

Getting started with Linux:
Debugging

Shreshtha Kumar



Getting started with Linux: Debugging

Introduction:

Programming in Linux for the first time, unable to find F10, F11 and visual breakpoints? So is this job looking tedious and frustrating? You might say yes, reason being, no one is perfect thus introduces bug of some kind or other in the program and without knowing the platform in which you are going to work, debugging becomes really cumbersome. So in Linux platform some pre-coding thought and "implementation based on it" is required.

Here are few techniques and thoughts which can speed up debugging, minimize errors and make you (newbie) comfortable in Linux. Although instead of exhaustive explanation, pointers are given and you need to explore each tip and methodology before using. It is assumed that you have already explored how projects are managed in Linux using Makefile and proper directory structure. This discussion is bit oriented towards writing device drivers in Linux.

Give a thought from debugging perspective before you start

You must be inspecting the system from every aspects before coding (am I talking of designing). All the aspects which will lead you to the goal, assuming ideal bug free path. Path will be ideal, remains in your mind either consciously or unconsciously. You work and if at all success is reached without any bug, although rare case, there remains little space for addition and up gradation in your code. It becomes tedious to manage after code is modified to add some feature.

To keep your code continue as good base and live long some additional aspects must accompany your thought process. Good base and live long means that your code remains in the project, blends easily with code up gradation to fulfill new requirements and reveals its state and configuration inside the system. For example I2C initialization module should reveal its clock frequency etc.

This additional thread is "how to debug if problem crops up?".

For example if you are writing a driver for a device which have only write-only registers. If you directly jump to coding, which is the most possible case with a newbie. You will straight away use the values received from the application to update the device registers directly. But if you have "how to debug" in mind then your driver will have a set of shadow registers. And this is the hidden gem which will later make your life much easier.

Programming considerations assisting debugging

As our main goal is bug free program during its continual development period, so in this perspective you should program in such a way that

- 1) Shields itself from probable bugs, for example using prototypes.
- 2) Help you to concentrate only on few subroutines on occurrence of particular class of bugs/errors.
- 3) Is easily understood by debugger and compiler you are going to use.
- 4) Adapts to any new requirement or new module integrate easily with it with-out/minor change in framework (read program flow).
- 5) Is simple enough to be understood by others easily.

What returns is important - check every time.

Following golden rule can be applied every where

(coz assumptions in the community of uncountable minds working on same bulk code are like hidden mines)

Caution Always, always, check return results.

Additionally check returned/manipulated value with expected one or in expected range. And this is like an extension to the above concept. For example if you request buffers from driver. Then above concept can be accomplished by use of following function.

assert (video_buf.index < n_buffers);

This aborts the program if assertion is false.

But make sure that you insert a debug statement between the function call and assert. Why?

Group all the bug producing areas logically in to subroutine and single master for controlling it. For example memory allocation and freeing always remained a potential member for producing bugs. So restrict its existence in only few blocks of program. Its always better to write your own custom memory alloc/free functions depending on application needs. Keep very strict checking inside such functions. This will surely ease the way you do memory handling.

If you intend to use any debugger then understand the tool. For example if your tool is GDB. Better use enum rather than #define for any #define names with constant integer values. As GDB knows about enumeration constants and will assist during debugging. But avoid obscure programming language constructs and reliance on language-specific rules. For example avoid using operator precedence rules. Instead force precedence by use of parentheses, so that meaning remains clear to other programmers.

Make your code flexible (as much as time constrains allows you).

So that you can stretch/patch your code later on discovery of any bug.

Good design always seamlessly accepts any change, however big it is, which suffice its primary goal. It assists you in any new addition and modification without introducing any major changes and complexity. Then your final program flow will be as easy as what you started with. For example if parameters to any subroutine is minimized by grouping logical data in to structures then

introduction of any new member can be done with minimal changes. There won't be any change in program skeleton and subroutine interface. If at any time a bug arises, it won't be difficult and time-consuming to track down and pinpoint the actual source of the bug.

Modular programming and good coding style assist debugging. If a program is modular, you need to rebuild only revised modules. Make modules simple, with well-defined interfaces and specific in nature. This increases the reusability of code pieces. Additionally, it eliminates any possibility of passing wrong parameter types or order of parameters. Subroutines should be short enough so that their full meaning can be grasped as a unit by the reader. In short, follow "Keep It Small and Simple".

Increase in program size results in scarcity of names. Naming conventions help enormously to identify the variables or functions' existence. Better declare all the variables as static and to use well-defined prefixes for symbols (usually module name).

An informal reading can surface out the most common errors in the code with above guidelines. Misspelled words (especially those words which are valid words for Linux) are also most common errors. This can be averted most of the time by copy-paste (some people may think opposite). Overall, these things make your program more robust. As each is a complete topic in itself, so not deviating from the topic any further lets move to first most-loved and effective tip.

Custom Printf - Lifeline of Linux

I support the heading by the fact that there is presence of -v (verbose) option in every program of Linux. It helps, especially in case when your program is neither crashing nor working properly. It helps to localize your bug. Debugging will become easier if the creator carefully explains what's going on and print can be a good way. But you never want your program to produce noisy messages every time. Not at least in the release version and when program is working fine. Simple print statements won't help in this case. Following custom print() statements which can be switched on and off would be most appropriate. Along with it, it doesn't pose extra load to the code size when switched off.

Following is not only for kernel code but it can also be very helpful in multi-threaded applications. The sequence of messages can help in understanding how threads are switching. But don't form it as basis for taking any decision. As printf statements are never of high priority and anything can happen to the sequence in the background fifo.

```
#define _D_enter() printk(KERN_INFO "%s ::ENTER::", __FUNCTION__)\n#define _D_exit()  printk(KERN_INFO "%s ::EXIT::", __FUNCTION__)
```

Insert `_D_enter()` and `_D_exit()` at proper place in functions. Then program will disclose the actual sequence of its execution.

Program do have repeated conditional blocks in a program.
To know where program execution has reached following can be helpful.

```
#define _D_chek(x...) printk("%s: %d", __FUNCTION__, __LINE__, x)
```

And if you are working with multiple files `__FILE__` is also available.

Although for large and complex programs `fprintf()` are uses as follows

```
fprintf(stderr, "going to call device with %d\n", arg);
```

Combine these custom printf's can be under one define flag or multiple define flags based on priority.

```
#ifndef _D1
#define _D_level1(x) printk(KERN_INFO "DEBUG1: "x) [CODE - Courier New]
...
#else if _D2
#define _D_level2(x) printk(KERN_INFO "DEBUG2: "x) [CODE - Courier New]
...
#endif
```

During compilation time debug with particular level can be enabled using `-D` as follows

```
gcc -D _D1 -o complex_prog complex_prog.c
gcc -D _D=2 -o complex_prog complex_prog.c /* If numbers are used */
```

By the way where are you going to put all these print statements?
Don't put debug message insanely, that could even hamper your debugging or hide errors. For this you have to do some pre-assessment of restrictions posed by different levels of system. Estimate which part to be emphasized vis-à-vis hardware, firmware or application software. Figure out important "function calls" for example blocking functions, memory allocation etc. and kind of software to be written.

Along with above do include a custom printf which routs the stderr messages to `errlog` file. Make sure that you have compiled your program with debugging info ON (`-g`) if you intend to use GDB.

Caution: With all above debug messages do adopt a consistent style of error message. If you adopt some method for naming, indexing or indenting, stick to it throughout your program. Switch off all the debug messages and test your system if your program is controlling/associated with hardware. As these messages also contribute some time in overall execution time of system.

Script makes life easier

Linux beauty and power lies with commands on console. While development some sequence of commands are to be done repeatedly till things are working. All these are also great source of error as it tends to be monotonous. If you use metacharacter "!" to execute previous commands then you are over a ticking time bomb. metacharacter "!" sometimes causes unrecoverable disasters. So better group required sequence of commands and make simple scripts for them. For this create a new file and add following line

```
#!/bin/sh
```

After that you can add all the commands what you used to write in shell.

For example you want to search word "init" in all the i2c related files. As i2c word will be present in the file name, we can do this using small script

```
for i in $(find ./ -name "*i2c*")
do
    grep -P "init" $i
done
```

Browsing the program and kernel code

To understand code flow you have to browse kernel for understanding the flow of any code. ctags [CODE - Courier New] can help greatly by letting you jump directly to the subroutine or structure declarations. It must be kept in mind that browsing the kernel itself is a very good job. It is the best way to understand the kernel directory structure, which is very important. So better use ctags [CODE - Courier New] after you are quite through with the directory structure.

To use ctags use following command.

This generate tag files for source code in the directory you want (generally kernel source directory).

```
ctags -R
```

Once you are done with the tagging use it with vi editor. For example to search for tvp5146_ctrl() function use following command

```
vi -t tvp5146_ctrl
```

```
grep -r "fn2find"
find -name "filename"
```

-r stands for recursively searching inside sub directories and above can be used with wild cards for example "*video*". Create a directory named grep and store result with same file name as search string. This helps as some times result of grep exceeds the buffer size of console. Open the saved file

in vi editor and use string search in that file. For example if you want to see all the files related to or modified for davinci use (hoping that writer do leave a tag after modifying original code or creating a new one)

```
grep -r davinci * > /home/shreshtha/grep/davinci.txt  
vi davinci.txt
```

Search any string in vi using string after "/" (e.g. /davinci) and pres 'n' for next.

If you find above cumbersome and want instantaneous result then use combinations and grep to filter out as required for example

```
ls -ltr "*.c" | grep "test"
```

This group of command will show all C files but with test in their name and most recent modified at the last.

Any function's definition, parameter or return values you want to know just use manual available with Linux. Do use it the moment smallest doubt comes to your mind related to param type or return value.

```
man function
```

Main purpose behind this topic is that you browse the code faster in a streamlined manner. As faster you browse the source code more changes are there that you try out more and more possibilities and reach the bug faster.

Some basic tool at your service

Tools in Linux are small yet powerful programs which can perform a specific task. They exist in abundant and in a best usable form. So that we need not to reinvent wheel every time we have to write a big application. Only work you have to do is to search for right tools which suits you and your work. Tools can be also be combined using pipes to form powerful strings of commands as shown in previous topic. Few are presented here

```
strace ./myprog
```

It intercepts, records and prints the system calls which are called by a process and the signals which are received by a process. Other tools that help in debugging are

"gprof" produces an execution profile of C programs. It includes the time spent in each function, how many times the function was called etc.

"gcov" It is a test coverage program. It can be used to find whether program execution skips a important part of code. Limitation is that programs must be compiled with the GNU gcc compiler.

other tools such as "valgrid" detect many memory

management and threading bugs.

These tools gives you various types of information for your entire program. This ample data can assist you in quickly seeing the overall system state so that you can figure out if any parameter is abnormal. But in my view these are firefighting methods and you should mostly rely on practicing caution in advance.

Conclusion

Although many advanced function in C and commands in Linux are available but I feel keeping the code as simple as we can and not using the functions we are not well acquainted make debugging much easier. Presence of debug messages enable programmer to crawl through the program and helps in understanding the program flow much faster. Understanding your debugger tool so as to enable your code for debugging is also important. Finally three things, if kept in mind while writing code can make it less buggy and debugging easily and fast

- * Think of from debugging perspective before writing.
Think of how to available tools can be employed for debugging.
- * Good combinations of above small yet powerful commands and defines are sufficient for debugging simple applications and even complex device drivers.
- * Search for "standard way" in Linux community for the thing you are about to code.

Thinking inline with these tips will surely improve the quality and maintainability of the software