Linux Kernel & Device Drivers

Chapter 8 - Debugging Mechanisms

strace

- `strace` is a user mode utility that monitors the interaction of user processes with the kernel (*i.e.* system calls).
- `strace` may attach and detach an executing process, or start a process.
- Tracing may be inherited via `fork`'s of traced processes when they spawn child processes.
- `strace` reports parameters passed and values returned by system calls.

strace Example

```
# strace ls
open(".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC)          = 0
getdents64(3, /* 21 entries */, 4096)    = 664
getdents64(3, /* 0 entries */, 4096)     = 0
close(3)                                 = 0
open("/proc/meminfo", O_RDONLY)          = 3
fstat64(3, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x4023d000
read(3, "MemTotal:      6235008 kB\nMemFre"..., 4096) = 771
close(3)                                 = 0
munmap(0x4023d000, 4096)                 = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1),
...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x4023d000
write(1, "bitops.c      copy_page.S  csum-"..., 87) = 87
```

More Features of strace

- The output may be captured to a file (**`-o`** ***`file`***).
- Generate "time-stamped" output (**`-r`**).
- Show the time spent in each system call (**`-T`**).
- Follow **`fork`**'s (**`-f`**).
- Limit tracing to a group of system calls (*e.g.* **`-e trace=file`** - trace only file system related system calls).

Printk

- `printk` is the simplest debug technique.
- It allows reporting information from any place in the code.
- Yet, it has some downsides:
  - Too many messages, make it hard to identify situations for which you are actually debugging.
  - Messages that were printed during a system panic, are likely to be lost. They won't appear in `/var/log/messages` following the reboot.
  - Hard to debug hang conditions.
  - Require a "post debug" cleanup.

/proc

- Many module activities, have a foot print in `/proc`:
  - Device numbers allocation / registration - `/proc/devices`
  - Interrupt handler registration - `/proc/interrupts`
  - Receipt of interrupts - `/proc/interrupts`
  - Registration of I/O ports - `/proc/ioports`
  - Registration of I/O memory - `/proc/iomem`
  - Slab caches - `/proc/slabinfo`
  - More ...

/sys

- Kernel modules have a foot print in `/sys`
  - Module parameters, and their current values - `/sys/module/<name>/parameters`
  - What modules are referencing a module - `/sys/module/<name>/holders`
  - Device classes that are currently defined - `/sys/class`
  - Device that are associated with a class - `/sys/class/<class-name>`
  - More..

The /proc File System

- `/proc` is a collection of kernel functions that are logically laid out in a structure of a file system.
- Each `/proc` entry corresponds to kernel function(s), each responsible for implementing a file system operation (read, write, etc.).
- The functions are called when a `/proc` entry is accessed with a corresponding system call.
- The `/proc` file system does not consume any storage.

The /proc File System - Continued

- The `/proc` file system is a linked list of structures that contain “file” names, access permissions, and pointers to functions.
- A read function that is associated with a `/proc` entry creates a byte stream that is returned, when the `/proc` entry is read from.
- A write function interprets and handles data that is written to a `/proc` entry.
- It is not mandatory to implement both read and write.

Using The /proc Filesystem

- `/proc` provides a wealth of information, in addition to the per-process directories that have already been discussed:
  - Each entry in `/proc/sys` corresponds to a kernel tunable parameter. Writing to entries under `/proc/sys` alters the values of kernel parameters, and the system's behavior accordingly.
  - Module information.
  - File Systems information.
  - Device drivers information.
  - More…

/proc API Change

- The old API for creating, and managing entries in /proc was retired in release 3.10 after 14+ years of faithful service (since release 2.2.26).

- The new API announces the full integration of the proc pseudo file system with the standard Linux file system API.

- The new API associates a `file_operations` structure with each proc file system entry.

- The `file_operations` structure grants the flexibility to expand the limited repertoire that was provided by the old API (only `read` and `write`).

Creating a Proc Entry

```
#include <linux/proc_fs.h>

struct proc_dir_entry proc_create_data (const char *path,
             umode_t mode, struct proc_dir_entry *parent,
             const struct file_operations *proc_fops,
             void *data);
```

- **`proc_create_data`**, creates a pseudo file in /proc that may be used for retrieving module information, and setting of module data.

- **`path`** is the path of the new proc entry that will be created. It may contain '/'es.

- **`mode`** is the creation mode of the proc entry. Symbolic constants for mode are defined in **`<linux/stat.h>`**.

Creating a Proc Entry - Continued

- `parent` is the `proc_dir_entry` of the directory to which `path` is relative. `path` will be created relative to /proc when `parent` is null.

- `proc_fops` is the set of file operations that are applicable to the proc entry.

- `data` is a block of data that will be assigned as this entries private data.

`data` is very useful when a set of proc file operations is to be used with multiple proc entries (*e.g.* in per process directories).

Removing a Proc Entry

```
#include <linux/proc_fs.h>

void proc_remove (struct proc_dir_entry *pentry);
```

- `proc_remove`, removes a proc entry that was previously created.
- `pentry` is the `proc_dir_entry` to remove.

A Mystery Solved

- The old `/proc` API passed `proc_dir_entry->data` as an argument to the reader, and writer functions.

- The new file operations based API lacks this argument. How can `proc_dir_entry->data` be accessed and made use of?

- The long answer:

  - The `proc_inode` structure embeds `VFS`'s `inode` structure, and a pointer to the `proc_dir_entry`.

  - Use `container_of` to acquire a pointer to the `proc_inode`, during open.

A Mystery Solved - Continued

- The short answer:
  - Call `PDE`, from your `open` file operation, and set `filp->private_data` to the returned pointer.

```
#include <linux/proc_fs.h>

static struct proc_dir_entry *PDE(const struct inode *inode)
```

Extending The Proc API

- The proc API supports has several additional operations:
  - `proc_symlink` - Create a symbolic link under /proc.
  - `proc_mkdir` - Create a directory under /proc.
  - `proc_set_user` - Set the user and grop of a proc entry.
  - `remove_proc_subtree` - Remove an entire subtree under /proc.
  - more ...

Exercise

- Extend the net filter that you implemented in the previous chapter to cache the source of the last 50 arp replies.

- As a part of your net filter module, implement a /proc file that will list the MAC addresses of remote hosts, that are currently cached by it.

debugfs

- `debugfs` is a RAM based file system intended for debugging purposes -
  - The use of `/proc` should be limited to process information and management.
  - An implementer of an entry in `/sys` is required to adhere to a policy of "single value per file".
- None of the constraints listed above is imposed on `debugfs`.

debugfs

- `debugfs` first appeared in Linux 2.6.10.
- `debugfs` support is enabled at
  `Kernel hacking->Debug Filesystem`
- Use the following command to mount `debugfs`:

```
mount -t debugfs nodev /sys/kernel/debug
```

- Various kernel modules have built in `debugfs` entries.
- Linux provides a simple API for adding `debugfs` entries.

a Scenario For Using debugfs

- The author of a kernel module may maintain a buffer, in which module events may be recorded:

  - Time stamp
  - PID
  - Module function name
  - Event type (*e.g.* function called, function returned, etc.)
  - Event data (*e.g.* function arguments, return value, etc.)

- A **`debugfs`** entry may be used to represent the data in the buffer in a human readable form.

Creating debugfs Entries

```
#include <linux/debugfs.h>
struct dentry
*debugfs_create_dir(const char *name, struct dentry *parent);
```

- A directory called `name` will be created within the directory described by `parent`.
- The directory will be created under `debugfs`' root when `parent` is NULL.
- `ERR_PTR(-ENODEV)` is returned to indicate that `debugfs` is not configured.

Creating debugfs Entries

```
#include <linux/debugfs.h>
struct dentry
*debugfs_create_file(const char *name, umode_t mode,
                     struct dentry *parent, void *data,
                     const struct file_operations *fops);
```

- A regular file called `name` will be created under `parent`.
- The files access permissions will be per `mode`.
- The operations possible will be defined by `fops`. A minimum of `read`, and / or `write` is required.
- `data` will be assigned to the `i_private` field of the inode, thus making it available to the file operations.

Removing debugfs Entries

```
#include <linux/debugfs.h>
void debugfs_remove(struct dentry *dentry);
void debugfs_remove_recursive(struct dentry *dentry);
```

- **`debugfs_remove`**, removes the entry described by **`dentry`**.
- **`debugfs_remove_recursive`**, removes all the file system sub-structure rooted at **`dentry`**.

eBPF - The Extended Berkeley Packet Filter

- The Berkeley packet filter (AKA `BPF`) originated in 1992 with the purpose of avoiding the flow on un-needed network packets from the kernel to user space.

- `BPF` implements a virtual machine that executes byte code.

- The byte code gets injected from user space, and implements that rules by which packets get filtered.

eBPF - The Extended Berkeley Packet Filter

- The ability to filter packets as early as possible, is extremely useful for network traffic monitoring software, such as `tcpdump` that heavily relies on it.

- `tcpdump` is an application that represents a model in which there are far more packets that are of no interest than packets that require processing.

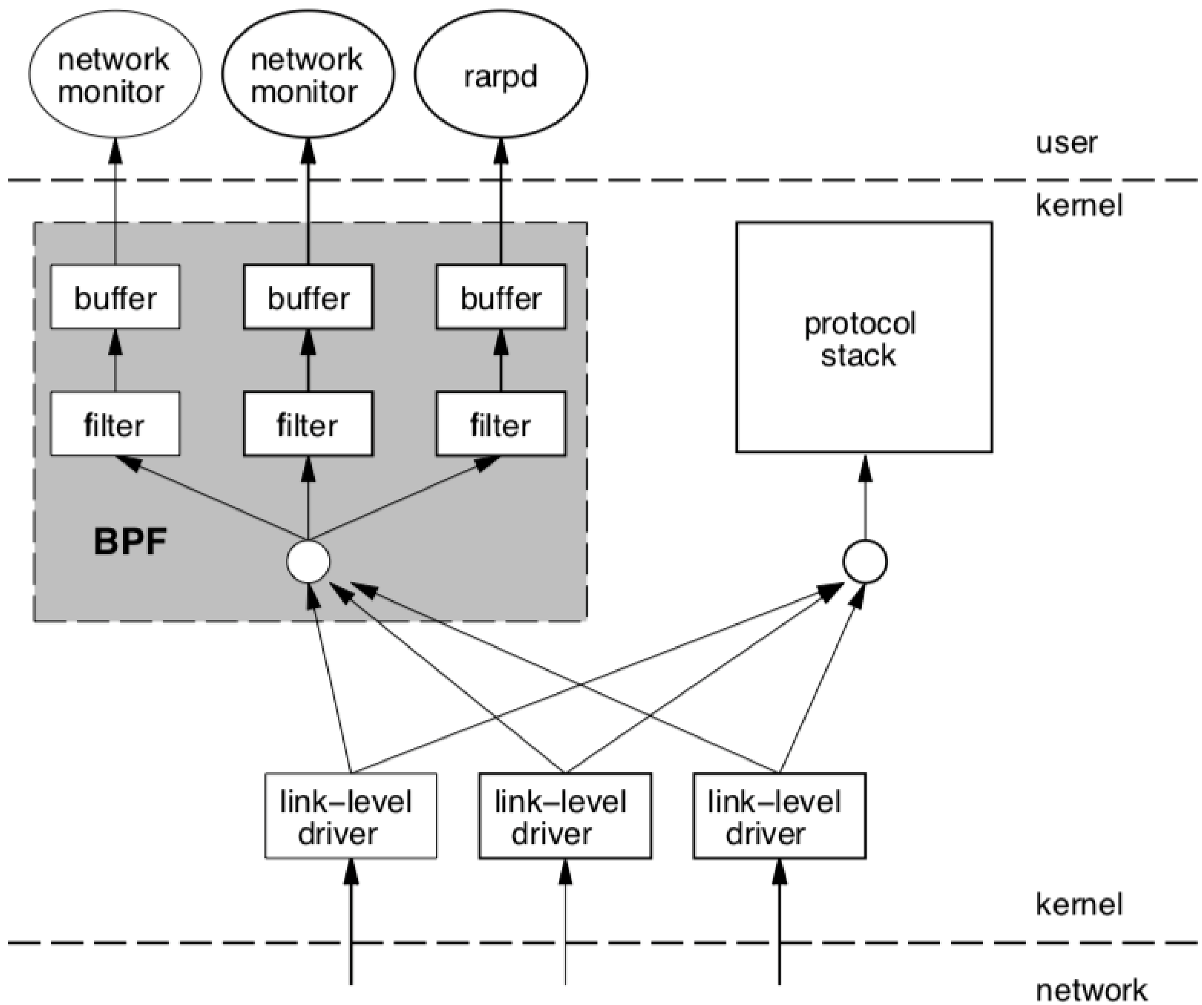
- In such a model, the ability to discard unneeded packets as early as possible saves both CPU and memory resources.

eBPF - The Extended Berkeley Packet Filter

- **BPF** was completely reshaped on 2013 by **Alexei Starovoitov**, to acquire its contemporary sense as an **extended BPF**.

- The original implementation of **BPF** is nowadays known as cBPF – **c**lassic **B**erkeley **P**acket **F**ilter.

cBPF



Michele Di Stephano – Berkeley Packet Filter: Theory, Practice & Perspectves

cBPF Byte Code Example

- Assuming that a packet is an ethernet packet, keep it only if it is an IPv4 packet:

```
(000) ldh     [12];  Load into the accumulator the half word
                  ;  at offset 12
(001) jeq     #0x800        jt 2   jf 3
(002) ret     #262144;      Keep 262144 bytes (enough to
                     ;      accommodate any packet)
(003) ret     #0     ;      Keep 0 bytes
```

- The pseudo Assembly instructions above end up as the following byte code:

```
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 1, 0x00000800 },
{  0x6, 0, 0, 0x00040000 },
(  0x6, 0, 0, 0x00000000 }
```

This code may be generated by macro expansion:
`BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12)`

cBPF Concerns

- Injection of user space code into the kernel rises some concerns such as:
  - The code might read, or write kernel memory beyond the scope of the `sk_buff` that it is supposed to handle.
  - The code might cause a kernel hang, or a kernel crash.
- These concerns are handled by a verifier that examines the filter code to assure that all instructions are valid, there are no backward jumps, and no jumps or memory references that are out of scope.

eBPF

- eBPF made its debut in kernel 3.15.
- It extends the functionality of the cBPF virtual machine to the extent of making it possible to write kernel code beyond packet / system call filtering, while preserving kernel / user space boundaries.
- It allows kernel / user space communication via data structures known as `maps`.

eBPF Architecture

- The number of general purpose registers has been increased to 11 32/64 bit registers.
- The instruction set has been extended to include some real CPU instructions (mostly derived from x86_64).
- eBPF supports multiple program types:
  - BPF_PROG_TYPE_SOCKET_FILTER – Retains the cBPF functionality. It attaches to a socket, and is applied to every `sk_buff` that is associated with that socket.

eBPF Architecture

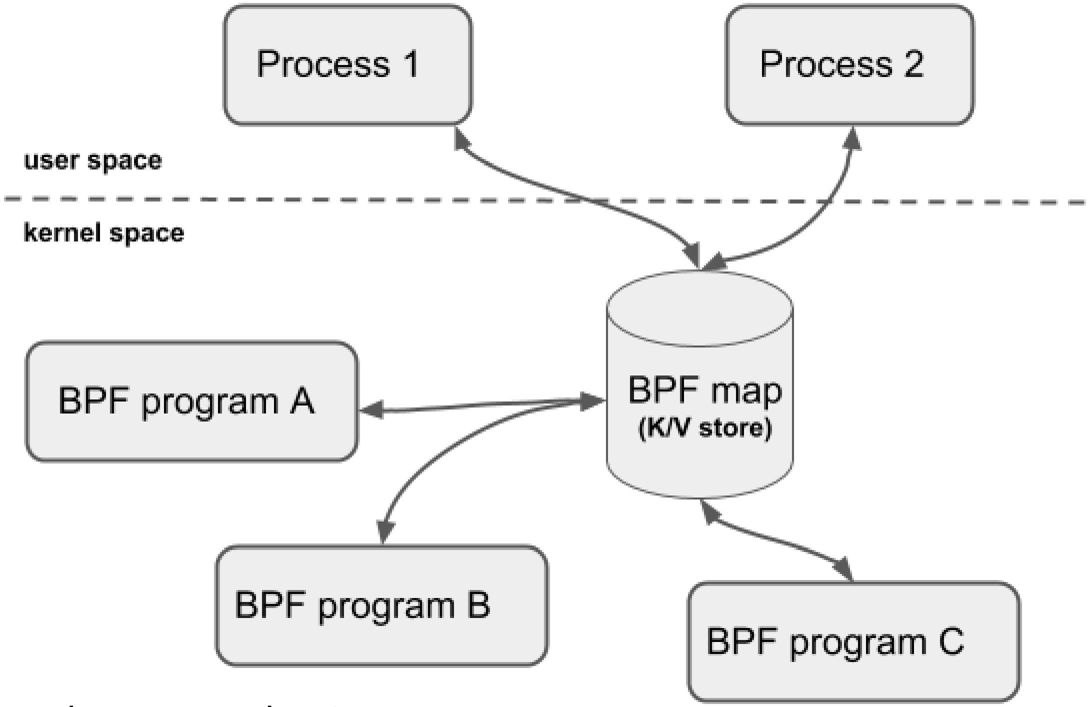
- BPF_PROG_TYPE_XDP – eXpress Data Path. This type of program execute as close as possible to the hardware. It has 3 possible modes of operation:
  - *driver* – Code is executed by the driver. The driver must have XDP support.
  - *generic* – Code is executed at a higher level in the stack.
  - *hardware offload* – Code is executed directly on the NIC

- BPF_PROG_TYPE_TRACEPOINT – Used for kernel instrumentation.

- Helper functions – A subset of the kernel functions that eBPF code may call. The callable functions vary, depending on the program type.

eBPF Architecture

- BPF Maps – Data structures that allow communication between user space programs and eBPF programs.
- BPF maps may be use to maintain states.
- Maps are key, value pairs, that have an arbitrary user defined structure.



Michele Di Stephano – Berkeley Packet Filter: Theory, Practice & Perspectves

Running eBPF Code

- eBPF has concerns, similar to these of cBPF. Hence, eBPF code is verified when it is loaded into the kernel. eBPF has implemented its own verifier that is far more complex than the verifier that was available with cBPF.

- All the operation that relate to eBPF are done with the system call **`bpf`**.

- eBPF programs are written in *restricted C* (a subset of C), and compiled by *llvm/clang* that generates the byte code.

Running eBPF Code

- The user-space code loads the eBPF program into the kernel specifying the program type, which determines accessible kernel's functions.

- The kernel checks if the program is safe through the verifier.

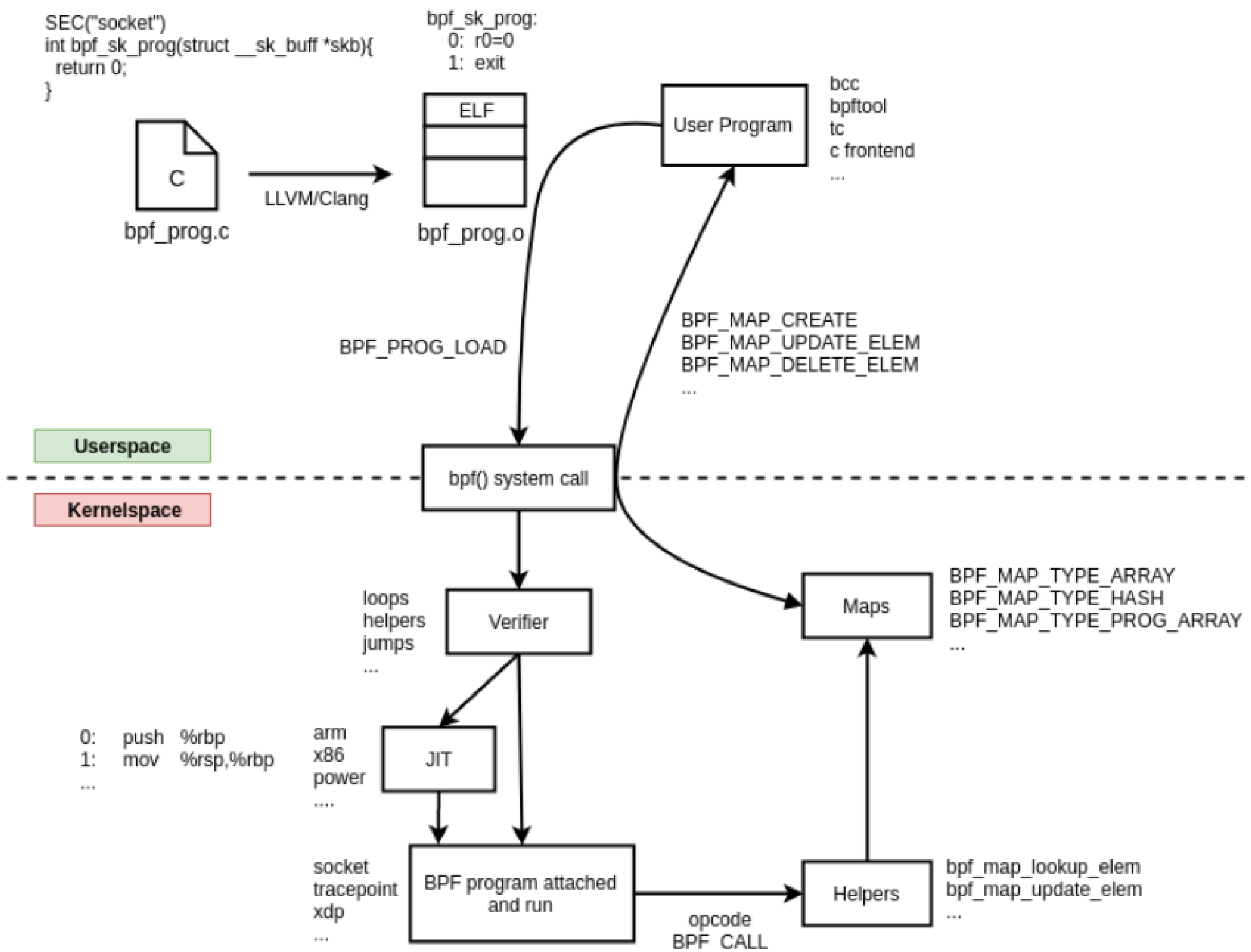
- The kernel, if possible and if it is desirable, JIT-compiles the eBPF bytecode to native machine code, otherwise the program is interpreted.

Running eBPF Code

- The injected code is attached to an event and is executed every time that event occurs.

- The loaded code through the helpers can write data to maps and ring-buffers and the user-space code can read/write from them.

eBPF Usage Overview



Michele Di Stephano – Berkeley Packet Filter: Theory, Practice & Perspectves

eBPF pre-requisites

- The kernel should be build with the following features enabled:
  - `General setup->Enable bpf() system call`
  - ```
    Kernel hacking->
      Compile-time checks and compiler options->
        Compile the kernel with debug info->
          Generate BTF typeinfo
    ```
- Failing to set the later will trigger unexpected program crashes
    ```

Available eBPF Program collections

- The Linux kernel source provides a collection of eBPF programs that were written in restricted C.
  - Compile with – make M=samples/bpf/ (assuming that your in the build directory.
- The bcc collection provide a python module that provides the “glue” between python and eBPF’s C based API, along more that 200 eBPF based tools.

kprobes

- Kernel probes are a method for collecting debugging and performance information at (almost) any arbitrary point in the kernel.
- Kernel probes are dynamically inserted and removed, without touching the code being probed.
- Probe types:
  - kprobe - May monitor any kernel instruction.
  - jprobe - Monitor entry to kernel functions.
  - kretprobe - Monitor return of kernel functions.

Implementing Kernel Probes

- Kernel probes are implemented as kernel modules.
- A kernel probe module:
  - Implements functions that report values of variables, and / or registers (*e.g.* **`printk`**, dedicated **`debugfs`** entries, etc.).
  - Registers the functions to be triggered when a specific code address is reached (module's init function).
  - Un-registers the functions (module's exit function).
- Each probe type has a type specific API, defined in **`<linux/kprobes.h>`**

kprobes - Probing Functions

```
typedef int (*kprobe_pre_handler_t) (struct kprobe *,
                                     struct pt_regs *);
typedef void (*kprobe_post_handler_t) (struct kprobe *,
                                       struct pt_regs *,
                                       unsigned long flags);
```

- The `struct kprobe` argument†, is the structure that was used to register this kprobe. It may be embedded into a structure, accessible via `container_of`, that may provide additional data to the probe.

- The `struct pt_regs` argument, stores the saved values of the CPU registers that were saved, when the probed address was hit.

† Explained later

kprobes - Probing Functions

- The `pre_handler` function is executed before the probed instruction is executed.

- The return value of the `pre_handler` indicates whether:
  - pt_regs were altered - return `1`.
  - Otherwise, return `0`.

kprobes - Probing Functions

- The `post_handler` function is executed after the probed instruction has completed.

- The `flags` argument should indicate the probe's status flags. Yet, it seems that it is not used and `0` is always passed here.

- See `Documentation/kprobe.txt` for additional probing functions.

kprobes - Registration

- Preparing the `kprobe` structure:

```
static struct kprobe kp = {
   .addr = <instruction-address>,
   .pre_handler = my_pre_hdlr,
   .post_handler = my_post_hdlr,
};
```

- Registration:

```
ret = register_kprobe (&kp);
```

  - Returns `0` on success, otherwise a negated error code.

- Un-Registration:

```
unregister_kprobe (&kp);
```

jprobes Entry Function

```
// Function to probe (defined elsewhere):
int probed_func (int i, char c) {
   .   .   .   .
}
// The jprobe entry function:
static int jprobed_func (int i, char c) {
   .   .   .   .
  jprobe_return ();
  return 0;
}
```

- The arguments of a jprobe entry function, have to be identical to these of the probed function.

- The function should always call `jprobe_return()`, to transfer control back to the kernel probes subsystem.

jprobes - Registration

- Preparing a `jprobe` structure:

```
static struct jprobe jp = {
   .entry = jprobed_func,
   .kp = {
      .symbol_name = "probed_func",
   },
};
```

- Registration:

```
ret = register_jprobe (&jp);
```

- Un-Registration:

```
unregister_jprobe (&jp);
```

kretprobe - Entry & Return Functions

```
int *kretprobe_handler_t
                        (struct kretprobe_instance *,
                         struct pt_regs *);
```

- The kretprobe entry function is called when the probed function is entered.
- The kretprobe return function is called when the probed function returns.
- The **`struct kretprobe_instance`** communicates data between the entry and return functions.
- The **`struct pt_regs`** is the set of saved CPU registers.

kretprobe - kretprobe_instance

```
struct kretprobe_instance {
  struct hlist_node hlist;
  struct kretprobe *rp;
  kprobe_opcode_t *ret_addr;
  struct task_struct *task;
  char data[0];
};
```

- The fields that are relevant to the entry and return functions are:
  - **rp** - The **struct kretprobe**† with which this kretprobe was registered.
  - **task** - The **task_struct** associated with this instance.
  - **data** - Arbitrary data for communication between entry and return functions.

kretprobe - Accessing Instance Data

```
/* per-instance private data */
struct my_data {
        ktime_t entry_stamp;
};

/* Here we use the entry_hanlder to timestamp function entry */
static int entry_handler(struct kretprobe_instance *ri,
                                               struct pt_regs *regs)
{
        struct my_data *data;

        if (!current->mm)
                return 1;       /* Skip kernel threads */

        data = (struct my_data *)ri->data;
        data->entry_stamp = ktime_get();
        return 0;
}
```

kretprobes - Registration

- Preparing a `kretprobe` structure:

```
static struct kretprobe krp = {
   .handler                       = ret_handler,
   .entry_handler                 = entry_handler,
   .data_size                     = sizeof(struct my_data),
   /* Probe up to 20 instances concurrently. */
   .maxactive                     = 20,
   .kp = {
     .symbol_name = "probed_func",
   },
};
```

- Registration:

```
ret = register_kretprobe (&krp);
```

- Un-Registration:

```
unregister_kretprobe (&krp);
```

Kernel Probes - Tips

- When the probed address is not in the beginning if a function:
  - Compile the kernel with "`CONFIG_DEBUG_INFO = y`".
  - Use '`objdump -d -l vmlinux`' to obtain source to object mapping.
- When probes are applied to functions symbol_name may be used to specify the probed function. This requires "`CONFIG_KALLSYMS_ALL`" to be set to "`y`".

Magic sysrq

- Magic SysRq - This is a “magical” key to which the kernel will respond regardless what it is doing.

- Configuring the Magic SysRq:
  - Kernel hacking->Magic SysRq Key sets **`CONFIG_MAGIC_SYSRQ`** to **`Y`**.

- Enabling the Magic SysRq:

```
# echo 1 > /proc/sys/kernel/sysrq
```

- Magic SysRq command may be entered writing to **`/proc/sysrq-trigger`**.

Magic SysRq - Commands

- **`b`** - Request an immediate reboot without unmounting or synching disks.
- **`h`** - Display a help message, that describes all the Magic SysRq commands.
- **`l`** - Request a back trace of all the active CPU's.
- **`s`** - Request an emergency sync of all mounted file systems.
- **`g`** - Enter the kernel debugger.

Kgdb

- Kgdb Is implemented by a stub that behaves like gdbserver
- Kgdb requires a debug host with the following elements:
  - A copy of the kernel image running on the target host.
  - The source tree that was used to build the kernel of the target host.
  - gdb , or a cross debugger (when the target system is an embedded system).
- Kgdb is a source level symbolic debugger.

Kdb

- `kdb` implements a shell on the system that is being debugged.
- `kdb` is accessible via the system's console, and may be redirected to any of the system' serial ports
- `kdb`'s shell implements a rich set of commands.
- `kdb` is an assembly level debugger.

Kdb

- `kdb` is designed to be extensible by developers of modules to suit their module's requirements.

- Details and an example how to add commands to kdb can be found in: `<kernel-root>/samples/kdb/kdb_hello.c`

- `kdb` was contributed to Linux, as a patch, by Sgi on 2002. Yet, it was integrated into the main line Linux kernel in version 2.6.35.

A Sample kdb Session (RPI 2b)†

- Enable the Magic SysRq:

```
# echo 1 > /proc/sys/kernel/sysrq
```

- Set the serial port that the debugger will interact with:

```
# echo "ttyAMA0,115200" >/sys/module/kgdboc/parameters/kgdboc
```

- Use the Magic SysRq to escape into the debugger:

```
# echo g > /proc/sysrq-trigger
[ 7623.603732] SysRq : DEBUG

Entering kdb (current=0xb922bf40, pid 3088) on processor 0 due to Keyboard Entry
[0]kdb>
```

† Raspberry PI 2b EVB

A Sample kdb Session (RPI 2b)†

- Display kdb's help:

```
[0]kdb> help
.    .    .    .
bp              [<vaddr>]             Set/Display breakpoints
bl              [<vaddr>]             Display breakpoints
bc              <bpnum>               Clear Breakpoint
be              <bpnum>               Enable Breakpoint
bd              <bpnum>               Disable Breakpoint
ss                                    Single Step
.    .    .    .
```

- Display the stack trace:

```
[0]kdb> bt
Stack traceback for pid 3088
0xb922bf40     3088     3087  1    0    R   0xb922c288 *bash
.    .    .    .
[<80197258>] (proc_reg_write) from [<8013d1c4>] (vfs_write+0xb0/0x1e0)
[<8013d1c4>] (vfs_write) from [<8013d7b4>] (SyS_write+0x4c/0xa0)
[<8013d7b4>] (SyS_write) from [<8000eac0>] (ret_fast_syscall+0x0/0x48)
```

A Sample kdb Session (RPI 2b)†

- Leave kdb and resume kernel execution:

```
[0]kdb> go
#
```

Panic

- The term panic applies to a condition when the kernel cannot continue execution, due to a severe internal inconsistency.

- If kernel dumps are enabled, the crash dump of the kernel may be used after reboot, to analyze the cause for the panic.

Oops

- An oops may be looked upon as a "civilized panic".
- An oops will occur when a severe condition, that would have otherwise caused a panic, can be associated with a specific process.
- When an oops occurs, the offending process is killed.
- The kernel logs a call trace when an oops occurs. The call trace may be used to debug the cause for the oops.

Handling a Grave Condition

- A grave condition in the context of the kernel, is a condition at which something bad that has no possible recovery, has been detected (*e.g.* a device driver has failed to initialize it's `file_operations` structure).

- The macro `BUG_ON` will assert a condition to check if a grave condition has incurred.

- If the assertion is true, it is assumed that a grave condition has incurred, `BUG_ON` will panic the system.

- Example - panic the system if a pointer is null:
  - `BUG_ON (ptr == 0);`

WARN_ON

- The macro **`WARN_ON`** can be viewed upon as a civilized version of **`BUG_ON`**. It will issue a warning when the assertion succeeds, but will not panic the system.

- Example - issue a warning if no data is available:
  - **`WARN_ON (data_avail == 0);`**