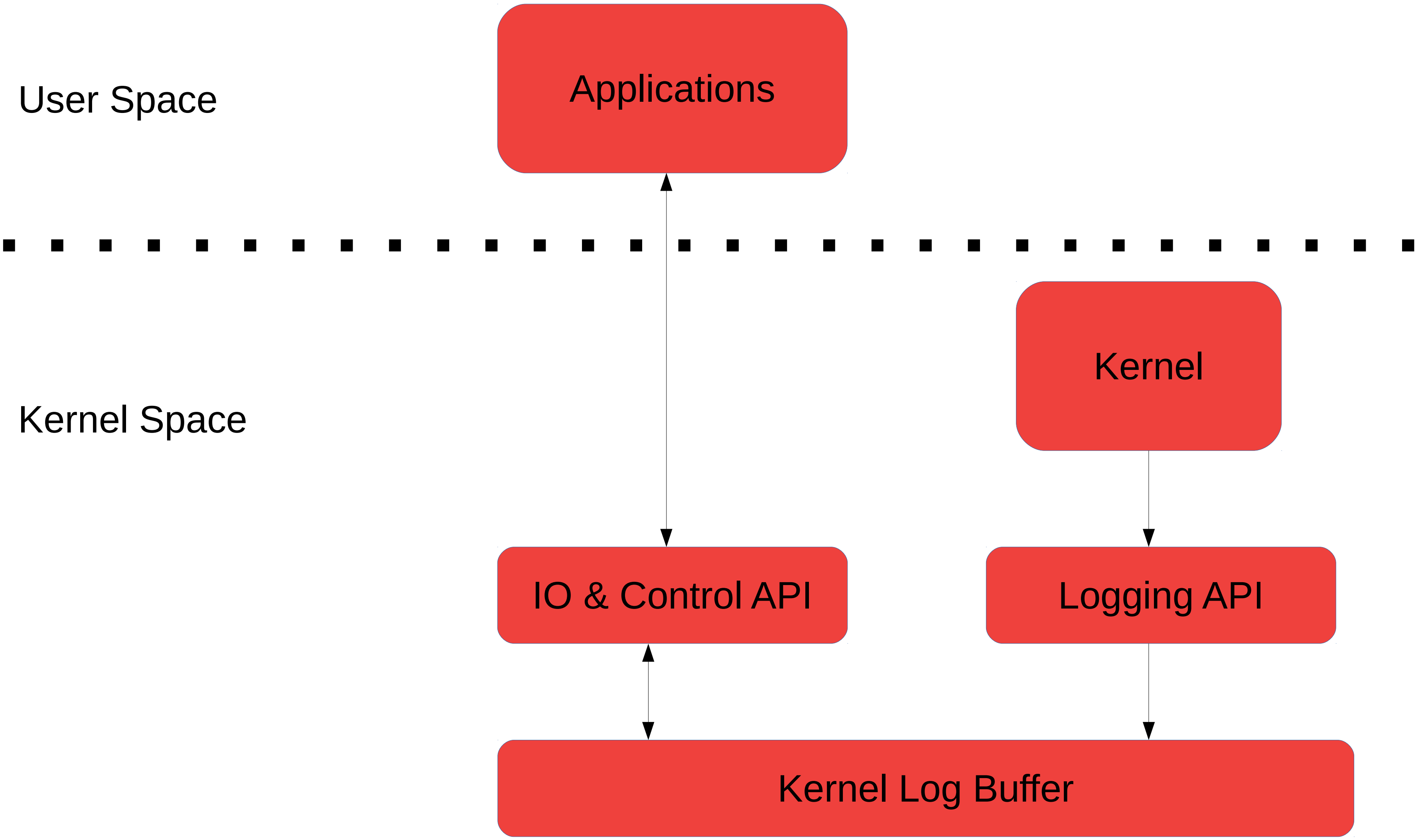
Kernel Debugging

What to Expect

- Debugging by printing
- DebugFS
- Kgdb
- Analyzing the oops
- Analyzing the boot up time
- Other debugging mechanisms

Kernel Logging System Architecture



Kernel Log buffer

- Default size is 64KB
- For modifying the size
  - Kernel Config Option
    - CONFIG_LOG_BUF_SHIFT=n
      - Menuconfig -> General Setup
  - Uboot bootargs: log_buf_len=n
    - Buffer Size = 2^n
      - n = 16: 64 KB
      - n = 17: 128 KB

Debugging by printing

- Simplest & the most commonly used debug method
  - printk(KERN_ALERT “reached line %d in function %s\n”, __LINE__, __func__);
- Limitations
  - Recompile & reboot every time, the new section needs debugging
  - Prints are relatively resource intensive
  - Might not help if the bug is related to timing/resource contention

Debugging by printing ...

- Using pr_* family of functions
  - Shorthand definitions for the respective printk call
  - pr_emerg, pr_alert, pr_crit, pr_err, pr_debug, pr_devel
  - pr_devel and pr_debug are replaced with printk(KERN_DEBUG ..) if the kernel was compiled with DEBUG, otherwise replaced with empty statement
- Example
  - pr_emerg(“Error in allocation\n”);

Debug prints

- dev_*
  - Special version of printk wrapper routines for device drivers
  - Show the extra information
  - dev_emerg, dev_crit, dev_alert, dev_err, dev_warn, dev_notice
  - Used when printing something related to devices
- dev_dbg & pr_debug
  - Not compiled by default
- #define DEBUG at the beginning of the driver
- Using ccflags-{CONFIG_DRIVER} += -DDEBUG in the Makefile

Log Level

- Log level in the message defaults to DEFAULT_MESSAGE_LOGLEVEL
  - Can be set via the CONFIG_DEFAULT_MESSAGE_LOGLEVEL kernel config option (make menuconfig-> Kernel Hacking -> Default message log level)
  - The log level is used by the kernel to determine the importance of a message and to decide whether it should be presented to the user immediately
  - To determine your current console_loglevel you simply enter:
    - cat /proc/sys/kernel/printk

      ```
      7        4          1          7
      current  default    minimum    boot-time-default
      ```
  - echo 8 > /proc/sys/kernel/printk
  - #set console_loglevel to print KERN_WARNING (4) or more severe messages
  - # dmesg -n 5

Rate Limiting & one time messages

- Inserting a printk in a section which gets called quite often might result in a severe performance impact
  - Could overwrite & spam the kernel buffer
- printk_once(...)
  - no matter how often you call it, it prints once and never again
- #include <kernel/ratelimit.h>
- printk_ratelimited(...)
  - it prints by default not more than 10 times in every 5 seconds (for each function it is called in).

Printing from user space

- To annotate, its quite useful to insert some messages in the kernel log buffer
  - echo "Hello Kernel-World" > /dev/kmsg
    - Prints with the default log level
  - echo "<2>Writing critical printk messages from userspace" >/dev/kmsg
    - To issue a KERN_CRIT message
  - Example:
  - echo “### TESTNOTE: unplugged thumb drive” > /dev/kmsg
  - echo “### TESTNOTE: waited for a couple seconds” > /dev/kmsg
  - echo “### TESTNOTE: re-plugged thumb drive” > /dev/kmsg

Printing buffers as hex

- print_hex_dump_bytes(const char *prefix_str, int prefix_type, const void *buf, size_t len)
- static inline void print_hex_dump(const char *level, const char *prefix_str, int prefix_type, int rowsize, int groupsize, const void *buf, size_t len, bool ascii)
- prints a buffer as hex values to the kernel log buffer (with level KERN_DEBUG)
- Useful for creating the memory dumps
- Example
  - char mybuf[] = "abcdef";
  - print_hex_dump_bytes("", DUMP_PREFIX_NONE, mybuf, ARRAY_SIZE(mybuf));
  - dmesg output:
  - 61 62 63 64 65 66 00   abcdef.

Dynamic debug

- Can be used to enable/disable debug information dynamically
  - Kernel needs to be compiled with CONFIG_DYNAMIC_DEBUG
  - Useful tool to only get the debug messages you are interested in
  - pr_debug()/dev_dbg() and print_hex_dump_debug()/print_hex_dump_bytes() calls can be dynamically enabled per-callsite
  - If CONFIG_DYNAMIC_DEBUG is not set, print_hex_dump_debug() is just shortcut for print_hex_dump(KERN_DEBUG).

Dynamic Debug ...

- Simple query language allows turning on and off debugging statements by matching any combination of:
  - source filename
  - function name
  - Line number (including ranges of line numbers)
  - module name
- Enable debug messages during boot process
  - dyndbg=“QUERY” <-- for kernel
  - module.dyndbg=“QUERY” < -- for module

Dynamic debug control options

- Using DebugFS
  - mount -t debugfs none /sys/kernel/debug/
  - # cd /sys/kernel/debug/dynamic_debug/
  - # echo “file xxx.c +p” > control
  - # echo “file svcsock.c line 1603 +p” > control
  - # echo “file drivers/usb/core/* +p” > control
  - # echo “file xxx.c -p” > control
- uboot bootargs
  - dyndbg=“QUERY” <-- for kernel
  - module.dyndbg=“QUERY”   < -- for module

Using DebugFS

- A simple memory based filesystem designed specifically to debug Linux kernel code
- Helps kernel devlopers export large amount of debug data into user space
- Kernel Configuration: CONFIG_DEBUG_FS
  - Kernel hacking -> Debug Filesystem
- Mount debugfs with command
  - mount -t debugfs nodev /sys/kernel/debug

DebugFS API

- Create a subdirectory in /sys/kernel/debug
  - Struct dentry *debugfs_create_dir(const char *name, struct dentry *parent)
  - Expose an integer using file in DebugFS
    - Struct dentry *debugfs_create_u8(const char *name, mode_t mode, struct dentry *parent, u8 *value)
  - Expose a binary blob
    - Struct dentry *debugfs_create_blob(const char *name, mode_t mode, struct dentry *parent, struct debugfs_blob_wrapper *blob)

Kernel Probes

- Mechanism to write the modules that can add debug information to the kernel
- An alternative to building custom kernels or custom modules
- Dynamically breaks into any kernel routine and can collect debugging and performance information non-disruptively.
- Typical use case
  - Debugging a remote machine where dmesg is not enough to debug. Build a kprobe module & then insmod on remote machine
- Types
  - jprobes
    - Function is called on the entry to the routine. All the arguments to the routine are passed
  - kprobes
    - Any arbitrary kernel instruction can be probed. A function is called passing the registers
  - Kretprobe
    - Call a function on the exit from the routine. The registers are passed

ftrace

- Stands for Function Tracer
- Can be used for
  - Debugging Linux Kernel
  - Analyzing latencies in Linux Kernel
  - Learn & observe the flow of Linux Kernel
  - Trace Context switches
  - Length of the time the interrupts are disabled
- Kernel Configuration
  - CONFIG_FTRACE --> "Tracers"
  - CONFIG_FUNCTION_TRACER --> Kernel Function Tracer
  - CONFIG_FUNCTION_GRAPH_TRACER --> Kernel Function Graph Tracer
  - CONFIG_DYNAMIC_TRACE --> Enable/Disable ftrace dynamically

ftrace operations

- Mount tracefs
  - Adding the entry into the fstab
    - tracefs   /sys/kernel/tracing tracefs_defaults 0 0
  - Using the mount command
    - mount -t tracefs nodev /sys/kernel/tracing
- available_tracers
  - Lists what all tracers have been enabled in the kernel configuration
- current_tracer
  - The tracer currently is running
- trace
  - Contains the tracing data in human readable format
- tracing_on
  - Enable/disable writing tracing data to ring buffer (ftrace uses a separate ring buffer to store tracing data)
- To enable function tracer
  - echo "function" > current_tracer

Function Graph

- Is Used to
  - track the entry of the function
  - track the exit of the function
  - find the Execution Time
  - get the CPU on which it is running
- Useful for following the flow of execution within the kernel

Tracing a specific process

- Steps to trace the process
  - Disable tracing
    - echo "nop" > current_tracer
  - Echo pid of the process which you want to trace in "set_ftrace_pid" file
    - echo "2588" > set_ftrace_pid
  - Enable the function tracer
    - echo "function_graph" > current_tracer

Dynamic Tracing

- Used to filter just the function we need and eliminate those we don’t need
- Can be done with the file ‘set_ftrace_filter’
  - cat available_filter_functions
  - echo vmalloc_* > set_ftrace_filter
- https://01.org/linuxgraphics/gfx-docs/drm/trace/ftrace.html

MMIO tracing

- Refer https://www.kernel.org/doc/Documentation/trace/mmiotrace.txt

trace_printk

- Limitations with printk
  - Using printk in interrupt context can create a live lock
  - The bug might disappear if printk is added, in case, its time sensitive
  - May take several milliseconds when writing to the console
- trace_printk advantages
  - Writing will be in the order of microseconds as it writes to a ring buffer instead of console
  - Can be used in any context (interrupt, scheduler, NMI Code)
  - Can be read via the 'trace' file

perf Tool

- A profiling tool which offers support for tracing applications and also inspecting the general aspects of the system
- Allows to take a look at what functions are being called at a given point
- Allows us to take a peak at where the kernel is spending most of the time, prints out the call stack and in general logs what the cpu is running
- sudo perf record -a -g
- perf report --header -F overhead,comm,parent
- sudo perf timechart record
- sudo perf timechart --> Generates the .svg file

Kernel Debuggers

- Two debugger front ends – KDB & KGDB
- KDB
  - Simplistic shell-style interface
  - Used to inspect memory, registers, process lists, dmesg and even set breakpoints to stop in a certain location
  - Not a source level debugger
  - Aimed at doing some analysis for developing or diagnosing kernel problems
- KGDB
  - To used as a source level debugger
  - Used along with the gdb to debug a linux kernel
  - Gdb can be used to break-in to the kernel to inspect memory, variables and look through call stack information

Kernel GDB

- Provides an interface to gdb via its remote serial protocol
- Implements a gdb stub that communicates to the cross gdb running on host
- Kernel Configuration
  - CONFIG_FRAME_POINTER=y
  - CONFIG_KGDB=y
  - CONFIG_KGDB_SERIAL_CONSOLE

KGDB setup

- Configure KGDB from command line
  - kgdboc = <tty-device>, <bauds>
  - Add kgdbwait to make kgdb wait for the debugger connection
- On the host
  - arm-linux-gdb ./vmlinux
  - Set remotebaud 115200
  - Target remote /dev/ttyUSB0

Kernel Oops

- An exception in the kernel code
- Kernel dumps this message when it finds something faulty
- Contains the processor state & the CPU registers of when the fault occurred
- The offending process gets killed without even releasing the locks or cleaning up the data structures
- System cannot be trusted further, once the oops have happened

Analyzing Kernel oops

- BUG: What caused the oops
- PC: Instruction pointer
- Internal error: [#1] SMP – This is error code in hex
  - Varies as per architecture
- CPU 1 – the CPU on which the error occurred
- Call Trace - the list of functions being called just before the Oops occurred
- Code: The Code is a hex-dump of the section of machine code that was being run at the time the Oops occurred

Debugging an Oops dump

- gdb test.ko
- (gdb) add-symbol-file test.o <address>
  - Add the symbol file to the debugger
  - The address of the test section of the module
    - cat /sys/module/test/sections/.init.text
- (gdb) disassemble my_oops_init
  - my_oops_init is the offending function
  - We can get it from the PC
- Add the starting address & the offset to pin point the actual line of offending code
- (gdb) list *(address)

Analyzing the boot up time

- Variety of tools available to measure the boot up time for the linux system
- grabserial
  - One of the simplest tool
  - Reads the serial port and wirtes the data to the standard output
  - grabserial -d /dev/ttyUSB0 -t

Other debugging mechanisms

- Adding the ioctl commands for debugging mechanisms
- Adding entries in the proc filesystem
- Adding debuging entries in sys filesystem

What all did we learn?

- Debugging by printing
- DebugFS
- Kgdb
- Analyzing the oops
- Other Debugging Mechanisms