :

[tracing]# echo 0> tracing_on

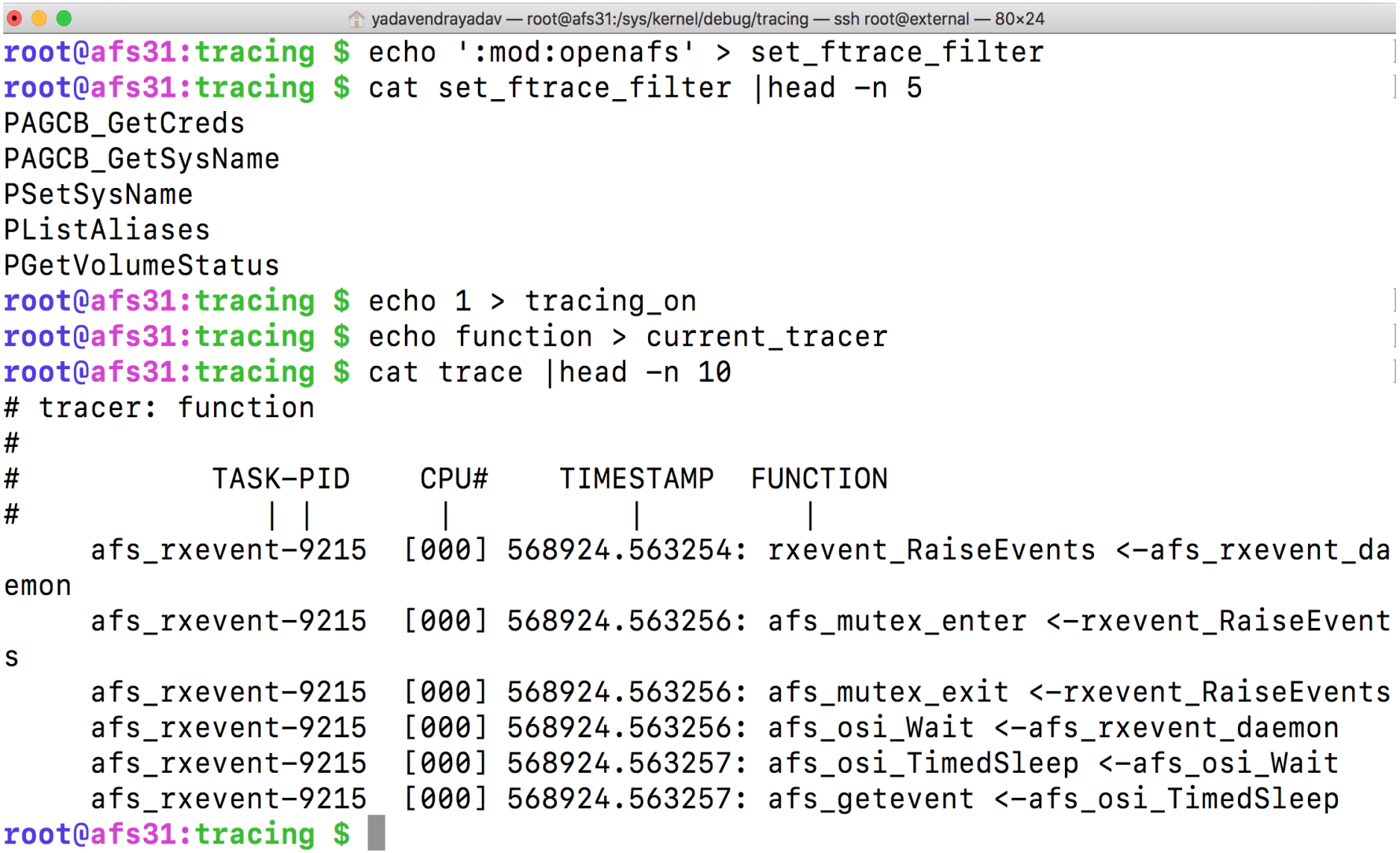
Function Tracer



Function Tracer (cont)



Function Tracer (cont)

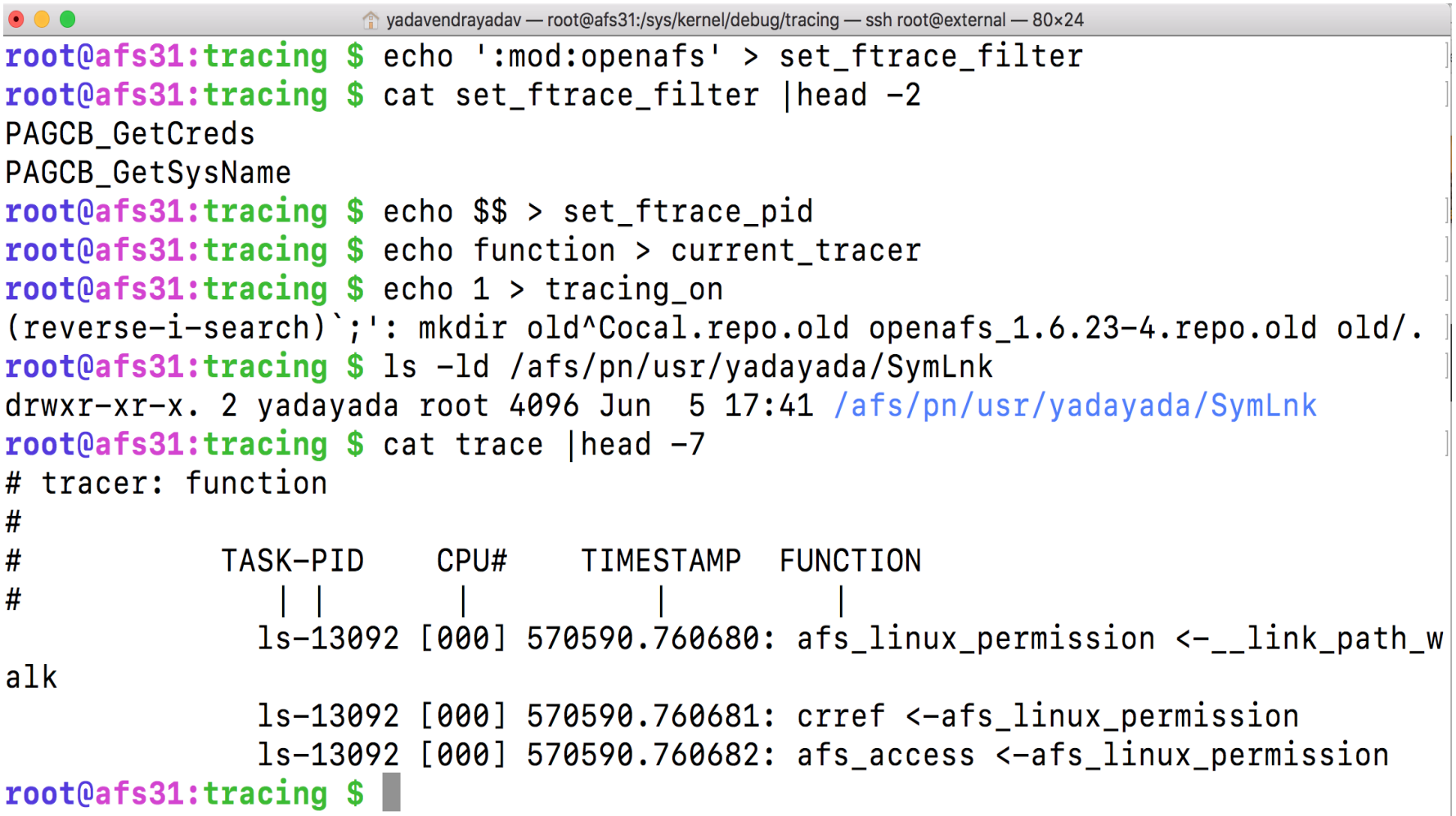


Function Graph Tracer

yadavendrayadav — root@afs31:/sys/kernel/debug/tracing — ssh root@external — 80×24

```
root@afs31:tracing $ cat set_ftrace_filter |head -n 5
PAGCB_GetCreds
PAGCB_GetSysName
PSetSysName
PListAliases
PGetVolumeStatus
root@afs31:tracing $ echo function_graph > current_tracer
root@afs31:tracing $ echo 1 > tracing_on
root@afs31:tracing $ cat trace |head -10
# tracer: function_graph
#
#     TIME        CPU  DURATION                  FUNCTION CALLS
#      |          |     |   |                     |   |   |   |
 0)               |  rxevent_RaiseEvents() {
 0)   0.711 us    |    afs_mutex_enter();
 0)   0.230 us    |    afs_mutex_exit();
 0)   3.223 us    |  }
 0)               |  afs_osi_Wait() {
 0)               |    afs_osi_TimedSleep() {
root@afs31:tracing $
```

Process Tracer



trace_printk()

- If you are debugging a high volume area printk() can bring lots of latency
- Ftrace introduces a new form of printk() called trace_printk().
- Trace_printk does not output to console instead it writes data to ftrace ring buffer

`trace_printk()` output will appear in any tracer, even the function and function graph tracers.

```
[tracing]# echo function_graph > current_tracer
[tracing]# insmod ~/modules/foo.ko
[tracing]# cat trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 3) + 16.283 us   |      }
 3) + 17.364 us   |    }
 3)               |    do_one_initcall() {
 3)               |      /* read foo 10 out of bar ffff88001191bef8 */
 3)   4.221 us    |    }
 3)               |    __wake_up() {
 3)   0.633 us    |      _spin_lock_irqsave();
 3)   0.538 us    |      __wake_up_common();
 3)   0.563 us    |      _spin_unlock_irqrestore();
```

Yes, the `trace_printk()` output looks like a comment in the function graph tracer.

Fetch Kernel Data when Application fails

- The tracing_on and trace_marker files work very well to trace the activities of an application if the source of the application is available. .
- Critical region starts : Write to trace marker stating “critical region started”
- Some call fails : Turn of tracing so that we get trace logs before issue happened.

```
                map_len = 134221824;
/* marker_fd is file descriptor for trace_marker file */

                if (marker_fd >= 0)
                    write(marker_fd, "In critical area\n", 17); /* Start of critical section */

                Ptr = mmap(NULL,map_len,PROT_READ|PROT_WRITE, MAP_SHARED, fd, ARRINDEX[count] );
                if (Ptr == MAP_FAILED) {
                        perror("MMAP failed:");
                        return -1;
                }
                PDEBUG("MMAP SUCCESS offset[%x] MMAPLEN[%d]",ARRINDEX[count],map_len);

                memset(Ptr,'H',map_len);
                PDEBUG("Written data to pages to make them dirty. Going to call msync");
                ret = msync(Ptr,map_len,MS_SYNC);
                if (ret < 0) {
/* During failure turn off tracing, so that we can get trace o/p before failure */
                        if (trace_fd >= 0)
                            write(trace_fd, "0", 1);
                PDEBUG("Errno[%d]",errno);
```

Stack Tracing

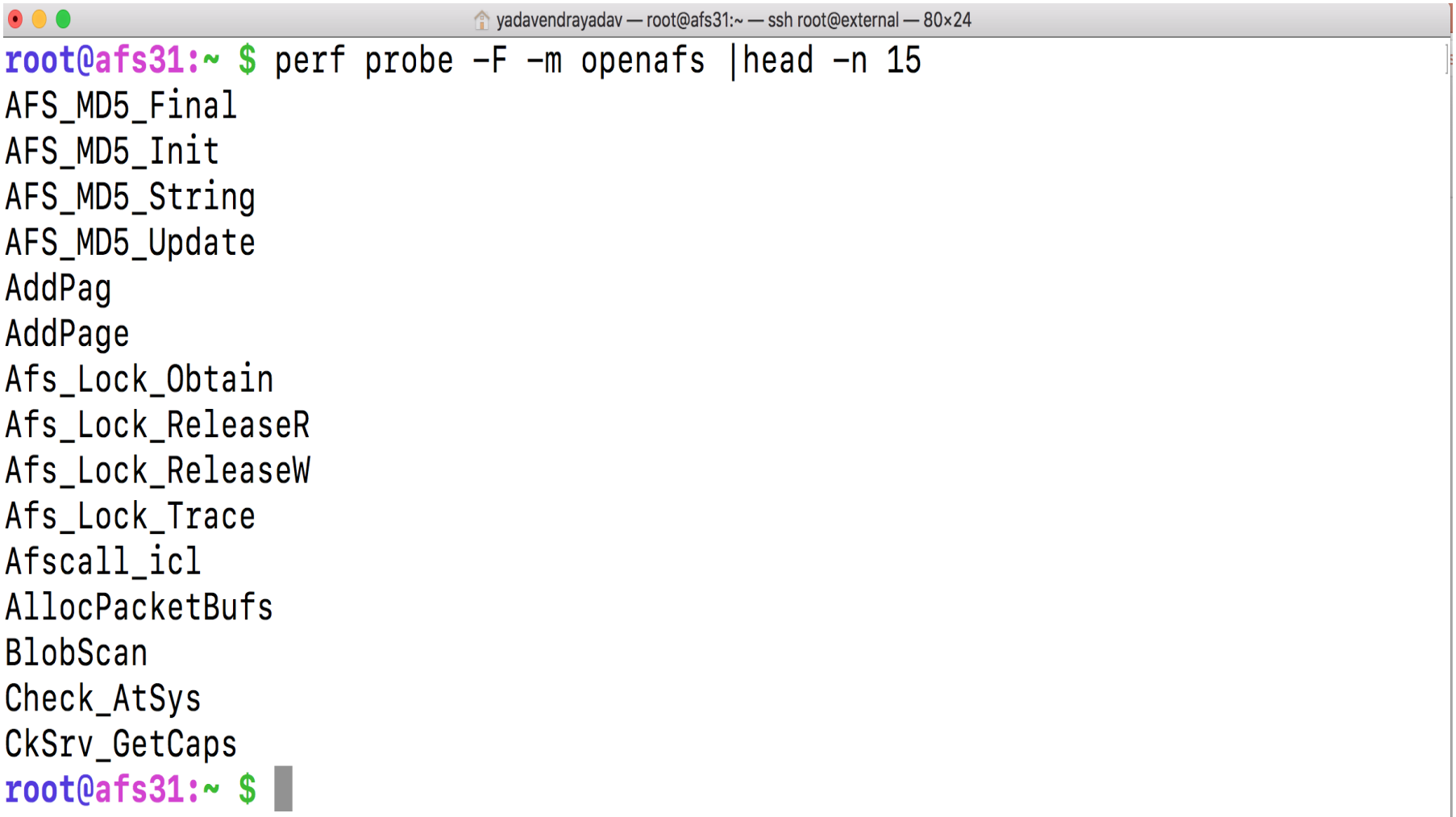
```
root@afs31:tracing $ clear

root@afs31:tracing $ echo 1 > /proc/sys/kernel/stack_tracer_enabled
root@afs31:tracing $ cat stack_trace
        Depth    Size   Location    (20 entries)
        -----    ----   --------
  0)     2336      16   ftrace_pid_func+0x2c/0x40
  1)     2320      16   ftrace_test_stop_func+0x1c/0x30
  2)     2304      96   ftrace_call+0x5/0x2b
  3)     2208      80   afs_lhash_enter+0x33/0x1a0 [openafs]
  4)     2128     112   osi_linux_alloc+0x169/0x440 [openafs]
  5)     2016      16   afs_osi_Alloc+0x37/0x40 [openafs]
  6)     2000     112   afs_DynrootNewVnode+0x176/0x650 [openafs]
  7)     1888     160   afs_GetVCache+0x289/0x590 [openafs]
  8)     1728     320   afs_lookup+0xc86/0x1c60 [openafs]
  9)     1408      96   afs_linux_lookup+0x81/0x350 [openafs]
 10)     1312      96   do_lookup+0x1ab/0x230
 11)     1216     224   __link_path_walk+0x79c/0x1090
 12)      992     224   __link_path_walk+0x501/0x1090
 13)      768      64   path_walk+0x6a/0xe0
 14)      704      64   filename_lookup+0x6b/0xc0
 15)      640     208   user_path_at+0x57/0xa0
 16)      432      96   vfs_fstatat+0x50/0xa0
 17)      336      16   vfs_lstat+0x1e/0x20
 18)      320     144   sys_newlstat+0x24/0x50
 19)      176     176   system_call_fastpath+0x35/0x3a
root@afs31:tracing $ cat stack_max_size
2432
root@afs31:tracing $
```

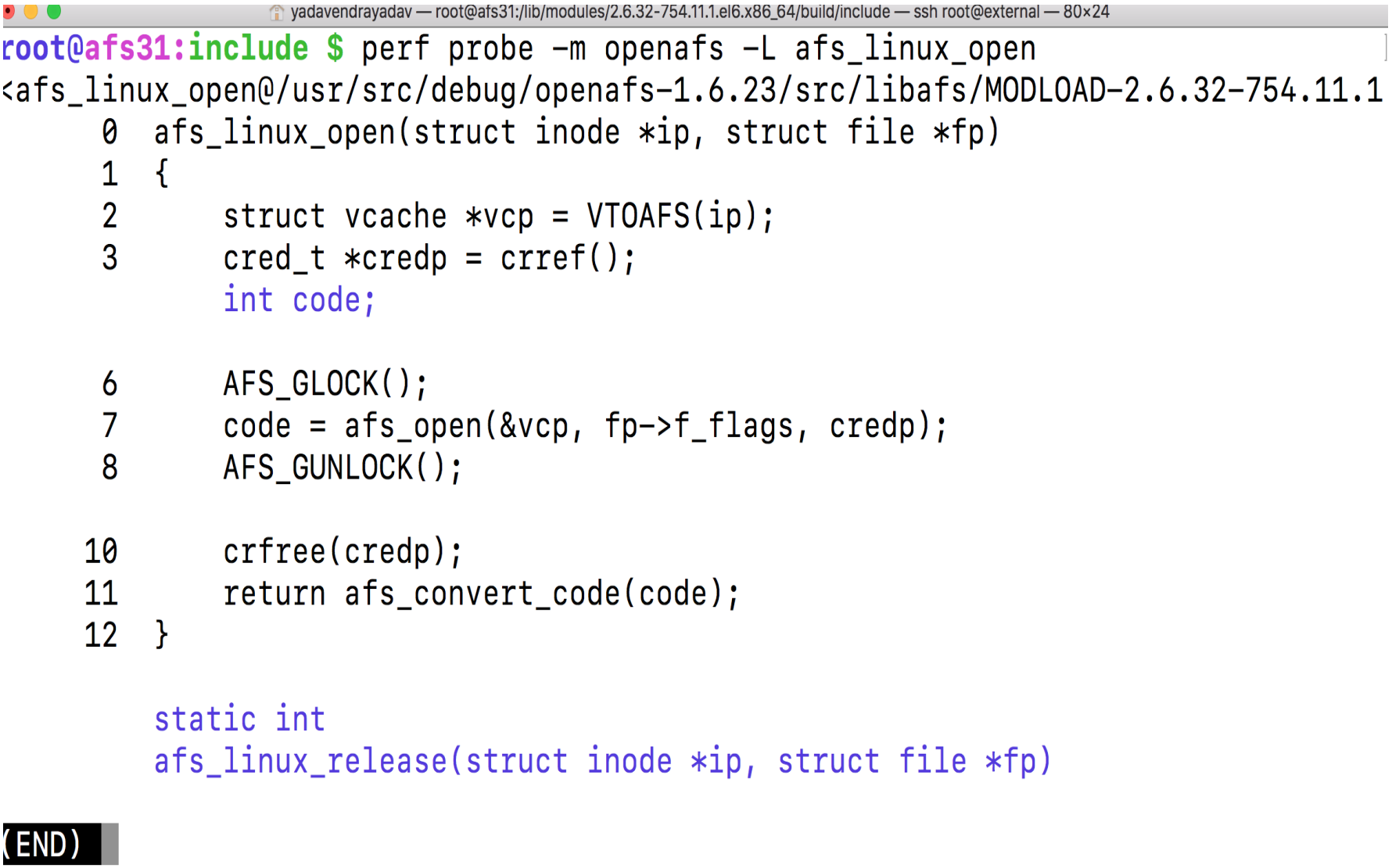
Perf probe

- Allows dynamic trace points to be added or removed inside the Linux kernel
- Besides instrumenting locations in the code, a trace point can also fetch values from local variables, global, registers, the stack, or memory
- Based on kprobe and kretprobe
- Can be used to trace user space also using Uprobes

Perf probe (Cont)



Perf probe (Cont)



Perf probe (Cont)

yadavendrayadav — root@afs31:/lib/modules/2.6.32-754.11.1.el6.x86_64/build/include — ssh root@external — 80×24

```
root@afs31:include $ perf probe -m openafs -V afs_linux_open
Available variables at afs_linux_open
        @<afs_linux_open+0>
                struct file*    fp
                struct inode*   ip
                struct vcache*  vcp
root@afs31:include $
```

yadavendrayadav — root@afs31:~ — ssh root@external — 104×24

```
root@afs31:~ $ perf probe -m openafs -a 'afs_linux_open fp->f_path.dentry->d_inode->i_ino'
Added new event:
  probe:afs_linux_open (on afs_linux_open in openafs with i_ino=fp->f_path.dentry->d_inode->i_ino)

You can now use it in all perf tools, such as:

        perf record -e probe:afs_linux_open -aR sleep 1

root@afs31:~ $ perf record -e probe:afs_linux_open -aR sleep 60
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.336 MB perf.data (1 samples) ]

root@afs31:~ $ perf script
             cat 15190 [000] 575200.498781: probe:afs_linux_open: (ffffffffa058ce80) i_ino=4f3f0172
root@afs31:~ $
```

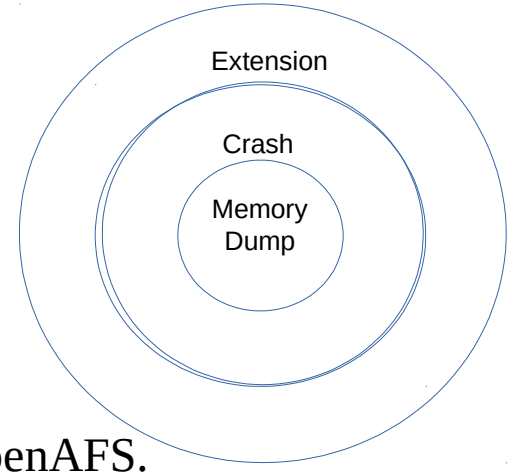
OpenAFS Crash Plugin

What is crash ?

Is the combination of kernel-specific traditional UNIX crash utility with the source code level debugging capabilities of gdb.

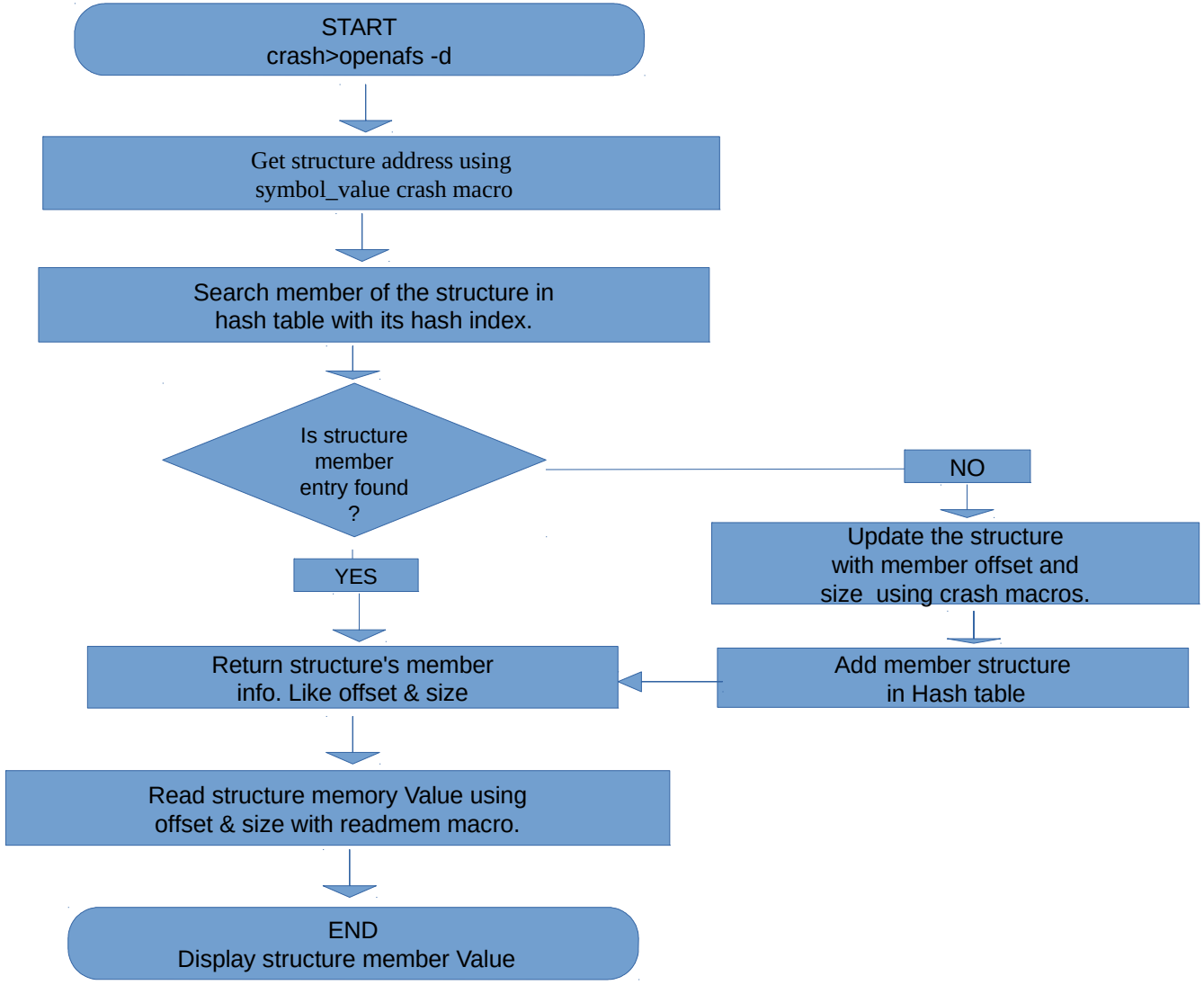
We have implemented crash plugin for Distributed Filesystem like OpenAFS to fetch information from Kernel dumps and Live kernel.

What are the Challenges?



- Displaying various Data structures from OpenAFS kernel dump is a cumbersome process.
- Assembling kernel data to identify issue is time consuming process.
- To avoid above issue we have written a “Crash” Plugin for OpenAFS.

OpenAFS Crash Plugin Architecture



OpenAFS Crash Plugin Benefits and Macros used

Benefits:

- In cloud environment, user can easily identify reason for kernel slowdown/panic with live kernel debugging using crash plugin.
- User can get kernel structure info. with single command.
- Crash" Plugin can also be used to log in-memory kernel information when some unexpected event happens (Log collection)

Crash plugin Macro used:

- Address of a Structure : “Crash” Utility API "symbol_value" is used
- Offset of a Member inside a Structure : “Crash” Utility Macro "MEMBER_OFFSET" is used.
- Size of a Structure Member : “Crash” Utility Macro "MEMBER_SIZE" is used
- Type of a Structure Member : “Crash” Utility Macro "MEMBER_TYPE" is used

Thank You