Day 4: Kernel Debugging

Swapnil Pimpale
(swapnil@geeksofpune.in)

Agenda:

- Need for kernel debugging
- Kernel debugging techniques (procfs, sysfs, lsof, fuser, objdump, debugfs)
- Kernel debuggers (gdb, kdb, kgdb)
- KGDB overview
- KGDB internals
- KGDB in action
- References

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it

Brian W. Kernighan's Law of

Debugging Difficulty

Need for kernel debugging

- Bugs in kernel code frequently result in a lockup or a reboot
- It's difficult to locate the problem – Entry points
- Kernel code is difficult to execute under a debugger
- Kernel code errrors can be hard to reproduce
- Debugging techniques help monitor kernel code, trace errors and collect useful information

Kernel debugging techniques

- Logging
  - printk, kernel logging framework, /proc, /sys interfaces
- Dump kernel state (crash dump)
  - registers, memory, threads
- Live debugging
  - kernel debuggers – KDB, KGDB

printks

- Easy, printf like
- Lets you classify messages according to their severity by associating different ***loglevels*** with the messages
- Can be used from most kernel code
- Can be turned on/off and can also be rate-limited *(**printk_ratelimited**)*

printk loglevels

- loglevels defined in ***include/linux/kern_levels.h***
- Specify severity
- **NO** comma between loglevel and format string
- If you don't specify a loglevel, default is used
- CONFIG_DEFAULT_MESSAGE_LOGLEVEL
- ***console_loglevel***
  $ cat /proc/sys/kernel/printk

```
7        4        1          7
current  default  minimum    boot-time-default
```

Name	String	Meaning	alias function
KERN_EMERG	“0”	Emergency messages, System is about to crash or is unstable	pr_emerg
KERN_ALERT	“1”	Something bad happened and action must be taken immediately	pr_alert
KERN_CRIT	“2”	Critical condition occurred like a serious HW/SW failure	pr_crit
KERN_ERR	“3”	An error condition	pr_err
KERN_WARNING	“4”	A warning, meaning nothing serious by itself but might indicate problems	pr_warn
KERN_NOTICE	“5”	Nothing serious, often used to report security events	pr_notice
KERN_INFO	“6”	Informational e.g. startup information at driver initialization	pr_info
KERN_DEBUG	“7”	Debug messages	pr_debug/pr_devel
KERN_DEFAULT	“d”	Default kernel loglevel	
KERN_CONT	“”	Continued line	pr_cont

printk Cons

- Slow
- Useless for analyzing races
- Requires code instrumentation – a compile-run cycle
- Mistakes can result in kernel crashes

/proc filesystem

- Virtual filesystem
- No 'real' files but runtime system info
- ***lsmod -> cat /proc/modules***
- File size ZERO (except some)
- Pointer to where the actual information resides

ls /proc/PID/

- */proc/PID/cmdline:* Command line arguments
- */proc/PID/environ:* Values of environment variables
- */proc/PID/cwd:* Current working dir
- */proc/PID/exe:* Link to the executable of this process
- */proc/PID/fd/:* Open file descriptors
- */proc/PID/status:* Process status in human readable form

/proc filesystem

- */proc/interrupts:* Number of interrupts per CPU per IO device
- */proc/cmdline:* Kernel command-line
- */proc/cpuinfo:* Type, make, model, flags
- */proc/filesystems:* Filesystems configured/supported into/by the kernel
- */proc/meminfo:* Memory usage
- */proc/modules:* Kernel modules currently loaded
- *procinfo* utility

/proc/sysrq-trigger

- ***/proc/sys/kernel/sysrq:*** Bitmask of allowed sysrq functions
- Kernel will respond regardless of whatever else it is doing
- Write a character (for ex. 'g') to /proc/sysrq-trigger
  - ***echo g > /proc/sysrq-trigger***

/proc filesystem

- Plenty of information exported
- */proc/[pid]:* Numerical sub-directory for each running process containing a lot of process related info
- */proc/cpuinfo:* Information about the CPUs
- */proc/slabinfo:* Information about kernel caches
- */proc/interrupts:* Number of interrupts per CPU
- */proc/vmstat:* Virtual memory statistics

sysfs

- Introduced in Linux 2.6 / originally ddfs
- Allows kernel code to export information to user processes via in-memory filesystem
- (Mostly) ASCII files with (usually) one value per file
- Sysfs mappings
    - Kernel objects --> Directories
    - Object attributes --> Regular Files
    - Object relationships --> Symbolic Links

sysfs History

- Device driver filesystem
- Written to debug the new driver model
- Originally used procfs
- Converted later – Linus
- *driverfs* – 2.5.1

sysfs

- Mount: *mount -t sysfs sysfs /sys*
- Navigating (tree):
  - *block:*
    - */sys/block/sda/queue/scheduler*
  - *bus:*
    - physical bus types that have support registered in the kernel
    - devices: every device discovered on that type of bus in the entire subsystem
    - drivers: each device driver that has been registered with that bus type

sysfs

- Navigating (tree):
  - *module:*
    - subdirectory for each module loaded in the kernel
    - *sections:* debugging
    - *refcnt:* number of users
- Code:
  - *fs/sysfs/*
  - *include/linux/sysfs.h*

sysfs

- Directory creation:
  - For every kobject registered with the system a kobject is created
  - represents a kernel object, maybe a device
  - *kref:* reference counting on the kobject
- Attributes:
  - Exported in the form of regular files
  - sysfs provides a means to read/write
  - one value / array of values of the same type

sysfs

- Callbacks:
  - *struct sysfs_ops*
    - *ssize_t (*show)(struct kobject *, struct attribute *, char *);*
    - *ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);*
- When an attribute is opened, a PAGE_SIZE buffer is allocated for transferring the data between the kernel and userspace (4096)

strace

- Shows a log of system calls, arguments to the calls and their return values in symbolic form
- Works on programs regardless of whether or not compiled with debugging support and stripping
- Locate cause to user or kernel land
- Compare strace log with expected set of system calls
- ***strace -o /tmp/log /bin/ls /***

strace

- trace system calls and signals
- name, arguments, return values of each system call printed
- diagnostic, instructional, debugging tool
- No recompilation required
- Bug isolation between user/kernel boundary

strace

- Trace execution of an executable
  - *$ strace ls*
- Trace execution of a specific system call
  - *$ strace -e open ls*
  - *$ strace -e trace=open,read ls*
- Save output to a file
  - *$ strace -o output.txt ls*

strace

- Trace a running process
- Print timestamp (-t)
- Relative time taken for system calls (-r)
  - What is taking time?
- What is wrong with my network connection?
  - ***strace -e poll,select,connect,recvfrom,sendto nc www.google.com 80***
- Generate statistics report (-c)
- ***strace ./failed_open***

lsof

- list open files
- ls + of
- information about files that are opened by various processes
- Everything is a file (pipes, sockets, directories, devices, etc.)
- FD – file descriptor
  - cwd – current working directory
  - txt – text file
  - mem – memory mapped file
  - mmap – memory mapped device
  - NUMBER – represents actual FD.
    - r – read, w – write, u – read/write

lsof

- TYPE – type of file
  - REG – regular file
  - DIR – directory
  - FIFO – first in first out
  - CHR – character special file
- List processes which opened a specific file
- List processes which opened a specific file *(+D, +d)*
- List opened files based on process names starting with
  - ***lsof -c ssh -c init***

lsof

- List processes using a mount point
- List files opened by a specific user *(-u)*
- List all open files by a specific process *(-p)*
- List all network connections *(-i)*

fuser

- Who is using the file or directory?
- Kill Processes that are using a particular Program
  - *fuser -v -k*
- Interactively Kill Processes using fuser
  - *fuser -v -k -i*

gdb

- Debugger for several languages, including C and C++
- Most helpful when used to debug a program which has debugging symbols
- gcc's option -g
- *backtrace*
- Value at an address: *x*
- Load a program to debug: *file <prog_name>*
- Run the program: *run*

gdb

- Set breakpoints:
  - *break <file_name>:<line_no>*
  - *break <function_name>*
- Continue to the next breakpoint: *continue*
- Single step – execute one line of code at a time: *step*
- Don't step into functions: *next*
- Print value of a variable: *print, print/x*

gdb

- Set watchpoints
  - *watch my_var*
  - Which my_var?
- Conditional breakpoints
  - *break file.c:6 if i >= 10*

gdb

- Pointers:
  - *struct student {*

    int id;

    *char *name;*

    } s1;
  - *print s1*
  - *print s1.id, print s1.name*
  - *print list_head->next->next->next->data*

gdb

- ***gdb /usr/src/linux/vmlinux /proc/kcore***
- *kcore* represents the kernel executable in core file format
- Can print variables, structures, follow pointers
- Cannot modify kernel data
- Cannot set breakpoints / watchpoints
- Cannot single step through kernel functions
- Needs to be taught how to examine LKM

objdump

- Object dump – information from object files
- ***objdump [options] objfile...***
- Display contents of file header *(-f)*
- Display object format specific file header contents *(-p)*
- Display contents of section headers *(-h)*
- Display contents of all headers *(-x)*
- Display assembly contents of executable sections *(-d)*
- Display assembly contents of all sections *(-D)*

objdump

- Display the full contents of all sections using *(-s)*
- Display debug information *(-g)*
- Display the contents of dynamic symbol table *(-T)*
- Display source code intermixed with disassembly *(-S)*

Debugging OOPs

- What is the most common bug in kernel?
  - *segfault – oops*
- Kernel exception handler
  - Kills offending process
  - prints registers, stack trace
- Some exceptions are non-recoverable
  - Panic()
- Oops generated by macros:
  - BUG(), BUG_ON(condition)

Debugging OOPs

- Tainted kernels
  - Some oops report contain the string “Tainted: ”
  - “G” if all modules loaded have GPL or compatible license
  - “P” if proprietary module has been loaded
- ***objdump -S char-read-write.ko***

gdb (again)

- *core-file <core_file_name>*
- *add-symbol-file <sym_file_name> <.text base address> -s .bss <sec_addr> -s .data <sec_addr>*
- Need to be educated

kdb

- Can set breakpoints
- Query/Change kernel data
- Single stepping (by instructions, not lines of C source code)
- Disassembling code
- Analysis of kernel state – registers, variables, stack traces
- 2.6.35 – *kdb* merged & uses same backend as *kgdb*

kgdb

- Source level debugger – gdb interface
- Analysis of kernel state – registers, variables, stack traces
- Live Analysis – single step (C source code), breakpoints, threads
- Module debugging
- 2.6.26 – *kgdb* merged into mainline

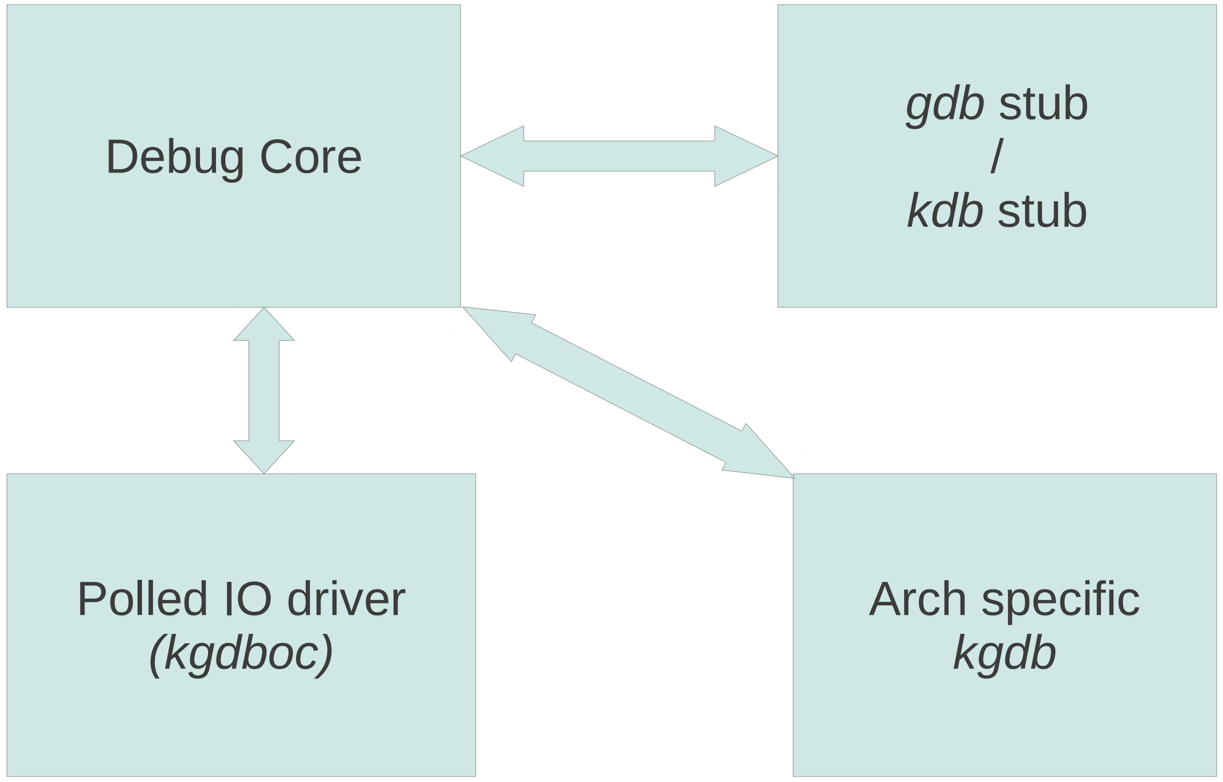
kgdb Setup

- Require two machines – development and test (host and target)
- Requires serial line between the development and test machines
- Test machine runs a *kgdb* enabled kernel
- Development machine runs a copy of *gdb*

kgdb config options

- *CONFIG_DEBUG_INFO=y*
- *CONFIG_FRAME_POINTER=y*
- *CONFIG_DEBUG_RODATA=n*
- *CONFIG_KGDB_SERIAL_CONSOLE=y*
- *CONFIG_HAVE_ARCH_KGDB=y*
- *CONFIG_KGDB_LOW_LEVEL_TRAPS=y*
- *CONFIG_MAGIC_SYSRQ=y*

kgdb Architecture



Debug core

- *kernel/debug/debug_core.c*
- Generic OS exception handler
- API to talk to *kgdb* IO drivers
- API to talk to arch-specific *kgdb*
- Logic to perform safe memory read/write while using the debugger
- Weak Implementation of software breakpoints
- API to invoke kdb/kgdb frontend to debug core

Architecture specific *kgdb*

- *arch/*/kernel/kgdb.c*
- Arch specific trap catcher which invokes *kgdb_handle_exception()*
- *gdb_regs_to_pt_regs(), pt_regs_to_gdb_regs()*
- Registration/unregistration of arch specific handlers
  - Die notifier handling/cleanup
- Hardware breakpoints (optional)

kgdb IO driver

- Configuration via *built-in* or *module*
- Read and write character interface
- Cleanup handler for unconfiguring from the *kgdb* core
- Early debug methodology (optional)
- *kgdb* core repeatedly “polls” *kgdb* IO driver for characters
- *kgdboc, kgdb_8250, kgdboe*

kgdboc

- *kgdboc=[kms][[,]kbd][[,]serial_device][,baud]*
- As a kernel built-in
  - *kgdboc=<tty-device>,[baud]*
- As a kernel loadable module
  - *modprobe kgdboc kgdboc=<tty-device>,[baud]*
- Example: *kgdboc=ttyS0,115200*
- Configure at run-time
  - *Enable: echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc*
  - *Dsiable: echo "" > /sys/module/kgdboc/parameters/kgdboc*

kgdbwait

- *kgdbwait* as a kernel command line argument will stop as early as the IO driver supports
- Useful mainly if you want to set breakpoints in early boot stages
- Can be used only if the IO driver is compiled into the kernel and driver config. is specified as kernel command line argument
  - *kgdboc=ttyS0,115200 kgdbwait*

kgdbcon

- Allows you to see *printk()* messages inside gdb while gdb is connected to the kernel
- Kgdb supports using gdb serial protocol for this
- Config:
  - Kernel Command Line: ***kgdbcon***
  - Using sysfs: ***echo 1 > /sys/module/debug_core/parameters/kgdb_use_con***
- Needs to done before configuring kgdb IO driver

kgdb – Living with optimizations

- Kernel compiled with optimizations
- Each C source line spread over instructions
- Control may appear to go backward in *gdb*
- Line numbers in inline functions make life difficult
- Disable some of the optimizations *(man gcc)*
- Run *objdump -S* on *vmlinux* or *module.ko* to find _exact_ line numbers from instruction pointer

kgdb in action

- *kgdb demo*

References

- kgdb.wiki.kernel.org/index.php/Main_Page
- kernel.org/pub/linux/kernel/people/jwessel/dbg_webinar/
- kgdb.geeksofpune.in/index.html
- geeksofpune.in/files/kerneldebugging-1.pdf
- geeksofpune.in/files/kerneldebugging-2.pdf

About *GEEP (GEEks of Pune)*

- GEEP is a non-profit group intended to create a platform for system software programmers in Pune.
- Founded in 2006 by a few kernel hackers in Pune. Since then it has grown to a more than 350 members group.
- GEEP organizes workshops for software professionals and engineering students. The workshops focus on kernel developments, embedded systems, networking, module programming, kernel debugging and more.
- geeksofpune.in