CSE/IT 113: Linux Kernel Coding Style

In this class, we follow the Linux Kernel Coding Style, with one exception we use spaces rather than tabs for indentation. Below is some relevant sections from their Coding Style. You can find the complete text at

`https://www.kernel.org/doc/Documentation/CodingStyle`.

Also some formatting has been changed to reflect a pdf rather than an ASCII text document.

Linux kernel coding style

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't *force* my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.

First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

Chapter 1: Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of π to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer

to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

The preferred way to ease multiple indentation levels in a switch statement is to align the "`switch`" and its subordinate "`case`" labels in the same column instead of "double-indenting" the "`case`" labels. E.g.:

```
        switch (suffix) {
        case 'G':
        case 'g':
                mem <<= 30;
                break;
        case 'M':
        case 'm':
                mem <<= 20;
                break;
        case 'K':
        case 'k':
                mem <<= 10;
                /* fall through */
        default:
                break;
        }
```

Don't put multiple statements on a single line unless you have something to hide.

Don't put multiple assignments on a single line either. Kernel coding style is super simple. Avoid tricky expressions.

Outside of comments, documentation and except in Kconfig, spaces are never used for indentation. ***NB*: In this class, we use spaces rather than tabs. Make sure to set your editor to replace tabs with spaces and the tab width to 8**.

Get a decent editor and don't leave whitespace at the end of lines.

Chapter 2: Breaking long lines and strings

Coding style is all about readability and maintainability using commonly available tools.

The limit on the length of lines is 80 columns and this is a strongly preferred limit.

Statements longer than 80 columns will be broken into sensible chunks, unless exceeding 80 columns significantly increases readability and does not hide information. Descendants are always substantially shorter than the parent and are placed substantially to the right. The same applies to function headers with a long argument list. However, never break user-visible strings such as `printk` messages, because that breaks the ability to `grep` for them.

Chapter 3: Placing Braces and Spaces

The other issue that always comes up in C styling is the placement of braces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the preferred way, as shown to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:

```
1       if (x is true) {
2               we_do_y
3       }
```

This applies to all non-function statement blocks (`if`, `switch`, `for`, `while`, `do`). E.g.:

```
1       switch (action) {
2       case KOBJ_ADD:
3               return "add";
4       case KOBJ_REMOVE:
5               return "remove";
6       case KOBJ_CHANGE:
7               return "change";
8       default:
9               return NULL;
10      }
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
        body of function
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are *right* and (b) K&R are right. Besides, functions are special anyway (you can't nest them in C).

Note that the closing brace is empty on a line of its own, *except* in the cases where it is followed by a continuation of the same statement, i.e. a "`while`" in a `do`-statement or an "`else`" in an `if`-statement, like this:

```
        do {
                body of do-loop
        } while (condition);
```

and

```
        if (x == y) {
                ..
        } else if (x > y) {
                ...
        } else {
                ....
        }
```

Rationale: K&R.

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on.

Do not unnecessarily use braces where a single statement will do.

```
        if (condition)
                action();
```

and

```
        if (condition)
                do_this();
        else
                do_that();
```

This does not apply if only one branch of a conditional statement is a single statement; in the latter case use braces in both branches:

```
        if (condition) {
                do_this();
                do_that();
        } else {
                otherwise();
        }
```

3.1: Spaces

Linux kernel style for use of spaces depends (mostly) on function-versus-keyword usage. Use a space after (most) keywords. The notable exceptions are `sizeof, typeof, alignof`, and `__attribute__`, which look somewhat like functions (and are usually used with parentheses in Linux, although they are not required in the language, as in: "`sizeof info`" after "`struct fileinfo info;`" is declared).

So use a space after these keywords: `if, switch, case, for, do, while` but not with `sizeof, typeof, alignof, __attribute__`

For example,

```
s = sizeof(struct file);
```

Do not add spaces around (inside) parenthesized expressions. This example is *bad*:

```
s = sizeof( struct file );
```

When declaring pointer data or a function that returns a pointer type, the preferred use of '*' is adjacent to the data name or function name and not adjacent to the type name. Examples:

```
1         char *linux_banner;
2         unsigned long long memparse(char *ptr, char **retptr);
3         char *match_strdup(substring_t *s);
```

Use one space around (on each side of) most binary and ternary operators, such as any of these:

```
=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ? :
```

but no space after unary operators:

```
&  *  +  -  ~  !  sizeof,  typeof,  alignof,  __attribute__,  defined
```

no space before the postfix increment & decrement unary operators:

```
++  --
```

no space after the prefix increment & decrement unary operators:

```
++  --
```

and no space around the '.' and "->" structure member operators.

Do not leave trailing whitespace at the ends of lines. Some editors with "smart" indentation will insert whitespace at the beginning of new lines as appropriate, so you can start typing the next line of code right away. However, some such editors do not remove the whitespace if you end up not putting a line of code there, such as if you leave a blank line. As a result, you end up with lines containing trailing whitespace.

Chapter 4: Naming

C is a Spartan language, and so should your naming be. Unlike Modula-2 and Pascal programmers, C programmers do not use cute names like `ThisVariableIsATemporaryCounter`. A C programmer would call that variable `tmp`, which is much easier to write, and not the least more difficult to understand.

HOWEVER, while mixed-case names are frowned upon, descriptive names for global variables are a must. To call a global function `foo` is a shooting offense.

GLOBAL variables (to be used only if you really need them) need to have descriptive names, as do global functions. If you have a function that counts the number of active users, you should call that `count_active_users()` or similar, you should not call it `cntusr()`

Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder MicroSoft makes buggy programs.

LOCAL variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called `i`. Calling it `loop_counter` is non-productive, if there is no chance of it being mis-understood. Similarly, `tmp` can be just about any type of variable that is used to hold a temporary value.

If you are afraid to mix up your local variable names, you have another problem, which is called the function-growth-hormone-imbalance syndrome. See chapter 6 (Functions).

Chapter 6: Functions

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it than you would have done).

Another measure of the function is the number of local variables. They shouldn't exceed 5-

10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

In source files, separate functions with one blank line.

In function prototypes, include parameter names with their data types. Although this is not required by the C language, it is preferred in Linux because it is a simple way to add valuable information for the reader.

Chapter 8: Commenting

Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a comment: it's much better to write the code so that the working is obvious, and it's a waste of time to explain badly written code.

Generally, you want your comments to tell WHAT your code does, not HOW. Also, try to avoid putting comments inside a function body: if the function is so complex that you need to separately comment parts of it, you should probably go back to chapter 6 for a while. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the function, telling people what it does, and possibly WHY it does it.

Linux style for comments is the C89 "`/* ... */`" style. Don't use C99-style "`// ...`" comments.

It's also important to comment data, whether they are basic types or derived types. To this end, use just one data declaration per line (no commas for multiple data declarations). This leaves you room for a small comment on each item, explaining its use.