How to Debug Real-World Problems in the Linux Kernel

James Pascoe

james@jamespascoe.com

www.jamespascoe.com

Hello and Welcome …

1. Why this talk?
   - A lot of kernel debugging is still done with `printk` and `debugfs`
   - These tools are fine, but really only suitable for debugging logic errors
   - Less useful for debugging code that runs ‘at speed’ or for intermittent and asynchronous problems

2. What I am going to present:
   - The Linux crash kernel as a means of debugging sporadic crashes
   - The use of `trace_printk` for debugging critical paths
   - Introduce the eBPF (an in Kernel VM)

The Linux Crash Kernel

- Great for debugging sporadic crashes and lock-ups
- Consists of a ‘dump’ kernel and the `crash` utility
- When a panic occurs, `kexec` is used to boot a new instance of the kernel which dumps the state of the compromised system to `/var/crash`
- The image can then be analysed using the (powerful) GDB like `crash` utility to look at `dmesg`, memory, lock state, back traces for all CPUs etc.

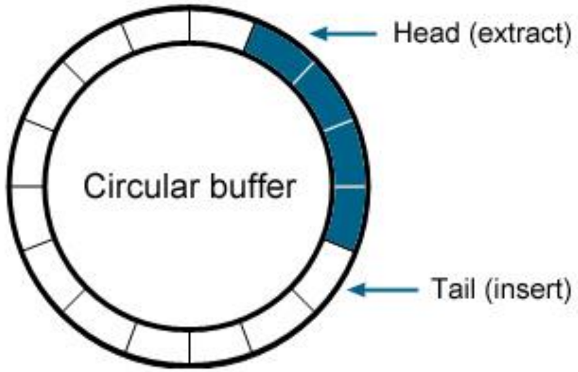
Installing the Crash Kernel on Ubuntu

1. Run: `sudo apt install linux-crashdump`
    - https://help.ubuntu.com/lts/serverguide/kernel-crash-dump.html

2. Recompile your kernel with debug symbols
    - (optional) Enable ‘Panic on Soft / Hard Lock-up’ in the Kernel config
    - https://wiki.ubuntu.com/Kernel/BuildYourOwnKernel

3. Build the `crash` utility from latest sources (recommended)
    - https://github.com/crash-utility/crash.git

Using `trace_printk`



- Part of the `Ftrace` utility built into the kernel
- Writes to the `Ftrace` ring buffer (0.1 microseconds) and not the console. A `printk` running over a serial connection can take several milliseconds per write !
- Trace is in `/sys/kernel/debug/tracing/trace`
- Output can be be piped and appears in all tracers

- See: https://lwn.net/Articles/365835

trace_printk Example


```
trace_printk("read %d bytes from %p\n", num, buffer);

[my_host]# cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
# TASK-PID CPU# TIMESTAMP FUNCTION
# | | | | | <...>-10690 [003] 17279.332920: : read 10
bytes from ffff880013a5bef8
```

The Enhanced Berkeley Packet Filter

- The eBPF is a programmable VM built into the kernel
- Originally for network packet analysis, but is now used for debugging, profiling answering ‘what if’ questions
- eBPF programs are ‘attached’ to kernel code paths
- When a code path is traversed, the code is run
- eBPF programs are written in C and can be compiled and reloaded dynamically, so no need to rebuild the kernel ☺ !
- Bindings for Python, Lua and Rust

- See: https://lwn.net/Articles/740157/

eBPF Example – Software Validation

- Implemented 802.11ad Packet reordering
- Performance critical – how can we validate it?

1. Define tracepoints in the Linux driver (useful for Ftrace)
2. Implement a Python ‘checker’ script (containing embedded eBPF code) to check the packet order.
3. Run the ‘checker’ under a variety of conditions

Learning Points

- The purpose of this talk was to extend knowledge of Linux Kernel debugging techniques ☺ !

1. Use a crash kernel for debugging sporadic crashes
    - Massively improves quality of bug reports
2. Use `trace_printk` for performance critical code
    - Use `printk` / `pr_` macros for syslog (user) messages
3. Learn about the eBPF:
    - Extremely powerful – replaces kprobes, jprobes etc.

```cpp
#include <iostream>

int main()
{
  std::string questions;
  while (1)
  {
    std::cout << "Questions?" << std::endl;

    if (std::cin >> questions && questions == "Y")
      std::cout << "Answers" << std::endl;
    else
      goto next_speaker;
  }

next_speaker:
  std::cout << "Thank you for listening !" << std::endl;
}
```