KERNEL DEBUGGING

Introduction to Kernel Debugging

!! Welcome !!

Outline

- What is Kernel Debugging
- Need of Kernel Debugging
- Methods of Kernel Debugging
  - By Printing
  - By Querying
  - By Watching
  - Debugging System faults
  - By Debuggers

What is Kernel Debugging?

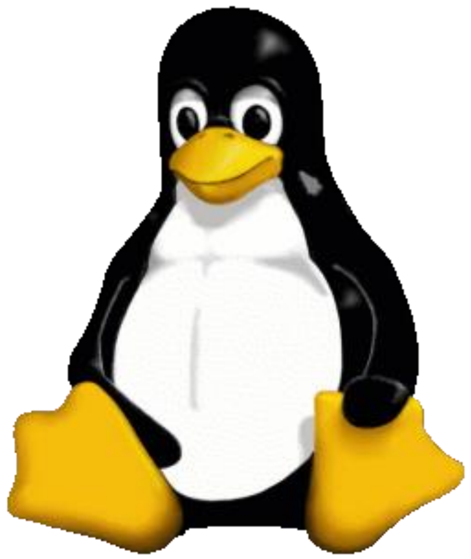
Kernel

The **kernel** is a computer program that is the core of a computer's O/S, with complete control over entire system. As Kernel is the most important part or core part of an O/S it must need to be completely functioning for proper functioning of O/S.

Kernel Debugging

It is the technique of finding and removing the errors in the kernel code so as to avoid any panic condition in operating system like lockup, reboot, crashes, etc.

Why kernel debugging ?



Kernel code:

→ Not always executed in context of a process

→ Cannot be easily executed under a debugger

→ Cannot be traced easily;
because it is a set of functionalities not related to a specific process

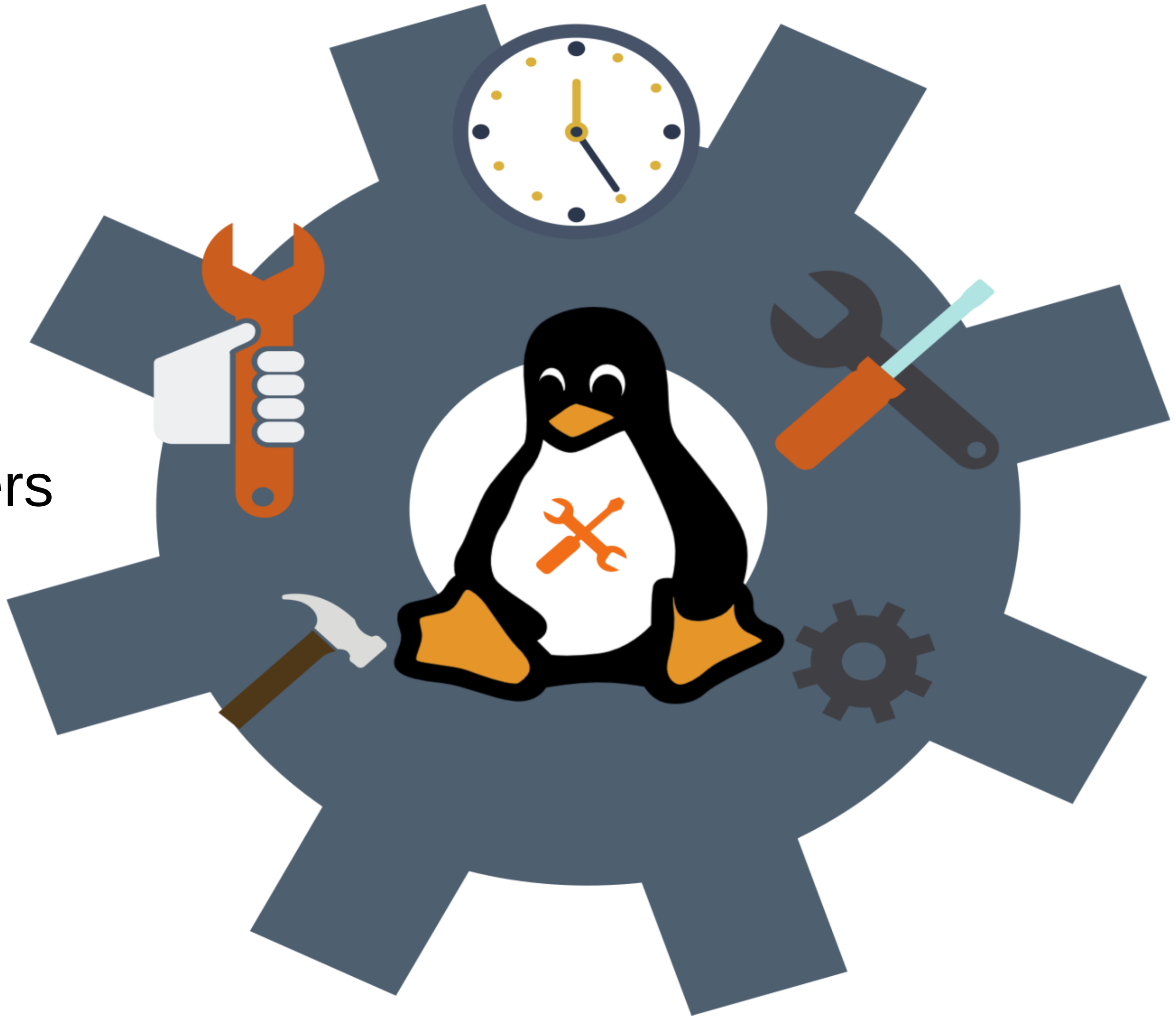
→ Errors can also be exceedingly hard to reproduce and can bring down the entire system with them, thus destroying much of the evidence that could be used to track them down.

→ Bugs in kernel code being developed frequently result in a lockup or a reboot.

→ It's difficult to locate the problem – entry points

How to debug Kernel:

Debugging By Printing
Debugging By Querying
Debugging By Watching
Debugging system faults
Debugging using Debuggers



Debugging by Printing

The most common debugging technique is monitoring, which in done in programming applications by calling printf at suitable points. When you are debugging kernel code, you can accomplish the same goal with printk.

We cannot use printf since it uses c-library support so we use printk, for printing logs.

printk lets you classify messages according to their severity by associating different loglevels.

There are eight possible log-level strings, defined in the header <linux/kernel.h>

We list them in order of decreasing severity:

KERN_EMERG
Used for emergency messages, usually those that precede a crash.
KERN_ALERT
A situation requiring immediate action.
KERN_CRIT
Critical conditions, often related to serious hardware or software failures.

Debugging by Printing .. (continue)

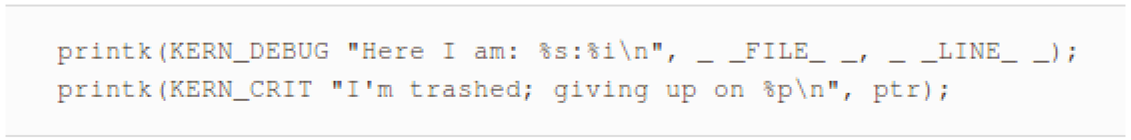
KERN_ERR
Used to report error conditions; device drivers often use KERN_ERR to report
hardware difficulties. Used for debugging messages.

KERN_WARNING
Warnings about problematic situations that do not, in themselves, create serious
problems with the system.

KERN_NOTICE
Situations that are normal, but still worthy of note. A number of security-related
conditions are reported at this level.

KERN_INFO
Informational messages. Many drivers print information about the hardware they
find at startup time at this level.

KERN_DEBUG
Used for debugging messages.



```
printk(KERN_DEBUG "Here I am: %s:%i\n", _ _FILE_ _, _ _LINE_ _);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

Example of usage of printk in kernel code

It is also possible to read and modify the console loglevel using the text file /proc/sys/kernel/printk.

Debugging by Printing .. (continue)

Advantages of using printk:

It is very simple and easy to use, Different log levels make debuggin easy.
Message can be dump to various log levels.

Disadvantages of using printk:
A massive use of printk can slow down the system noticeably, even if you lower
console_loglevel to avoid loading the console device, because syslogd keeps syncing its output files; thus, every line that is printed causes a disk operation. This is the right implementation from syslogd 's perspective. It tries to write everything to disk in case the system crashes right after printing the message; however, you don't want to slow down your system just for the sake of debugging messages.

But, i dont wanna slow down my kernel
now what....??

Try Debugging by Querying

Debugging by Querying

More often than not, the best way to get relevant information is to query the system when you need the information, instead of continually producing data. In fact, every Unix system provides many tools for obtaining system information: ps, netstat, vmstat, and so on.

A few techniques are available to driver developers for querying the system:

creating a file in the /proc filesystem

using the *ioctl* driver method

Using the /proc Filesystem

The /proc filesystem is a special, software-created filesystem that is used by the kernel to export information to the world.

Each file under /proc is tied to a kernel function that generates the file's "contents" on the fly when the file is read.

/proc is heavily used in the Linux system. Many utilities on a modern Linux distribution, such as ps, top, and uptime, get their information from /proc

Implementing files in /proc, All modules that work with /proc should include <linux/proc_fs.h> to define the proper functions.

Debugging by Querying (continue ...)

Using the *ioctl* Method

ioctl, is a system call that acts on a file descriptor; it receives a number that identifies a command to be performed and (optionally) another argument, usually a pointer.
As an alternative to using the **/proc** filesystem, you can implement a few ioctl commands tailored for debugging.
These commands can copy relevant data structures from the driver to user space where you can examine them.
more difficult than using /proc, because you need another program to issue the *ioctl* and display the results. This program must be written, compiled, and kept in sync with the module you're testing.
Then Why ..??
Because, There are times when ioctl is the best way to get information, because it runs faster than reading /proc.
It retrives the data in binary from, retrieving the data in binary form is more efficient than reading a text file.

Debugging by Watching

My favorite

Debugging by Watching

Sometimes minor problems can be tracked down by watching the behavior of an application in user space

There are various ways to watch a user-space program working.

You can run a debugger on it to step through its functions.
Add print statements.
Run the program under *strace*.

Using *strace command*

The strace command is a powerful tool that shows all the system calls issued by a user-space program.
Not only does it show the calls, but it can also show the arguments to the calls andtheir return values in symbolic form. When a system call fails, both the symbolic value of the error (e.g., ENOMEM) and the corresponding string (Out of memory) are displayed.
strace prints tracing information on stderr.
strace receives information from the kernel itself. This means that a program can betraced regardless of whether or not it was compiled with debugging support.

Debugging System Faults

Debugging System Faults

Even if you've used all the monitoring and debugging techniques, sometimes bugs remain in the driver, and the system faults when the driver is executed. When this happens, it's important to be able to collect as much information as possible to solve the problem.

Oops Messages

Most bugs show themselves in NULL pointer dereferences or by the use of other incorrect pointer values. The usual outcome of such bugs is an oops message.
An oops displays the processor status at the time of the fault, including the contents of the CPU registers and other seemingly incomprehensible information.
Example of how oops message occurs,

```
Unable to handle kernel NULL pointer dereference at virtual address
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:    0
EIP:    0060:[<d083a064>]    Not tainted
```

Using Debuggers

Sound obivious

Using debuggers

The last resort in debugging modules is using a debugger to step through the code, watching the value of variables and machine registers. This approach is time-consuming and should be avoided whenever possible.

Using gdb

The debugger must be invoked as though the kernel were an application. In addition to specifying the filename for the ELF kernel image, you need to provide the name of a core file on the command line.

A typical invocation of gdb looks like the following:

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

From within *gdb*, you can look at kernel variables by issuing the standard *gdb* commands. For example: *p jiffies* prints no. of clock ticks from system boot to current time.

Using debuggers Contd…

Numerous capabilities normally provided by *gdb* are not available when you are working with the kernel.

For example, *gdb* is not able to modify kernel data;
It expects to be running a program to be debugged under its own control before playing with its memory image.
It is also not possible to set breakpoints or watchpoints, or to single-step through kernel functions.

So we introduce,

Using kdb (Linus Style)

According to Linus, interactive debuggers lead to poor fixes, those which patch up symptoms rather than addressing the real cause of problems.

Once you are running a *kdb*-enabled kernel, there are a couple of ways to enter the debugger. Pressing the Pause (or Break) key on the console starts up the debugger. *kdb* also starts up when a kernel oops happens or when a breakpoint is hit.

Note that just about everything the kernel does stops when *kdb* is running.

References

Linux Device Drivers, 3rd Edition by Greg Kroah-Hartman,
Alessandro Rubini, Jonathan Corbet

https://www.safaribooksonline.com/library/view/linux-device-drivers/0596005903/ch04.html

!! Thank You !!